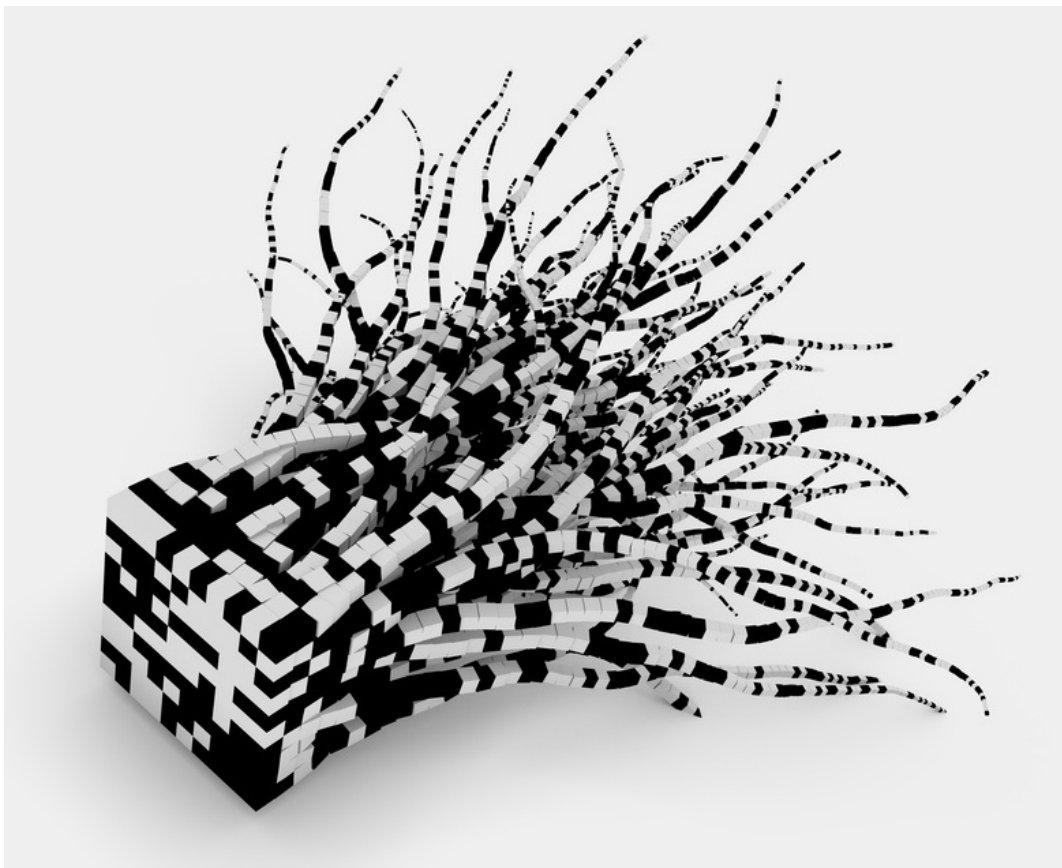


Field Of Threads

Master's Thesis on a tool for clarifying thread behavior.



Bert Dekkers

Field Of Threads

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Bert Dekkers
born in Vlissingen, the Netherlands



Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

© 2011-2012 Bert Dekkers. All rights reserved.

Cover picture: A generated Binary Kite by Syntopia.

Field Of Threads

Author: Bert Dekkers
Student id: 1149733
Email: FOT@bertdekkers.com

Abstract

Multithreaded programming is becoming increasingly important because of recent increase in the use of multiprocessor computing. Multithreaded or concurrent programming is inherently more complex than single threaded code, requiring the use of synchronization and causing possible problems like deadlock or dormancy. Dynamic analysis tools can aid in understanding the execution of concurrent programs and help with improving and debugging of these programs. Current tools express the complexity of concurrent programs with varying degrees of success. We have evaluated and analyzed the visualizations of these tools in relation to the inherent concurrency problems they were meant to solve and come up with an alternative approach to tracing, analyzing and visualizing program executions. By focusing on the use of shared fields we show memory based interaction between threads by use of sequence diagrams, structured in a novel way. This new approach is implemented in a Java based, Eclipse plugin, dynamic analysis tool called Field Of Thread (FOT). FOT provides a new low-level perspective to concurrency oriented dynamic analyses for visualizing shared memory based thread interaction.

Thesis Committee:

Chair: Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor: Dr. A. Zaidman, Faculty EEMCS, TU Delft
Committee Member: Dr. M. Birna van Riemsdijk, Faculty EEMCS, TU Delft

Preface

First I would like to thank my supervisor, Andy Zaidman for his help, guidance and patience throughout this research. I would like to thank the thesis committee and for sparing the time to evaluate this work. Thanks go out to my friends and colleagues that gave valuable ideas and insight: Paul Quist, Stephanie de Jong and all my fellow Software Evolution Research Laboratory students. Finally I would like to thank my parents for their unwavering support through difficult times, allowing me to come this far.

Bert Dekkers
Delft, the Netherlands
March 21, 2013

Contents

Preface	iii
Contents	v
List of Figures	vii
List of Tables	ix
1 Introduction	1
1.1 Objectives	2
1.2 Boundaries and scope	2
1.3 Research Questions	3
1.4 Thesis Structure	3
2 Related Work	5
2.1 Unified Modeling Language	5
2.2 JThreadSpy	7
2.3 JACOT	8
2.4 JIVE	10
2.5 JRET	13
2.6 TPTP	16
2.7 Others	18
2.8 Recap	19
3 Analysis	21
3.1 Problems	21
3.2 Tracing options	23
3.3 Representation options	25
4 Field of Threads (FOT)	29
4.1 Overview	30

CONTENTS

4.2	Visualization	33
4.3	Structure	40
5	Evaluation	45
5.1	Case Study	45
5.2	Functional Comparison	53
5.3	Limitations	57
5.4	Discussion/Reflection	60
6	Conclusions and Future Work	63
6.1	Contributions	63
6.2	Conclusions	64
6.3	Future work	66
	Bibliography	69
A	Glossary	73
B	Related Works Visualizations	76
B.1	UML1.0 Sequence Diagram	76
B.2	JThreadSpy Sequence Diagram of Threads competing for a lock.	77
B.3	TPTP Visualizations	78
B.4	FOT Appendices	80
C	Case Study	85
C.1	Test Program Case 1 Race Conditions	85
C.2	Test Program Case 2 Dining Philosophers	86

List of Figures

2.1	UML2.4 Sequence Diagram with added timing information.	6
2.2	JThreadSpy sequence diagram example.	8
2.3	JACOT Sequence Diagram.	9
2.4	JACOT State Diagram.	10
2.5	JIVE Contour Diagram.	12
2.6	JIVE Sequence Diagram.	13
2.7	Example of the SEQUENCE visualization generator tool.	15
2.8	Example of the SEDIT visualization generator tool	15
2.9	TPTP agent architecture.	16
3.1	Java Platform Debugger Architecture.	23
3.2	JVMTI operation using static instrumentation.	24
3.3	JVMTI operation using library events.	25
4.1	FOT Flow Diagram.	32
4.2	FOT Table Tree view.	33
4.3	Table Tree View Context Menu.	35
4.4	Table Tree View Value Info.	35
4.5	FOT Sequence Diagram view.	36
4.6	Sequence Diagram View Message Description.	37
4.7	Sequence Diagram Relative timings.	38
4.8	Sequence Diagram Preferences Dialog.	39
4.9	FOT Configuration Dialog.	40
4.10	FOT Class Diagram.	41
4.11	FOT Trace Data Structure.	42
4.12	FOT View Data Structure.	43
5.1	FOT sequence diagram of Case 1 Race Conditions.	47
5.2	Dining philosophers scenario.	48
5.3	Case 2 FOT Tree Table.	51
5.4	Case 2 deadlock scenario FOT sequence diagram "taken" field.	52

LIST OF FIGURES

5.5 Case 2 dormancy scenario FOT sequence diagram "state" field. 52

List of Tables

4.1	TPTP Call Tree Relation.	30
4.2	FOT Object Tree Relation.	30
5.1	Case Problem Matchup.	45
5.2	Functional Comparison table.	53

List of Listings

5.1	Case 1 test program Race Conditions output.	46
5.2	Case 2 test program sections Class Ohashi and Philosopher	49
5.3	Case 2 test program Dining Philosophers output.	50

Chapter 1

Introduction

Before the year, 2006 computer-processing units were growing at such a rate that the performance doubled on average every 18 months [3]. Programmers would benefit from this trend, as their program's performance would grow as well. However, this growth rate is no more, the maximum speed for a single processor has reached its maximum at about 3.8GHz because of the physical limitations of transistor size. The last few years have instead seen a steady trend to increase the number of processing units within computers, graphic-accelerators and mobile devices to accelerate processing. This new approach to better performance has many benefits and limitations. Multiple processors are usually used by multiple programs distributed evenly among the processors in a divide and conquer manner. But this does not necessarily speed up a single running program. Lately many programs have been written in a way that they can run multiple processing threads across multiple cores independently, thus speeding up calculation. This type of work division through threads has been in common use for a very long time, but its use in calculation speed acceleration is becoming more vital in software optimization.

As the use of threads within a program increases, so does its complexity. A good analogy would be a company: a one-man business can spend all its time on running the business, but it is limited to what that one man can do. Businesses with many employees require a lot of managing, communication, meetings, common knowledge and protocols to function properly. These activities slow the work down, increase complexity and cause errors to occur. Finding problems and solving them within multithreaded code is just as problematic because of miscommunication, bad timing or unforeseen situations. When the number of employees reaches a certain point the advantage of adding more workers vanishes as the cost of management overhead starts to outweigh their productivity. But if every employee is given a task that can be performed completely independently then the number of employees that can work depends only on how many big jobs you can chop up into independent tasks. Sadly no task is ever completely independent. You have to pass on the results of finished work for instance. The discipline of concurrent parallel programming is the discipline of spreading the workload as efficiently as possible while guaranteeing correctness of the results and dealing with the inherent problems.

There are tools and methodologies to combat the occurrence of these problems and miscommunications. But when these mechanisms are used too often, or incorrectly, it becomes more difficult to spot, diagnose and solve these concurrency problems. Since multithreaded programs are often created by refactoring single threaded programs, these problems often occur. Solving these problems often requires restructuring the program code in counter-intuitive ways. Even causing some problems, that can be solved only if the software is redesigned from the bottom up.

1.1 Objectives

To detect, display, diagnose and solve concurrency based problems there are many tools and models. And there are standard low level paradigms and algorithms. To narrow the rift between the models and the algorithms, we will dissect the current tools and models to analyze their effects and formulate an alternative solution.

The model we will mainly be focusing on is the *Unified Modeling Language* [20] sequence diagram. This diagram usually shows how object oriented classes cooperate, but is sometimes used to show thread communication and behavior. This application is useful but limited in and by itself.

This thesis presents a new model based on the UML sequence diagram shifting the focus from a high-level co-operational model to a lower level, field oriented model.

We elaborate on a tool developed during this research named Field Of Threads (FOT) implementing the field centric model, designed to visualize behavior of running concurrent test programs. Its operation consists of:

- Tracing the execution of a test program.
- Analyzing the inter-thread communication.
- Visualizing the execution in a field-centric way for detecting and diagnosing problems.

1.2 Boundaries and scope

This research will mainly be limited to Java based applications, because of the object-oriented properties of the language and great support for tooling. A lot of previous work has been done using this platform ensuring that older techniques are compatible and the solutions are comparable.

Since the goal is to trace, analyze and visualize concurrent behavior, we will not be focusing much on non-concurrent visualizations, techniques programs and tools.

Dynamic analysis is preferred to static analysis because we do not mean to analyze the design facets of static code, but the dynamic inner-workings of executing programs and their non-deterministic nature during operation.

1.3 Research Questions

The goal of this project is:

“Improving the way running threads, thread functionality and interaction are represented and visualized so that they can be understood more easily.”

In order to reach the goal for this project, we aim to answer the following questions:

RQ1 How are threads visualized and analyzed now?

RQ2 What are the major problems impeding thread comprehension?

RQ3 What kinds of operations are the current representations of threads useful for?

RQ4 What elements provide the best way to view, analyze and comprehend thread execution?

RQ5 How can the representation of threads be improved further?

1.4 Thesis Structure

The research questions will be answered in the following sections of the thesis: Chapter 2, *Related Work*, describes related work in terms of approaches to detect, analyze and visualize concurrent behavior answering the first research question [RQ1]. The third chapter *Analysis* breaks down the operation of current models and tools, and identifies the major problems they are designed to solve. Combined with the inherent and unsolved problems we will be answering the second and third research question [RQ2], [RQ3]. By answering [RQ2] and [RQ3] we will have an approach for answering question four [RQ4] by the end of the third chapter. This formula will be used to create our alternative solution outlined in chapter 4 *Field Of Threads*. The fifth chapter *Case Study* demonstrates and evaluates the alternative solution and its merits in more detail, answering the fifth and final research question [RQ5]. This is followed by chapter 6 *Conclusions and Future Work* where we pose some future problems and solutions, draw conclusions and reflect on this work. This is followed by a bibliography and the appendices, also containing a glossary detailing the more specialized terms.

Chapter 2

Related Work

New solutions in the area of static analysis, dynamic analysis and the visualization of program behavior are constantly being researched. Due to differences in focus, the innovations resulting from this research contribute in varying degrees to a better comprehension of thread executions. This chapter outlines some of the different approaches to monitoring and visualizing behavior of executions in reference to multithreaded programs. We will mainly be mentioning tools that work on the Java platform and preferably those that are already focused on object orientation and concurrency. The tools described in the related works will be evaluated on the way they acquire their information, if it is through tracing or other means. How they make the information comprehensible through visualization by diagrams and possible plus-alpha features that might help in combating concurrency problems. Finally, we will do a short evaluation of the applicability of the related work tools' approaches to solving our problems.

2.1 Unified Modeling Language

The UML *Unified Modeling Language*¹ is a standardization of object-oriented design methods and notations, formulated in the 1990s by the *Object Management Group* [20, 9]. UML is a graphic modeling language used to express program designs and it specifies a possible process to create such a design. UML defines interaction diagrams that model how groups of objects collaborate in some behavior.

In a sequence diagram, as can be seen in appendix [B.1], objects are shown as a box at the top of a vertical line called a *lifeline*. It represents the objects life during the described interaction. Arrows between lifelines are called *messages*. These messages are shown in chronological order; earlier messages are placed higher than later messages. Messages can be called and received by the same lifeline. If an object is active, an *activation box* can be shown overlapping the lifeline. Because the sequence diagram as described by UML is a design specification it can also indicate conditions for deciding whether to send messages or iterative markers for repeating messages. There is some variation in how sequence diagrams

¹UML: <http://www.omg.org/spec/UML/>

2. RELATED WORK

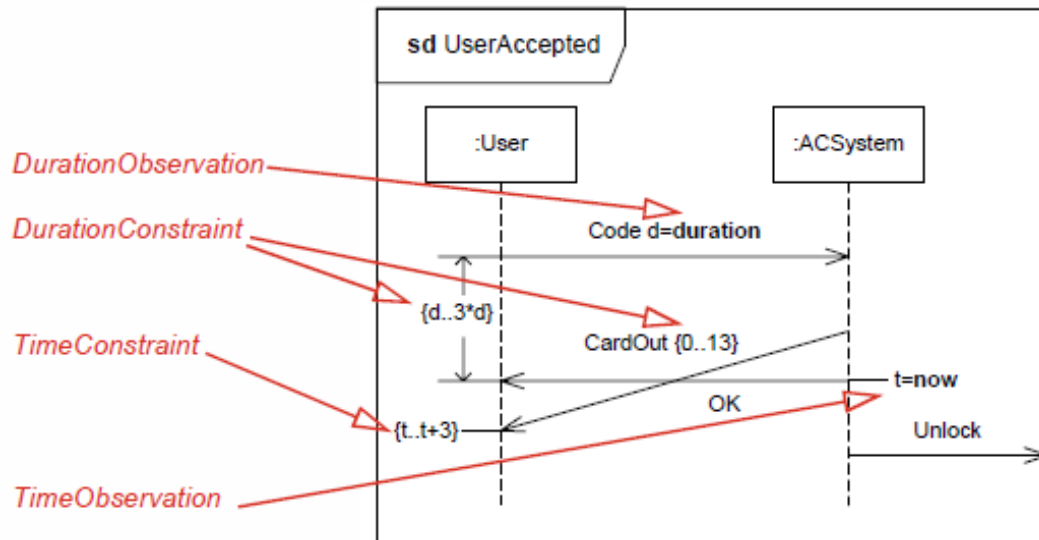


Figure 2.1: UML2.4 Sequence Diagram with added timing information.

are used in practice. A very simple sequence diagram as just described can be augmented by icons and notations indicating reply messages, synchronous or asynchronous messages, intentional diagram omissions, ends of lifelines, etc.

We investigate UML sequence diagrams because UML is an accepted standard in the industry and academia. Interaction scenarios in UML are best depicted by sequence and collaboration diagrams. Sequence diagrams are superior at depicting states compared to other communication diagrams in terms of expressivity. The downside to sequence diagrams is that they only display one scenario at a time, obfuscating the surrounding context. This is not a problem if only one specific scenario is needed or the surrounding context has been made clear in other ways. The UML sequence diagram designates active objects and execution specifications, which depict the life of an object and its method activations, by callers. A sequence diagram models each object as an active environment. It depicts an abstract view of the dynamics of synchronization and the ordering of interactions.

The current UML2.4 specifications² offer little support for thread-level synchronization. In the current specifications, additions have been made to the sequence diagram model [20]. The addition of timing information has been specified, as can be seen in figure 2.1. The state and interaction overview diagrams that are used describe any added complications, but concurrency is still not sufficiently modeled to solve our problems [29]. Since our ultimate objective is to visualize the execution state of a running application. When viewing the UML specification in this light some of the more advanced features, for example conditions and iterations, are hard to derive correctly from execution tracing. On the other hand, more

²UML2.0 specification: <http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF>

references to the current execution may be required to link the abstract UML model to the execution environment and the code definition, so that the model can be put into the proper context. For instance, threading and monitor information in the sequence diagram would not only show what is happening, as the model does, but would also indicate why the environment does it this way. This is what we will need to improve understandability of the model.

2.2 JThreadSpy

JThreadSpy is an Eclipse plug-in developed by Malnati et al. [16, 17] created to teach object-oriented programming and illustrates the difficulties of concurrency to students. It is a dynamic analysis tool that creates a sequence-diagram visualization from program execution traces for each thread. Its focus is on understandability of concurrency. JThreadSpy features added visualization items to a standard UML sequence diagram and a variable detail scaling display.

2.2.1 Tracing

JThreadSpy uses an agent with a JVM-TI [22] [3.2.2] interface and an *ASM framework binary interpreter* [13] to dynamically instrument Java class files during class loading inside the Java Virtual Machine and generate trace logs. Instrumentation is done by the Java agent, which implements the JVM tooling interface to add extra lines of code to some places in the program code. The lines inserted are for example: a log output writer at every start and end of a method. This writer registers and logs the entry and exit of each thread into and out of each method. The ASM framework code is necessary to understand and modify code that is already in binary form, so the proper lines can be inserted in the proper places. Only the classes of the application being compiled are instrumented. JThreadSpy cannot show any information about for example libraries outside the scope of the subject application.

2.2.2 Visualization

The traces created by the agent are interpreted by a visualizer to generate the UML sequence diagrams. The UML-based sequence-diagram shows all objects not at the top of the diagram but at their execution moment, which means an object instance is shown lower than the main executing class. The main class is also shown in a balloon while other classes are shown as a rectangle. JThreadSpy adds detailed thread information. Every thread is shown in a different color as a message and activation sequence running over the objects' lifelines. Figure 2.2 is the JThreadSpy output of an example case, depicting money transfers between bank accounts. A more elaborate case about threads competing for a lock is included as appendix B.2. In JThreadSpy new concurrency specific operation icons have been added to the activation bars indicating; *thread starts*, *thread stops*, *requesting locks*, *acquiring locks* and *releasing locks* on shared objects.

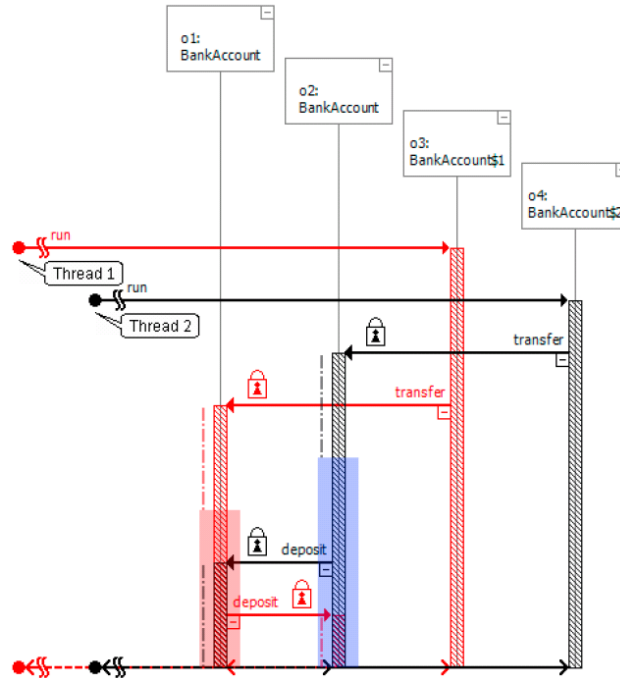


Figure 2.2: JThreadSpy sequence diagram example.

The progression of time is shown on the left side of the diagram and timing icons are shown in the diagram when disproportional amounts of time have passed during an activation of a class. Methods called from libraries or other un-instrumented code is shown in a dotted rectangle. The JThreadSpy visualization can be configured by stack depth, range of thread colors and thread filtering.

2.2.3 Evaluation

The additions to the sequence diagram in JThreadSpy show a high level of concurrency specific detail while preserving user freedom and understandability of synchronization constructs. The instrumenting Agent as used by JThreadSpy offers no insurance that instrumentation is flawless. Differences in coding, compiler and interpreter can cause differences in thread interleaving, or failure to trace at all. In addition, because the instrumenting agent is changing the executing code, it becomes harder to evaluate code in relation to the diagram.

2.3 JACOT

JACOT Stands for Java Platform Object Tool [14]. It is a prototype tool that takes a concurrent Java program as input and visualizes its execution using two views based on the *Unified Modeling Language paradigm*. It uses the Java Standard Kits Java Platform Debugger JPDA [23] interface to dynamically trace the program executions and generate events.

It runs stand-alone on the same platform as the program that is being studied. The basic operation is done by starting *JACOT* and the test program. When running the user has the ability to play, pause, slow down and stop the test program.

2.3.1 Tracing

JACOT's tracer uses the *JDI* [3.2.4] which is based on the *JPDA* architecture, which is part of the *Java Development Kit (JDK)*. On top of the *JDI* event based interface, *JACOT* has built an event gatherer, processor, view formatter and display. The *JDI* interface is connected to the debug-gee VM and generates events as a program runs inside this VM. These events are caught and entered into an event queue. The events are wrapped into a message and sent to the processing layer. There are many types of events generated by *JDI*, the event types *JACOT* listens to are: *method entry*, *method exit*, *VM start*, *VM Death*, *Caught Exception*, *Thrown Exception* and *Class Prepare*. The method of identification and tracking of objects and methods in *JACOT* is by a combination of the object ID and object name. This is important because references within the VM are based on Java native object references that use storage locations. These can change if the object is moved, and become useless when the running program inside the VM stops and exits.

2.3.2 Visualization

The two visualization methods of *JACOT* are a sequence diagram view and a thread state diagram view. The sequence diagram endeavors to be as close to the *UML* specification

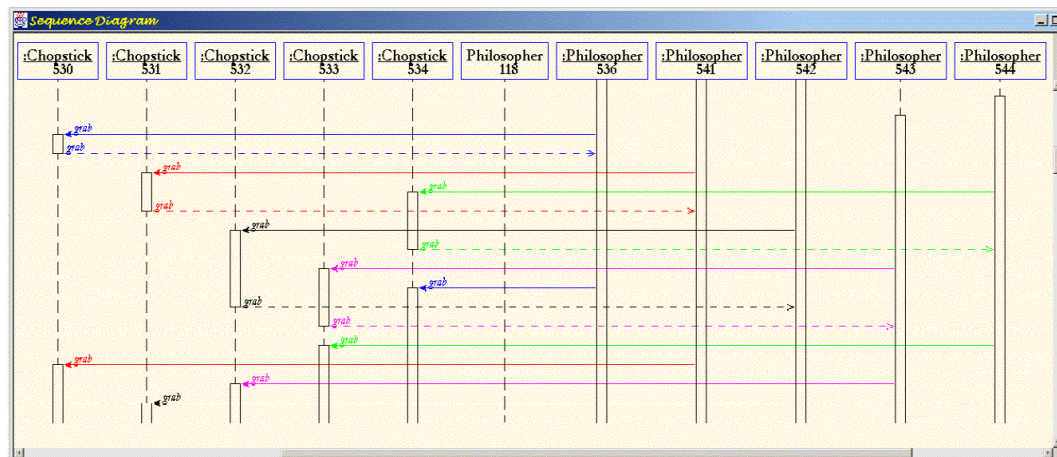


Figure 2.3: JACOT Sequence Diagram.

as possible. The classes are automatically listed on top. From there their lifelines extend showing the creation, activation and termination of the classes. Activation is shown in a vertical bar widening the vertical life bar. The method invocations are categorized into synchronous, asynchronous and self-calls. What is not standard UML is the way the distinction between threads is represented. The different threads are shown in different colors.

2. RELATED WORK

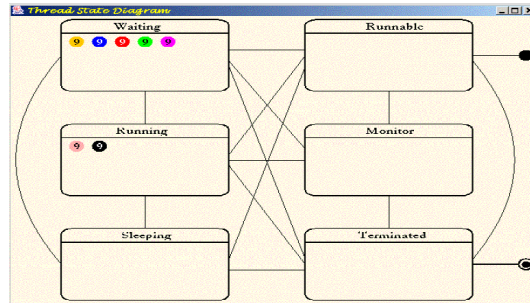


Figure 2.4: JACOT State Diagram.

This makes recognizing thread activity easier and shows when multiple threads are using the same object. Thread activation is shown with a wavy symbol and thread termination is shown by a circled x. The UML state diagram view shows the different thread states. It distinguishes between seven states: *created*, *waiting*, *runnable*, *running*, *waiting* (on a monitor), *sleeping* and *terminated*. The *created* and *terminated* states are not actively visible in the state diagram though. The state diagram shows the other six different thread states with a colored and numbered representation of each thread shown inside the state area of the state it resides in. The other two states: *created* and *terminated*, are represented by icons within the diagram, but are not populated with threads.

2.3.3 Evaluation

The use of colors to identify threads has its limitations: Because of a limited number of colors, only a limited number of threads can be displayed. Using more colors will result in loss of distinction between colors or reuse of colors. Alternative methods like double coloring can be used but this approach may cause confusion as well. The stated uses of the visualizations of this prototype tool are: The visualization of starvation is possible by observing threads in the state diagram that spend unreasonable long times in non-running states or observing an absence of this thread in the state diagram. Dormancy is observable in the sequence diagram by monitoring invocation of synchronization primitives together with thread activity. Deadlocks can be found more easily because of the sequence diagram, by following the problematic threads, their interleaving and the lock sequence of the objects.

2.4 JIVE

Gestwicki et al. present techniques and architecture concepts for the comprehension of concurrent object oriented software are presented and illustrated in *JIVE*³ [10]. *JIVE*, Java Interactive Visualization Environment, is a prototype system for visualizing program states and execution as environments of execution. Their starting point is based on the fact that object-oriented programs are not just data structures, but serve as environments

³*JIVE*: <http://www.cse.buffalo.edu/jive/>

within which procedure activations take place. Object-oriented programs engender the use of smaller methods and more complicated actions among objects. Runtime comprehension therefore needs both structure and interaction views. According to their methodology, a successful interactive visualization (especially for Java) should display seven fundamental properties:

Depict Objects as Environments. Method calls, internal objects, procedures must be visualized both as a microenvironment and a whole to show all the needed relationships.

Provide Multiple Views of Execution States. The current execution state should be observable at various levels of granularity. Abstract and simplified relationships should be shown in case of complex relational structures.

Capture History of Execution and Method Interaction. The history should be observable by using collaboration or time-sequence diagrams that can be set to any point in time for flexibility.

Support Forward and Backward Execution. Backward execution is important for debugging. The granularity of stepping at different levels should be decidable by the user. This should also be applicable for multithreaded programs.

Support Queries on the Runtime State. The change of values is important to understand program execution. Therefore, any changes should always be queryable at any time from any step of the execution.

Produce Clear and Legible Drawings. The visualization should arrange diagrams for understandability of sequence and structure. Lists, arrays, tables should be provided and shown; patterns and structures should be highlighted.

Use Existing Java Virtual Machine. An existing VM must be used, so no custom interpreter is required. This insures compatibility with new Java technology. Any and all libraries should be supported.

2.4.1 Tracing

JIVE has a *JPDA* [3.2.1] based tracer which stores all changes in a *LiveJog* [10] subsystem. The log is coupled with a database in which execution history is stored. Every event in the subject VM is stored in an object making up a delta execution state allowing JIVE to step backwards affecting only the visualization not the subject program, which is suspended during inspection of previous states. Multithreaded applications are supported in this model, but because JIVE is restricted to uni-processing systems, this results in only one operation being processed at one time. These operations can be paused without discrepancies occurring. This stepping system may also allow different executions to be compared to one-and-other.

2.4.2 Visualization

Jive uses many visualizations at once to show execution in detail.

Compact and Detailed views. The detail view shows the objects and the fields and values they contain. Including references and the objects where to is referenced, resembling the user interface of *DDD* [31]. The compact view shows a more concise representation of the object hierarchy, discriminating between instance and static space objects. This provides more of an overview of the object structure. When completely minimized, only the call path is shown, so the interaction of the methods calling objects is visualized in the diagram without any details.

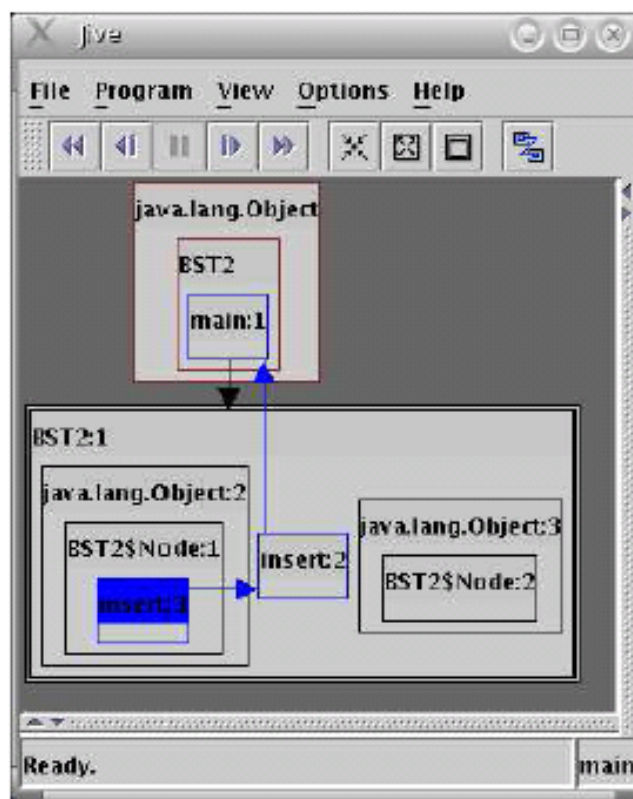


Figure 2.5: JIVE Contour Diagram.

Sequence Diagrams. The sequence diagrams of JIVE adhere to the UML-specification, so the objects (dynamic and optionally static) are shown with their lifelines extending down. The main (initializing) method is shown on the left the other object lines extend to the right. The method lines have the method names and method argument shown above them. There can be interaction through this diagram to the rest of the visual interface. For example, the diagram is linked to the code by highlighting. The activation bar extends over the lifeline

and can be of different colors. With these colors, different threads are shown within the sequence diagram. In relation to different code sections, stack information is shown for different threads.

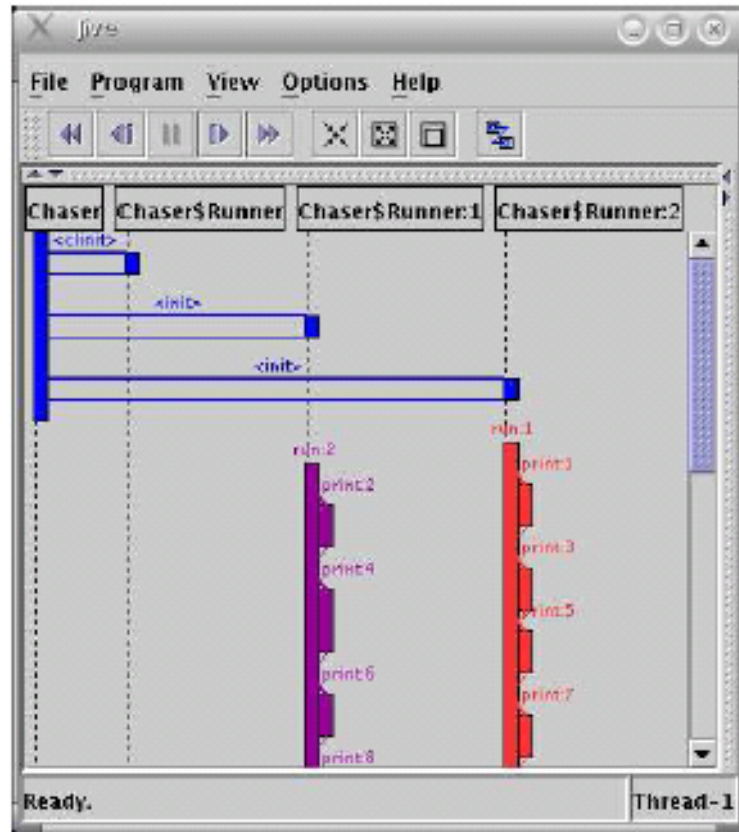


Figure 2.6: JIVE Sequence Diagram.

2.4.3 Evaluation

Gestwicki et. al. have constructed some fundamental properties that are important for future visualization of object-oriented program execution. The JIVE environment delivers a wide view on all aspects of an execution, except for the inclusion of multi colored threads, the use of sequence diagrams can be considered rudimentary and UML compliant.

2.5 JRET

JRET is a *Java Reverse-Engineering Tool*⁴ made by Voets [28]. Its objective is the evaluation of different abstractions for sequence diagrams to ensure simplicity and comprehen-

⁴JRET: <http://jret.sourceforge.net/>

sibility of generated sequence diagrams by dynamic analysis. JRET uses JUnit [4] tests to represent different input scenarios.

2.5.1 Tracing

JRET uses a JVMTI Agent to instrument classes and generate traces. The byte-code interpreter used in the agent is from AspectJ [11]. The tracer instruments code into loaded classes to distinguish between beginnings and endings of method calls, static method calls, and constructor calls. The instrumentation is done on two levels of granularity, class level and object level. *Class level* shows a higher abstraction because one class would represent all instances of that class. *Object level* would represent every instance of every object, exposing more detailed information about interactions, polymorphism and late binding. The user can choose between the different levels. The user can also choose to include or exclude system classes.

2.5.2 Visualization

JRET implements several abstractions to make the sequence diagram more understandable:

- The possibility to choose between object and class level instrumentation.
- Filtering of the system classes in the packages `java.util` and `java.lang`.
- Abbreviate method names, return values, and arguments.
- Remove getters.
- Remove setters.
- Remove constructors.
- Remove asserts.
- Remove internal messages.
- Stack depth upper and lower boundaries definition.
- Message selection and examination, obfuscating other messages.
- Pattern recognition and replacement with representation.

JRET uses two different sequence diagram generators for visualization. The user can choose between generating the sequence diagram to a PNG image file or show it in an editor. Alex Moffat's Sequence⁵ [19] is a tool that uses a specification text file to generate a PNG representation of the sequence diagram depicted in figure 2.7.

⁵Moffat's sequence: http://www.zanthan.com/itymbi/archives/cat_sequence.html

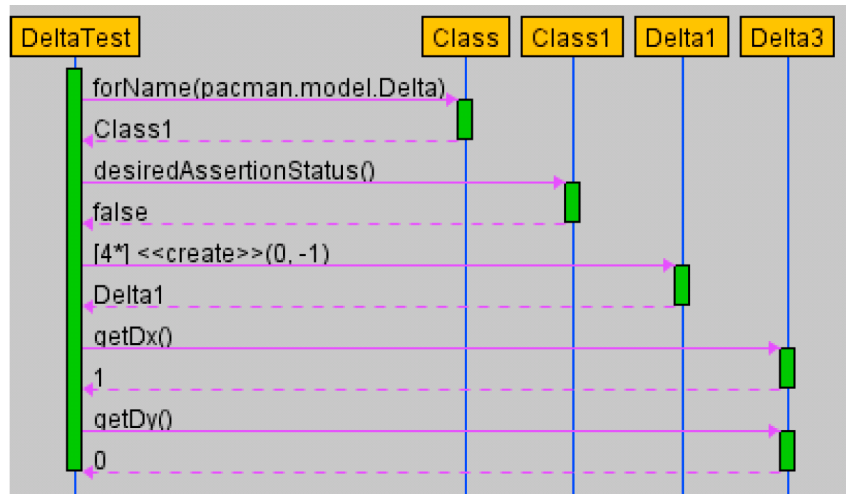


Figure 2.7: Example of the SEQUENCE visualization generator tool.

SDEdit⁶ [26] is also a tool that converts directly from plain text input files. SDEdit can not only generate an image file, but also show its output in the visualizer itself. An example is shown in figure 2.8.

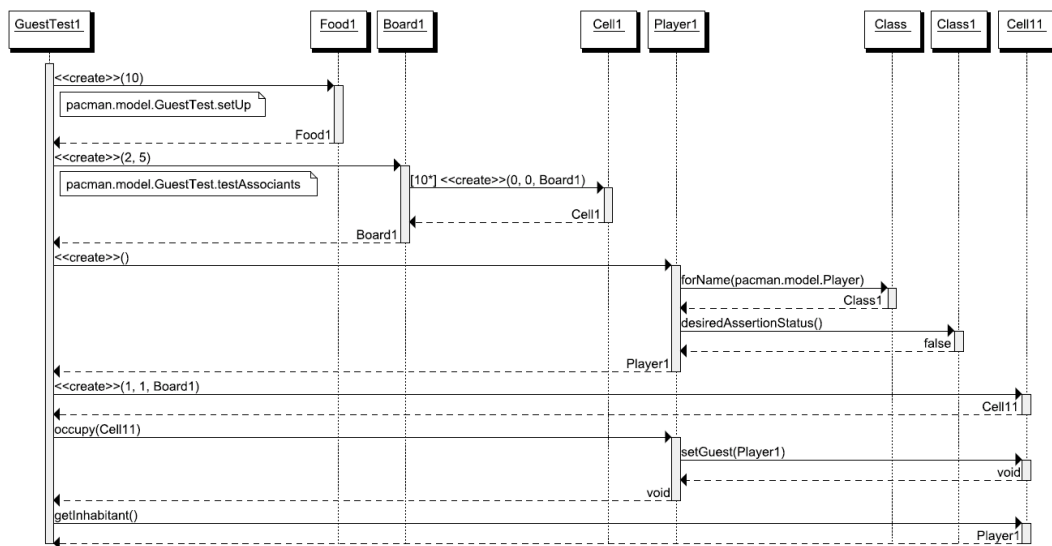


Figure 2.8: Example of the SEDIT visualization generator tool

It has more functionality than the Sequence tool. It shows patterns of multiple messages, labels and has support for threads.

⁶SDEdit: <http://sdedit.sourceforge.net/>

2.5.3 Evaluation

JRET's use of abstractions is an interesting addition to sequence diagrams. Its removal of constructs like messages, getters and setters might give a skewed image in case of concurrent applications, because this can cause the loss of some interactions. Abbreviations of names may suit specific sequence diagrams but can cause confusion when referencing the code because of collisions. The instrumentation level distinction is an important granularity division.

2.6 TPTP

The Eclipse *Test and Performance Tools Platform*⁷ Project is an open platform to allow software developers to build performance tools and integrate these with the platform and other tools [7]. The eclipse TPTP plugin allows an eclipse integrated profiler to add one of the TPTP profiler agents to any Java application's execution and display information and statistics about this applications runtime. The four standard data collectors are; execution time analysis; memory analysis; thread analysis and probe insertion.

2.6.1 Tracing

The TPTP profiling contains two frameworks, the *Martini Framework* and the *Data Collection Framework*. *The Martini Framework* is used to get the data from the VM and send it to the *Data Collection Framework*. *The Martini framework* contains the *profile kernel component*, which holds the *event manager*, *profiling interface*, *external control agent*, etc. The second part holds the profilers: *The Call Graph Profiler*, *Heap Profiler*, *Thread Profiler* and *ProbekitAgent*. The third part is the *External Interface Module JPIAgent*.

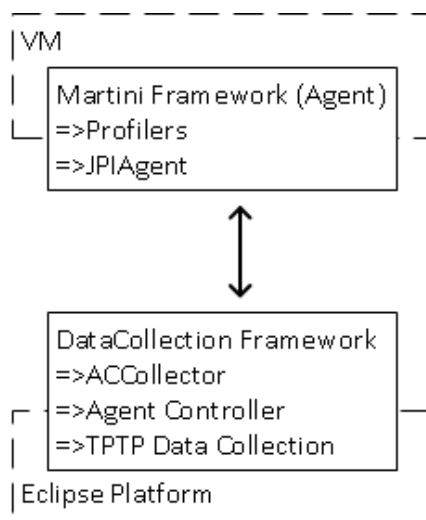


Figure 2.9: TPTP agent architecture.

⁷TPTP plugin: <http://www.eclipse.org/tptp/test/>

The Martini Framework is TPTP's Agent. It creates an interface to *JVMTI/JVMPi* [22] and shares collected data from the VM to the *DataCollection Framework*. It uses different types of profilers to extract different kinds of data. It uses both dynamic and static binary code instrumentation for profilers:

- The *Call Graph profiler* registers all method calls between classes.
- The *Heap Profiler* can traverse the heap to view every variable stored in heap memory.
- The *Thread Profiler* registers all timing data of threads and the events they create.
- The *ProbekitAgent* is used to insert probes, configurable sections of code into methods to target and monitor specific sections of code.

2.6.2 Visualization

Due to the many profiler options and tools, TPTP has many visualizations. Due to the extensible nature of the framework, many more visualizations are possible. We only review those interfaces here that provide new insights into concurrent execution and threading. Examples of the described visualizations are available in appendix B.3.

The *TPTP Execution Time Analysis profiler data collector* can show timing information about an execution in various interesting ways.

Call Tree shows the time spent per method, for every thread with the calling object in a cascaded tree. It also shows the number of calls and various timing statistics. This view shows an overview of a lot of timing information per thread in a simplified way.

UML2 Class Interactions shows a sequence diagram letting us see the interactions between classes conforming to the *UML2* [20] specification. In addition, it adds timing information to the left bar of the diagram, indicating the time spent during and between calls.

UML2 Thread Interactions shows the same visualization as the *UML2 Class Interactions* but instead of interactions between classes, it displays events occurring inside the threads. This diagram also features timing information to the left of the diagram. It does not show much of inter-thread communication but we can see the concurrency between the threads well.

The *TPTP Thread Analysis profiler data collector* collects thread information during execution and can visualize this in the following interesting ways.

Threads Visualizer shows the threads and their life-lines. It draws a line from left to right, which is colored according to its state. The length of the axis contains the events that occur during the lifetime of the threads; they become visible in an alternate view when selected. At the top of the diagram, a time indication ruler is displayed to let us see at what time the state or events have occurred. When a selection is made a timeline is shown which intersects the time ruler so it's clear what moment is referred to, when status or events are displayed. The threads are divided into thread groups which can be hidden if collapsed.

Thread Statistics shows a table of thread data, we see per thread; current states, timing information, times blocked and times deadlocked. The amount of information shown is small but it gives a good indication of the health of each thread. Especially the deadlock information is valuable for when deadlocks are being debugged. Some monitor statistics can also be viewed but this functionality is limited to some timing statistics.

2.6.3 Evaluation

From all referenced works TPTP has the most intricate and high-end tracer. The use of *JVMTI/JVMPi* [22] is built to be dynamic and very speed oriented. Though the framework is designed to be very extensible, it lacks the support to be very developer friendly. The TPTP tools development has stalled and is therefore no longer supported by newer versions of eclipse.

This also means that the profilers, which use JVMTI, are version specific and their usefulness will therefore decline if no more regular development is done. The Call tree allows a good timing overview with adaptable granularity but since its main goal is to show timing information; its value is limited for solving our problems. The *UML2 Class Interaction* gives an attractive diagram showing inherently a reasonable interaction view, but lacking in support for concurrency. The lack of thread visualization is partly covered by the *UML2 Thread Interaction view*, but seeing as very little method communication and variable sharing is shown it still lacks expressivity when it comes to concurrency. *The Threads Visualizer* is a functional overview of the threads timing and lifespan, but provides too little granularity for use in the understanding and debugging of multithreaded problems. The value of the *Thread Statistics view* could be greatly increased if this information could be shown for different moments in or during the execution.

2.7 Others

2.7.1 JaVis

JaVis [18] is a UML-based [20] visualization and debugging environment designed for debugging concurrent Java programs. It uses the JDI for tracing and the *UML CASE tool* for visualization. All visualization is done after termination of execution. Its focus is on deadlock detection and debugging. It is non-invasive to the tested programs because the JDI interface is used. Because of the *UML CASE tool*, no extra visualization features have been added to the standard UML representation. The only interesting addition is a sort of crime trail plot of the sequence diagram, which shows only the sequence of events leading to the deadlock.

2.7.2 JAVAVIS

JAVAVIS [21] is a visualization tool to illustrate the running of small, uncomplicated Java programs. It uses the JDI interface and a specially made Java graphical user interface. JAVAVIS displays two representations of the test program. The first representation is a

sequence diagram which shows a simplified UML diagram, which always starts with the main method and branches other classes to the right. All methods, activations are shown between classes in color. Different threads are shown in different colors. Showing overlapping thread colors is done by drawing them side-by-side, widening the lifeline as it were. JAVAVIS also shows an object diagram window showing the real-time content of the objects in the test-program. The tree-like visualization shows the branching of references from container-objects to the contained instances for all method arguments, local variables and objects.

2.8 Recap

With this related work chapter, we now have some examples of tools and methods to perform dynamic analysis and analyze executions. We specifically take notice of the ways of using the sequence diagram other than those specified in UML. An example is the use of thread coloring as seen in JThreadSpy, JACOT and JIVE.

The limitations of sequence diagrams are just as, if not more important. The use of alternate visualizations in the described tools indicates that the sequence diagram that we focused on is not omnipotent. These alternate visualizations also form an indication of what the users of these tools want to see and what is lacking from the sequence diagram. Examples are: the state diagram in JACOT, the many visualizations of TPTP and the *Compact* and *Detailed* views of JIVE.

In the next chapter, we will further analyze the functionality of the tools and relate them to the problems they are meant to solve. An in-depth evaluation of the tools and the FOT tool created in the course of this research is included in section 5.2.

Chapter 3

Analysis

In this chapter, a breakdown of the different approaches shown in the related work section is presented and an evaluation of features and limitations in relation to our problems will be given. It will also show a more in-depth description of the common tracing and representation options that were only briefly mentioned. These approaches will be considered for a new and improved thread representation tool. In this chapter, we will answer research question RQ4 also by answering RQ2 and RQ3.

3.1 Problems

First, we make a summary of the major problems impeding thread comprehension found in the related work, answering the research question (RQ2). These problems will be used to evaluate whether the tracing and representation options in the next sections handle these problems effectively. These problems will be also be used as a reference, so any subsequent alternative solution can strive to solve these as well.

We subdivide the problems into two genres: Expressivity and Concurrency.

3.1.1 Expressivity

Object oriented code can be difficult to understand, but when concurrency support is necessary, it can be even harder. The use or misuse of synchronization patterns cannot be solved without having a good idea of how the implemented code functions. Dynamic analysis can help display what is happening during a program execution [5]. But there are some problems that need to be addressed in the visualization to ensure understandability of the implementation, execution and the relation between these:

Non-determinism

The non-deterministic nature of threads scheduling causes differences between executions makes it probable that subsequent executions of the same program will produce different execution flows. Complicating the impression of the developer of what is really happening during runtime.

Detail and Complexity

A lot of structured information and details are necessary to understand something as complex as an execution. Therefore, any created visualizations must contain enough detail to capture and relay enough detailed information. It must filter, structure and display the execution's complex situation in to an as non-complicated picture as possible [25, 2]. The more details gained from an execution the more granularity can be provided to the user. This allows the user to gain more insight into the execution without having to return to the implemented code.

Context and Environments

The application of procedural programming approaches to object oriented programs causes problems. Objects in object oriented programming must be seen as separate environments which execute their own and others code. Though a thread is in fact a sequentially executing process, the interaction of other threads can cause problems. To get this right you need to put visualizations in a proper context. So it becomes clear, what is running where with which implementation and limitations.

Information Overload

When using any kind of visualization of a program code, structure or execution the amount of information must be managed because otherwise the important information is not distinguishable from the other information. This can be done by structuring information, hiding information or providing it on demand [25]. A balance must be found to insure there is not too much information nor to little, this could cause a loss of context.

3.1.2 Concurrency

These are some of the main inherent concurrency problems that we define.

Race conditions

Race conditions are non-determinism problems that cause the output or effect of a calculation to be purely dependent on timing or interleaving. This can cause unpredictable behavior if the program was written with a specific execution order in mind.

Deadlocks

When two or more processes are waiting for each other to finish their work, this is called a deadlock situation. This situation can freeze or stop a program.

Starvation and Dormancy

When a process gets no resources or execution time, compared to other processes this is called starvation. When a process is inactive for too long for any reason it is called dormancy.

Synchronization

The use of synchronization constructs and patterns are an integral part of concurrent programming. Synchronization paradigms are usually explained and shown in a very abstract way, which allow for their application in a wide range of situations. These situations are described in patterns so they can be directly applied to common problems. This causes standard pattern based solutions, to not always be applicable to novel situations. Use of synchronization to solve misconstrued problems can cause abuse or overuse of synchronization constructs, resulting in a slower and less predictable execution.

3.2 Tracing options

Here we discuss the different methods for gathering execution information from the Java Virtual Machine [30] environment and the executions that run within.

3.2.1 Java Platform Debugger Architecture

The Java Platform Debugger Architecture¹ (JPDA) is a debugging architecture that allows tool developers to easily create cross platform debugger applications [23]. JPDA consists of three layers:

- **JVM-TI** Java VM Tool Interface
- **JDWP** Java Debug Wire Protocol
- **JDI** Java Debug Interface

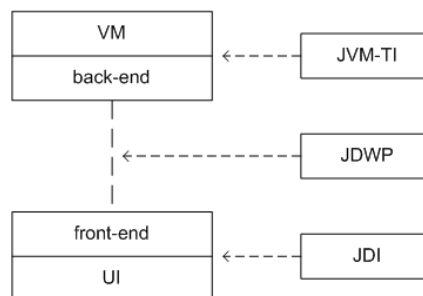


Figure 3.1: Java Platform Debugger Architecture.

3.2.2 JVM-TI

JVM-TI² is a native API which allows native libraries to capture and control a VM [22, 24]. Binary programs implementing these libraries are often called Agents and they are often

¹JPDA: <http://docs.oracle.com/javase/6/docs/technotes/guides/jpda/architecture.html>

²JVMTI: http://java.sun.com/developer/technicalArticles/J2SE/jvm_ti

3. ANALYSIS

used for profiling and tracing of programs. In earlier Java Development Kit (JDK) versions, the Agents would use the Java Native Interface (JNI) with the JVM Debug Interface (JVMDI) for debugging and or the JVM Profiling Interface for profiling. While these techniques are still mentioned in related work, both JVMDI and JVMPI are removed from current and future JDKs.

Instrumenting Using the JVMTI an agent can instrument the binary files, to change them before they enter the VM. The code that is introduced can take control of the VM during runtime and for example write trace information about the ongoing execution [30]. The advantages of using such an agent are usually speed related. Since all that is run in the VM is native code and the only overhead introduced is the outputting of logging information, the slow-down is relatively minor in relation to other tracing or profiling methods. This way of tracing requires some code to read and write binary class files, because if the files cannot be read they cannot be altered. A usually third-party created binary evaluator library is used to convert the classes to a legible format and converts it back after the changes are implemented. Agents are usually are not written in Java because they need to be compiled to binary. This way of changing code is not foolproof and the way the subject program reacts to the injected code can vary, causing debugging to become more problematic.

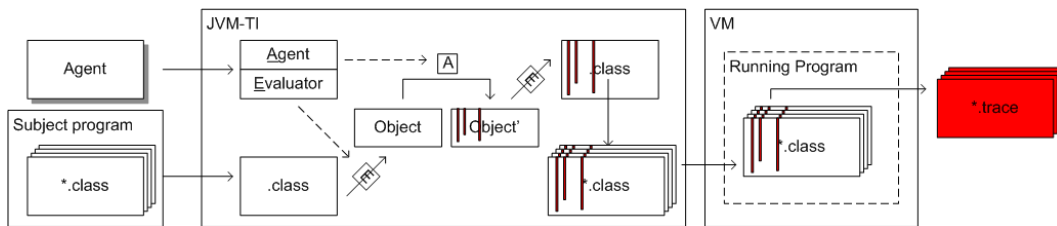


Figure 3.2: JVMTI operation using static instrumentation.

With *static instrumentation*, code is added prior to execution. Usually a jar file is selected that needs to be instrumented and the code is injected before the test program's execution. After the new jar file is created the new code is run when this new jar is run. The only way to undo this kind of instrumentation is to recompile or reuse the original jar file. With *dynamic instrumentation*, classes are instrumented in-memory without modifying data on disk. This allows for quick changes to the code without having to recompile or re-instrument files. Its influence on the normal operation of the program is less invasive. To cleanup, any changes made to the original program the instrumented code can simply be unloaded.

Library Events Other ways of getting trace information using agents without injecting code is by the use of JVM-TI and JNI libraries to request and capture events from the VM. These events can vary from method entry and exit to thread CPU timer information. With these events, other information can be accessed like field values at that time. This information can also be outputted as trace logs. The advantage is that the subject code is

not altered in any way. The disadvantage is that it causes a slowdown for each type of event you subscribe too.

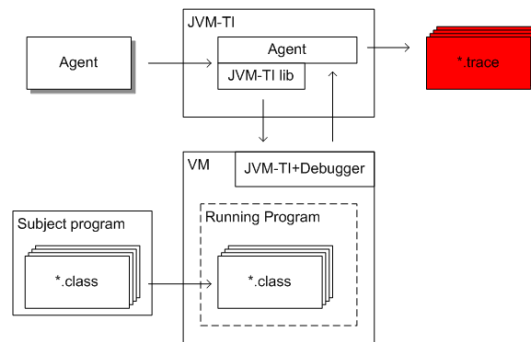


Figure 3.3: JVMTI operation using library events.

3.2.3 Java Debug Wire Protocol

JDWP is an XML based communication protocol that allows communication between the debugger and debug gee. Though communication can be established from any xml-based sender, this protocol is geared towards use in combination with the Java Debug Interface. There is little specification for use without JDI.

3.2.4 Java Debug Interface

The JDI is a high-level Java language interface for using the JDWP to communicate with the JVM-TI enabled VM. It is built so that tool developers can easily write debugger applications. The JDI API features many but not all of the JVM-TI functions and supplies direct reference objects to the VM. This means that variables and their changes can be immediately observed in any Java tracing or debugging application but this also means that references are transient in nature like in lower-level programming languages like C and C++. All references to an inactive VM therefore revert to null. For any post-mortem traces or analysis, an alternative storage is necessary. JDI slows down execution considerably due to communication overhead and VM operations. The communication from the JDI-enabled application and the VM is done over TCP socket communication or through shared memory.

3.3 Representation options

In this section, we will answer the research question of; what kinds of operations are the current representations of threads useful for (RQ3). We will focus on sequence diagrams, we will then discuss alternative diagrams that add structure lacking from sequence diagrams. We will then make a list of ways to recreate these diagrams and evaluate them for possible use in this thesis' alternative solution.

3.3.1 Sequence diagrams

We start with solutions the basic UML sequence diagram brings and enumerate the most prominent additions and advancements in sequence diagrams for the visualization of concurrency. We analyze why they are employed and what problems they are meant to solve.

UML Sequence Diagram

When an execution trace is drawn in a sequence diagram, interaction between objects is shown. This allows the developer to get an impression of the non-determinism of his program, by comparing the differences in message interleaving between execution traces. This helps the developer to cut through some of the complexity of object oriented concurrent code and ease the debugging process. Because we use dynamic analysis, only the current execution path and its interleaving are shown. This counteracts information overflow that would occur if all possible interactions extrapolated from static analysis would be shown.

Colored threads

The coloring of thread paths have been applied in JThreadSpy, JACOT and JIVE. This coloring shows the execution of the thread laid over the activations of the lifelines and the messages between the objects. Showing threads this way, can better show non-determinism and the resulting concurrency problems like; race conditions, starvation and dormancy. For example, when a thread is not seen enough it may indicate starvation if a thread is not seen at all it may be dormant.

Timing info

The addition of timing information to the sequence diagram as has been shown in JThreadSpy, UML2.4 and TPTP makes noticing starvation and dormancy more noticeable. When a process takes a lot of time to finish a task, thereby depriving other processes the resources to complete their tasks, this can be more-easily seen.

Variable granularity

Many approaches to vary granularity have been seen in JThreadSpy, JRET and TPTP. The main reason for these approaches is to avoid information overload, unnecessary complexity and to increase the amount of useful information at the same time. JThreadSpy, TPTP and JIVE all use zooming functionality to allow for the display of larger more intricate sequence diagrams. JRET focuses on the use of abstractions to refine and compress the amount of information based on patterns. JACOT, UML and JIVE also recognize the need for difference in granularity but try to solve this by showing multiple representations of the same executions in different diagrams.

Alternative visualization

Alternative visualizations as seen in JACOT, Jive, UML and TPTP, show some need for a different perspective or macro view. Using sequence diagrams in conjunction with an overview, allows for the use of more concise sequence diagrams preventing information overload by delegating the comprehension of the main structure to an overview diagram. This helps to make diagrams simpler because a lot of data that is not necessary for the current analysis can be hidden. Only what is in focus will be shown, what is out of focus is defined elsewhere.

Added indicators and icons

The use of extra indicators and icons is useful in sequence diagrams. The use in JThreadSpy is very expressive in its display of concurrent behavior. Showing the contestation of locks is intuitive and novel, though it requires some extra legend to understand the icons. Its expressivity towards synchronization allows for debugging more concurrency problems, like deadlocks and starvation.

3.3.2 Diagram Libraries

There are multiple options for displaying sequence diagrams in a dynamic analysis tool.

Image generator

As shown in JRET there are sequence diagram generators that can produce a single sequence diagram image that can be displayed. Examples of these are SEQUENCE and SEDIT. The advantages of these tools are the ease of integration with the tool and the simplification of the graphical user interface. The disadvantages are; the fact that the granularity of the diagram becomes more fixed, the lack of user interaction options causes the amount of elaborative information that can be requested to be lower.

Internal GUI

Producing a custom graphical user interface for the tool was done in JACOT, JThreadSpy and JAVAVIS. This GUI approach gives the most control over what is displayed. The development of the sequence diagram drawing takes more work but can be as expressive as desired. Any interaction with the diagram can be programmed but JACOT and JThreadSpy did not implement further interaction. Timing information, custom icons, etc. can be added at will. This way of generating sequence diagrams makes it harder for reuse.

Display library

The use of a visual library to generate the sequence diagram can be seen in TPTP. It has a separate package generating and displaying sequence diagrams in eclipse. It allows other developers to use the diagram generators in their tools. The expressivity of this library

3. ANALYSIS

allows for little addition in the way of new sequence diagram operation icons or thread coloring.

UML suite

Existing UML development and display suites can be adapted to generate sequence and other UML based diagrams as mentioned with JaVis. It works the same way as the image generator. A specification is generated by the tool that can be viewed in an UML suite. No more information can therefore be requested from the diagram through interaction than the information contained within the specification. The use of the tool would require the UML suite to be present for viewing the output.

Chapter 4

Field of Threads (FOT)

In this chapter we describe the Field of Threads (FOT) tool. It is based on the lessons learned from the previous chapter *Analysis*. We start with the design choices and objectives for this tool and then we discuss the tracer and finally the visualizations. The FOT tool is a plugin for the eclipse development platform written in native Java. It can trace any Java application local or remote through the VM debug interface. FOT can generate an interactive field oriented sequence diagram and a tree-table based, field overview interaction analysis.

When debugging an object oriented application in Java, the debugger usually suffices to resolve the problem. However, when facing a concurrency-based problem, the problem becomes harder to grasp using regular stepping debugging. The non-determinism makes it so that one pass-through is not enough anymore to grasp the events leading up to a faulty situation. Most visualizations focus on objects that interact with each other. Others show threads and what fields they contain. Although these visualizations present the necessary information in a structured and comprehensible way, it remains up to the developer to remember causes and effects, recompose the situation and find a solution. This solution is often the application of a pattern. Mutual exclusion, semaphores, synchronization and other concurrency patterns reason from one or more variables being read and modified by threads. They are not based on objects with threads running through them, represented by thread-added sequence diagrams, and they are not based on threads, containing objects, containing fields, represented by a thread inspector.

What if we can represent the situation the other way around? Lift out the variable or field from which the value is not what it's supposed to be and map the interactions that shaped the value to its current value in sequence. This would create a diagram closer to the in-mind representation of the problem. The transformation of the visualization would show, not the thread containing fields, but the field with the threads that shape them. We have now centralized the field that should be the starting point. From this idea comes the name of this tool: Field of Threads.

The old relation between fields and threads can be seen for example in the *TPTP Thread Visualizer* [7] or *Call Tree* [B.3.1]. A generic tree representation of this relation is shown in table 4.1. We change the relation to field centered one shown in table 4.2.

4. FIELD OF THREADS (FOT)

-Thread		
+>	-Object 1	
	+>	Field 1
+Thread 2		
+Thread 3		

Table 4.1: TPTP Call Tree Relation.

-Field 1	
+>	-Thread 1
	-Thread 2
+Field 2	
+Field 3	

Table 4.2: FOT Object Tree Relation.

4.1 Overview

The objectives and choices that were made for the development of the FOT-tool are described here:

OBJ1: Dynamically trace any concurrent Java programs.

OBJ2: Visualize the execution trace in a way understandable for any developer.

OBJ3: Add a new visualization that would show the field centered idea.

FOT is a Java only tool: It is written in Java and can trace any Java program running inside a VM with a debug interface. FOT uses JDI as a tracer which conforms to the following requirements:

- There is no change to the code of the traced application.
- JDI supports remote debugging.
- Any JDI-based tracer is supported from JDK 1.5 onwards.

Some limitations of the use of JDI are:

- JDI cannot read information about non test-classes, like method names from system libraries, due to Java security restrictions.
- The traced application will experience a significant slowdown during execution.

FOT is built as an eclipse plugin [6]. Eclipse was used to facilitate the visualization of the generated diagrams and to make it easier to use FOT alongside a regular debugger.

FOT generates its visualizations post-mortem, meaning the visualizations are generated after the test application has run and stopped. FOT will not use back stepping functionality to step through the execution in reverse order. This functionality would limit the use on concurrent programs because stepping back would be problematic to non-deterministic multithreaded applications [15]. Figure 4.1 provides a flow diagram of the workings of FOT.

4. FIELD OF THREADS (FOT)

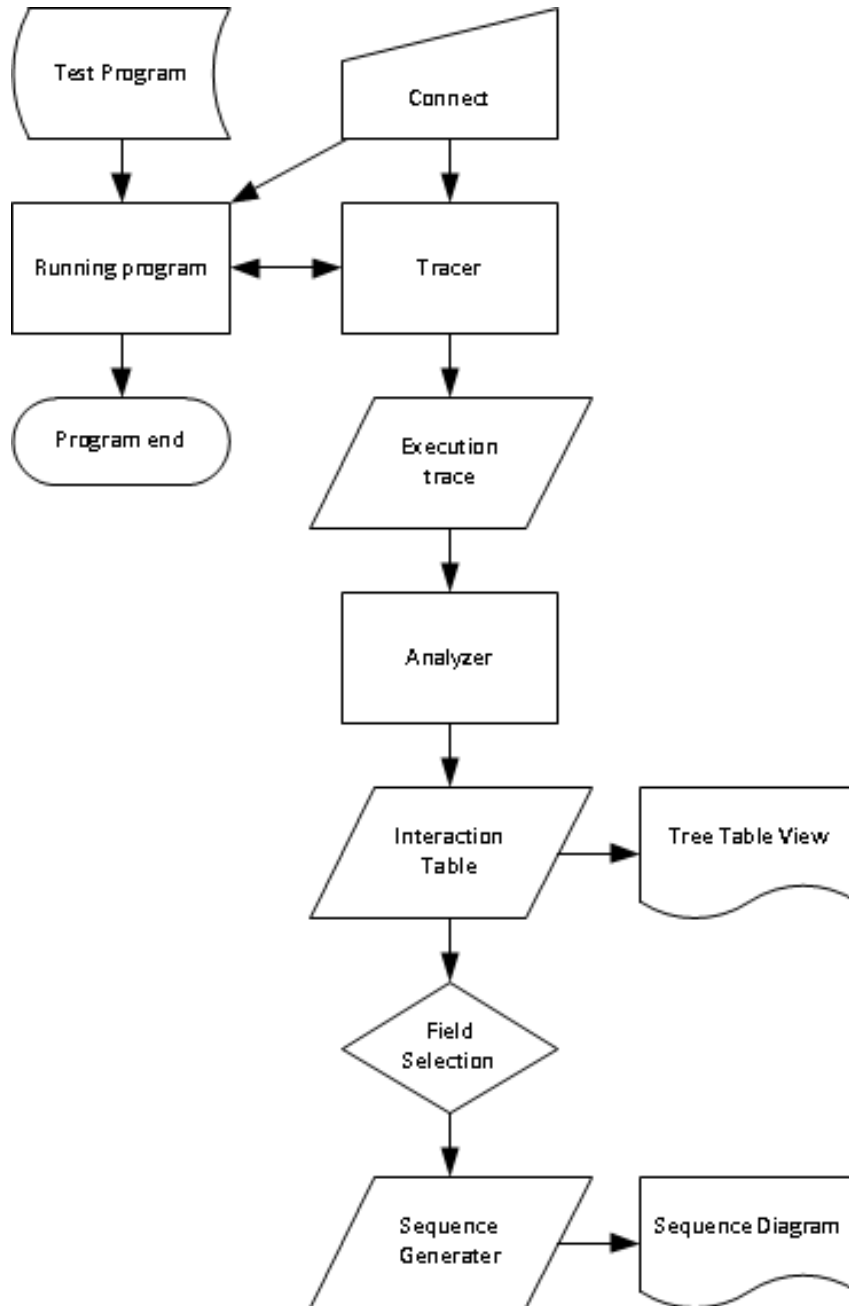


Figure 4.1: FOT Flow Diagram.

The tool can connect to any running VM with the debug interface enabled listening to a TCP/IP port. When a program is operating or waiting, depending on the execution arguments specified to the VM, FOT can connect via the debug port. It does not matter whether the test program or any other Java application is running inside the target VM. Once connected the tracer receives trace data until the VM stops. This tracing data is stored

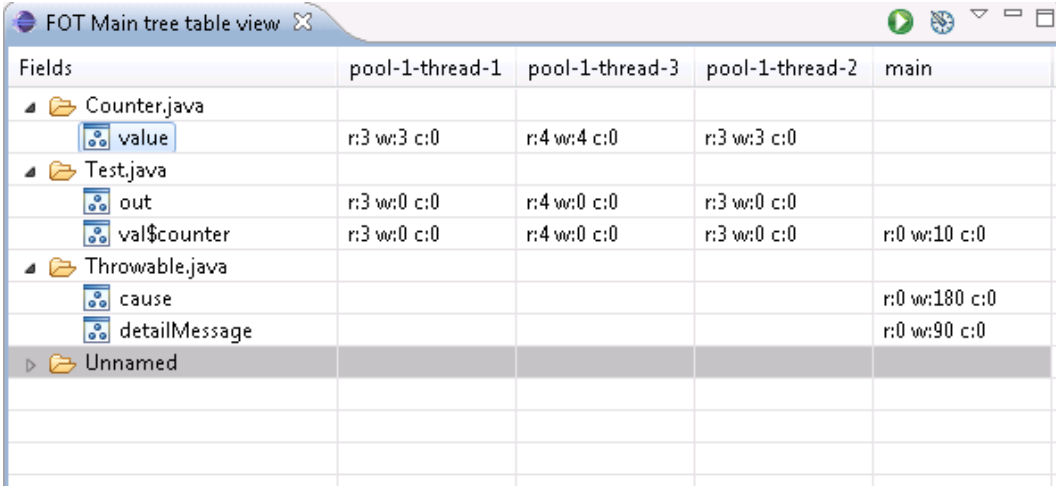
in memory into an execution trace. This trace is then analyzed by an analyzer filtering and structuring the data into an interaction table. This table is then used to show the *Tree Table View*. When a field is selected in the *Tree Table View*, a sequence diagram is generated and shown in the *Sequence Diagram View*. The mentioned Views are tabs in the eclipse platform that can be activated and disabled at will.

4.2 Visualization

FOT features two visualizations as mentioned before. The *Table Tree* view and the *Sequence diagram* view. The Field centered concept as described earlier is applied to the two views. The *Table Tree* view is meant to give an overview of fields of interest and the *sequence diagram* view is meant to show the interactions for analyses. A global overview of the visualizations and controls is also included in appendix B.4.1.

4.2.1 Table Tree view

The *Table Tree* shows a table with on the columns the threads that run in the traced VM.



Fields	pool-1-thread-1	pool-1-thread-3	pool-1-thread-2	main
Counter.java				
value	r:3 w:3 c:0	r:4 w:4 c:0	r:3 w:3 c:0	
Test.java				
out	r:3 w:0 c:0	r:4 w:0 c:0	r:3 w:0 c:0	
val\$counter	r:3 w:0 c:0	r:4 w:0 c:0	r:3 w:0 c:0	r:0 w:10 c:0
Throwable.java				
cause				r:0 w:180 c:0
detailMessage				r:0 w:90 c:0
Unnamed				

Figure 4.2: FOT Table Tree view.

On the rows are the fields nested in a *tree parent* folder with the name of the class in which the field was defined. The class folders can be collapsed like the folder *Unnamed* is in the example figure 4.3. The example shows the execution trace of the example program *Counter* also shown in appendix B. Its function is to use multiple threads, in this case three, to count from one to ten. The highlighted field in figure 4.2 is the incremented integer field named *value*. The threads, shown as columns, are the *main* thread and three threads in *thread pool 1*. The table cells in the thread columns show a string of values in the format of “r:# w:# c:#” indicating the number of reads, writes and collisions, the thread has done on the field. The collisions indicate the situation that during a read or write another thread is

4. FIELD OF THREADS (FOT)

waiting for other threads to finish.

These collisions can be detected because FOT checks if a thread is in an objects wait list. Every time a value is read or changed, an object (containing the field that has a value) is temporarily owned by a thread. Only the thread temporarily owning the object can change or read the value of a field.

If for example a *thread T_A* tries to access a *field F* while a *thread T_B* is working with *field F*, *thread T_A* ends up in *field F_I*'s wait list. Here *thread T_A* and perhaps other threads await their turn to use and temporarily own *field F*.

This construction is built into Java to prevent any field from being read by one thread while being written to by another thread, or to prevent two threads from writing to the same field at the same time.

Buy this only works when a field is protected by a monitor through use of a mutex or synchronization paradigm or construction. This construction works on an object level, meaning that a thread is waiting in a wait set of an object, not in a wait set of a field. This causes FOT to show a waiting thread (or collision as it is called in FOT) when a thread is waiting for the object containing the field. This distinction of fields and objects can cause false positives, but it was the only way to observe this behavior provided by the VM.

The reason why waiting threads are called collisions or "c" in FOT is because the "w" from waiting was already taken by writing. There are different ways for a thread to be waiting, for example, timed waits, but this behavior is not seen as a collision because that thread will not be registered in the wait set of the object in question.

When observing the highlighted field *value* (in the top left of figure 4.2) it becomes apparent that all three threads interact with the *value*, but no collisions occur. When a collision does occur, the whole field row of the table is highlighted in gray.

The four parent folders show the three super classes *Counter.java*, *Test.java* and the native class *Throwable.java* that only operates on the main class. The fourth *Unnamed* folder contains all the library and native binary classes from which the tracer could not obtain the un-compiled code reference, location or filename. The list of binary fields that are necessary for background processes and VM operation are often large in number and placed in the *Unnamed* folder. An example of these fields is shown in appendix B.4.4. The *Unnamed* folder is shown highlighted in gray, indicating it contains a field with collisions in its execution.

When right clicking on a field a context menu is shown containing three options. This is shown in figure 4.3.

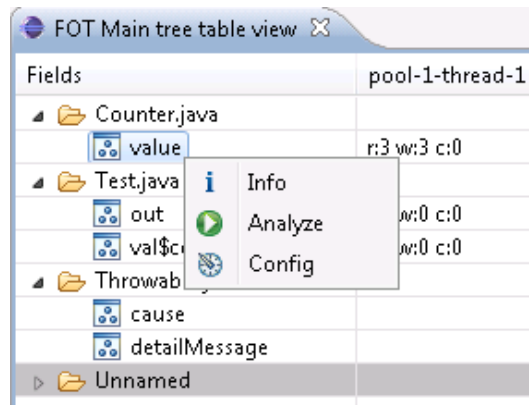


Figure 4.3: Table Tree View Context Menu.

The top option opens the info option, figure 4.4, showing more background info on the field's location, type, name and numerical hash. The other options *Analyze* and *Config*, start and configure a trace operation and correspond to the same icons in the top right of the table tree view of figure 4.2. The function will be described in more detail in the operations section.

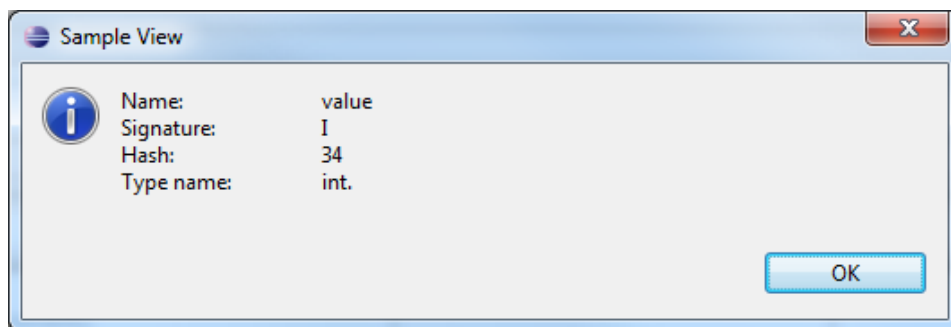


Figure 4.4: Table Tree View Value Info.

If a class super-element with a folder icon is selected, only its name is shown. Any field can be double-clicked to show a sequence diagram representation of the thread interaction with the selected field in an open sequence diagram view.

4.2.2 Sequence Diagram View

The *Sequence Diagram view* shows a generated sequence diagram of the field that was double-clicked last in the *Table Tree View*. Figure 4.5 shows the generated sequence diagram related to the *value* field shown highlighted in the *table tree* view of Figure 4.2.

4. FIELD OF THREADS (FOT)

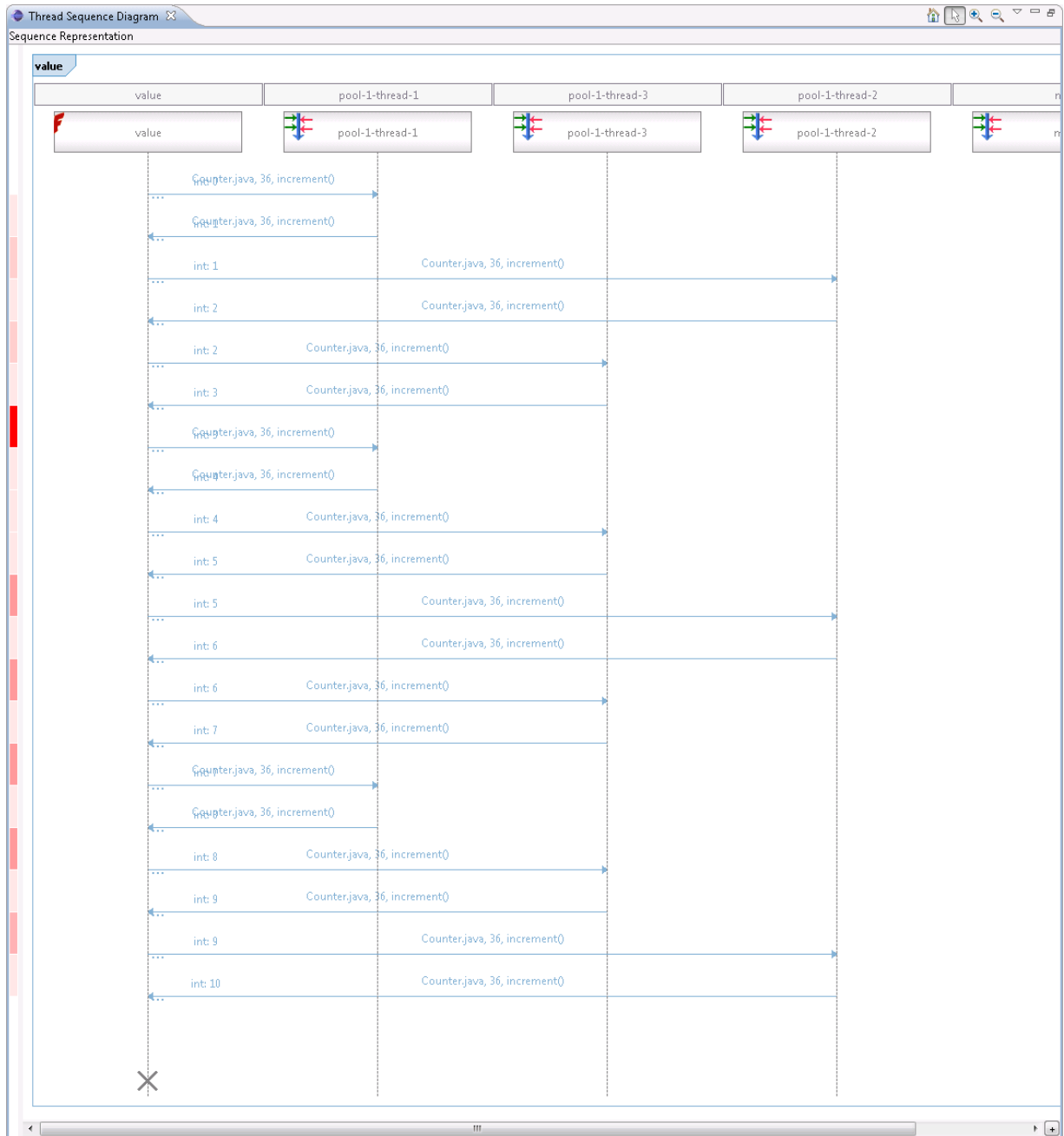


Figure 4.5: FOT Sequence Diagram view.

This sequence diagram is also based on the field-centered approach of FOT. At the top in blue, the selected field name is shown. Below that is the field in question, in this case *value*. The field is always shown on the left side of the sequence diagram and what would be the lifeline's object name shows the field name and a red "F" (F) on the top left side. The other

lifelines are the threads that were also shown in the *Table Tree View* indicated by an icon with five arrows (↕). The messages in this sequence diagram only run between the field and one of the threads. A message from the field to a thread indicates a read operation and a message from a thread to a field indicates a write operation. Each message has a written description above it. This description describes in-order; the class in which the method causing the message is defined, the line-number on which it is defined and the name of the method followed by "()". Arguments and return values of the method are not shown. When a message is selected, a more verbose version of this description is shown as in figure 4.6.

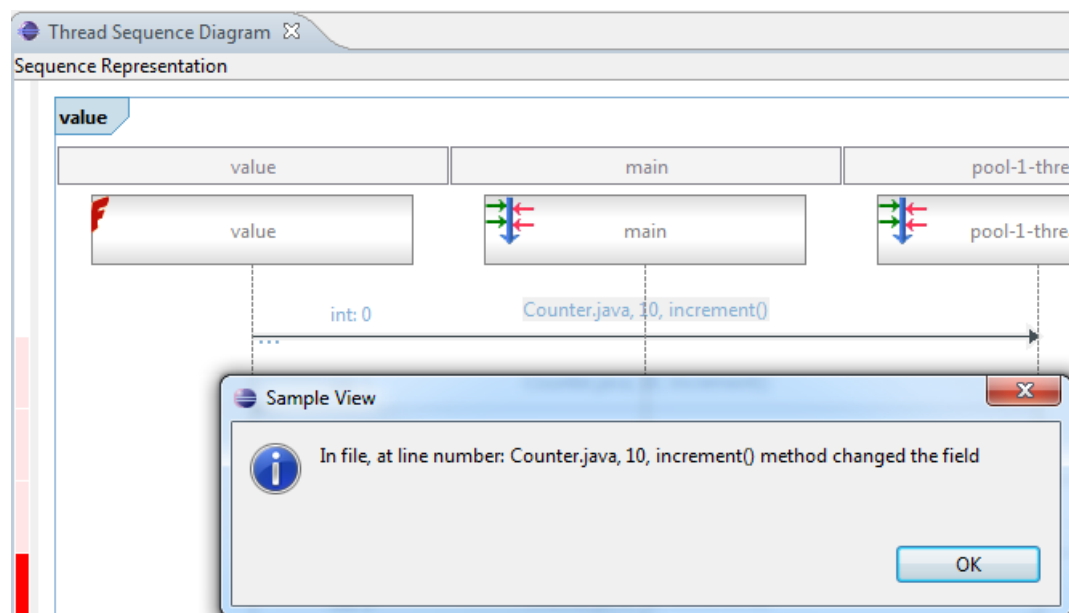


Figure 4.6: Sequence Diagram View Message Description.

When a mouse-over operation is performed on a message, a tooltip balloon is shown with the method description and the timestamp of when that message has occurred. To the right of the field lifeline, for each message the type and value of the field is shown. An example of this type and value representation is: “int: 3” as visible on the left side, near the field lifeline in figure 4.7. However, this value is displayed only if that value comes from a recognized simple field object.

Timing information is shown in a vertical bar to the left of the diagram, also in figure 4.7. This bar shows the delay between two messages. If the relative delay is large, it shows a redder color on the bar than when the delay is small. When this bar is selected, the timing color is shown overlapped with the lifeline and the number of nanosecond delay and relative range is displayed.

4. FIELD OF THREADS (FOT)

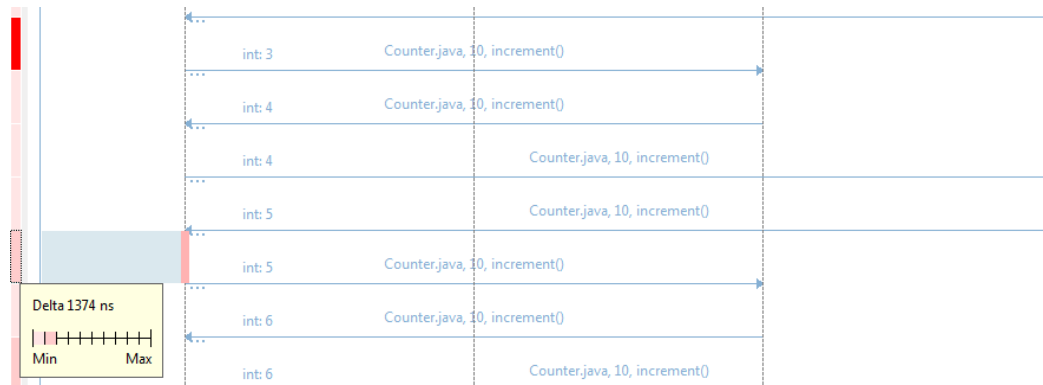


Figure 4.7: Sequence Diagram Relative timings.

The diagram can be scrolled and zoomed by use of the scrollbars and the loop icons at the top-right of the Thread Sequence Diagram view. There is also an overview zoom function visible when + icon at the bottom-right corner of the view is selected. At the top right there are also some other options to reset the zoom, show the node stop or end and to redefine the timing scale for the timing bar. The sequence diagram generated is the same as that from TPTP [7]. This plugin view has been reverse-engineered by the *Linux Tool Project*¹ to a plugin called *TMF* [1]. This plugin allows generation of the sequence diagrams seen in FOT though a loader interface. An example of the generation code is shown in listing B.4.2. The TMF plugin also adds preferences to the standard eclipse window preferences shown in Figure 4.8 customizing of colors and sizes of the sequence diagram.

¹LTP: <http://eclipse.org/linuxtools/>

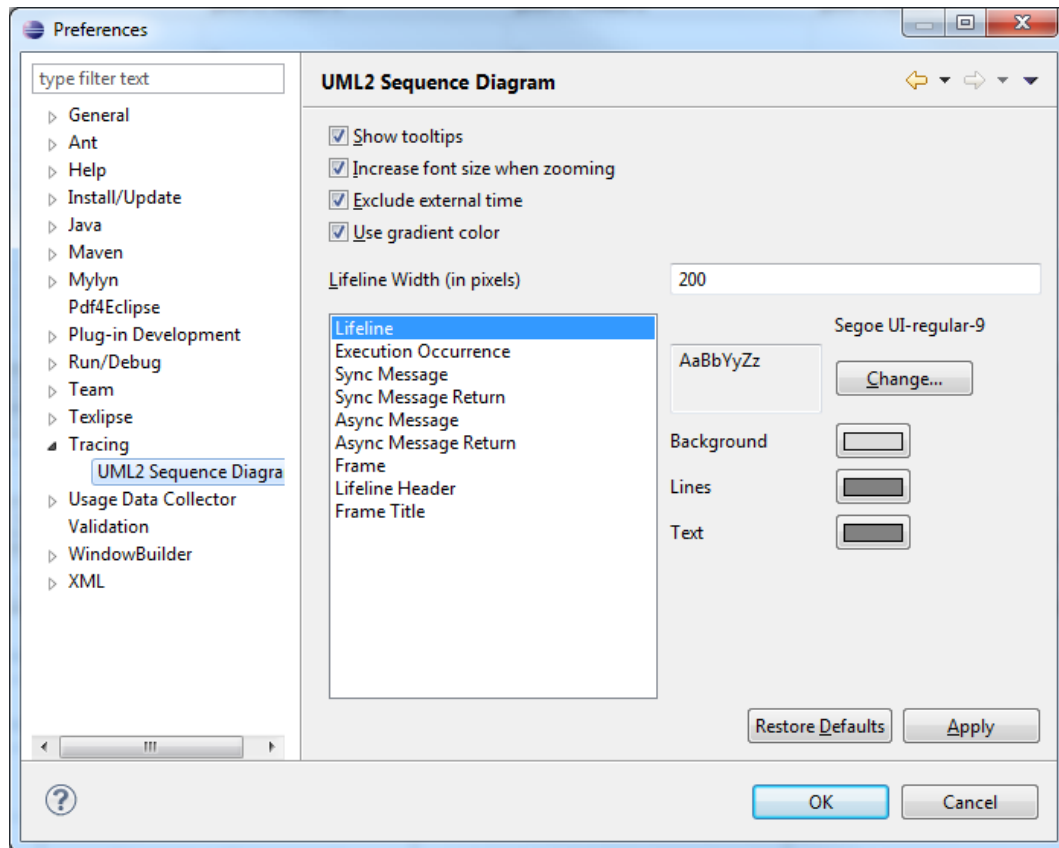


Figure 4.8: Sequence Diagram Preferences Dialog.

4.2.3 Operation

To operate the FOT plugin, it must be installed on the *eclipse platform* [7]. Then a test program must be run with the debug interface enabled. This can be done by adding a line to the VM-arguments of the Java applications run configuration: `-Xdebug -Xrunjdpw : transport = t_socket, address = 8000, server = y, suspend = y`. Here the suspend flag is set to "yes" which means that the application starts running only when FOT is connected to the VM. The debug-gee application can reside in the same environment as FOT or any other environment, as long as a socket connection can be made, on port 8000 in this case. Port 8000 is the default port for FOT. This can be modified in the *configuration dialog* accessible by the round gray icon on the top-left side of the *Table Tree* view user bar, visible in figure 4.9. It is also visible in the context-menu of the field's column. The configuration dialog also allows changing the connection address and setting filters to prevent some native classes from appearing in the trace. Standard it filters out the `sun.*` and `java.*` native libraries. In the counter example, this effectively hides the Java garbage collection thread.

4. FIELD OF THREADS (FOT)

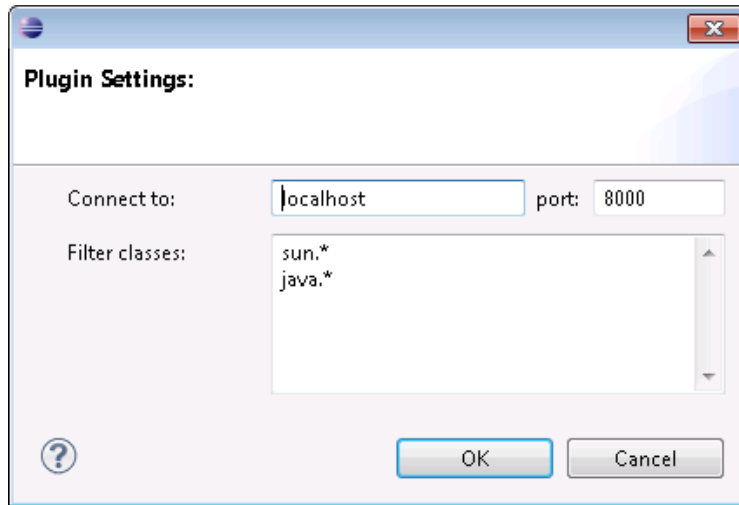


Figure 4.9: FOT Configuration Dialog.

The last green round icon with the play-triangle (▶) in it starts the FOT tool visible in the *Table View* from figure 4.2. This connects FOT on the specified port through the *JDWP* [23] described in: 3.2.3 to the target VM. The plugin will display a progress dialog while the test application is running. After the test application is stopped, the *Table Tree* is populated.

4.3 Structure

FOT uses the *Java Debug Interface* to generate trace events. When FOT's tracer connects to a VM it requests a list on any fields specified in the program. Then it requests all those fields to generate an event when they are either accessed or changed. Such a request is called a watch-point. The VM is also ordered to request the same of any field registered thereafter. The only events stored by the VM tracer are the *com.sun.jdi.event.AccessWatchpointEvent* and the *com.sun.jdi.event.ModificationWatchpointEvent*. These events contain references to a lot of information within the VM at that time. The events are handled immediately so the variables do not change. It is very important to handle and store events and the relevant information from these events quickly because the runtime inside the VM-environment is constantly changing. Any information referenced in a VM can be dereferenced or changed quickly. All events and the extracted data are stored in an execution trace data structure.

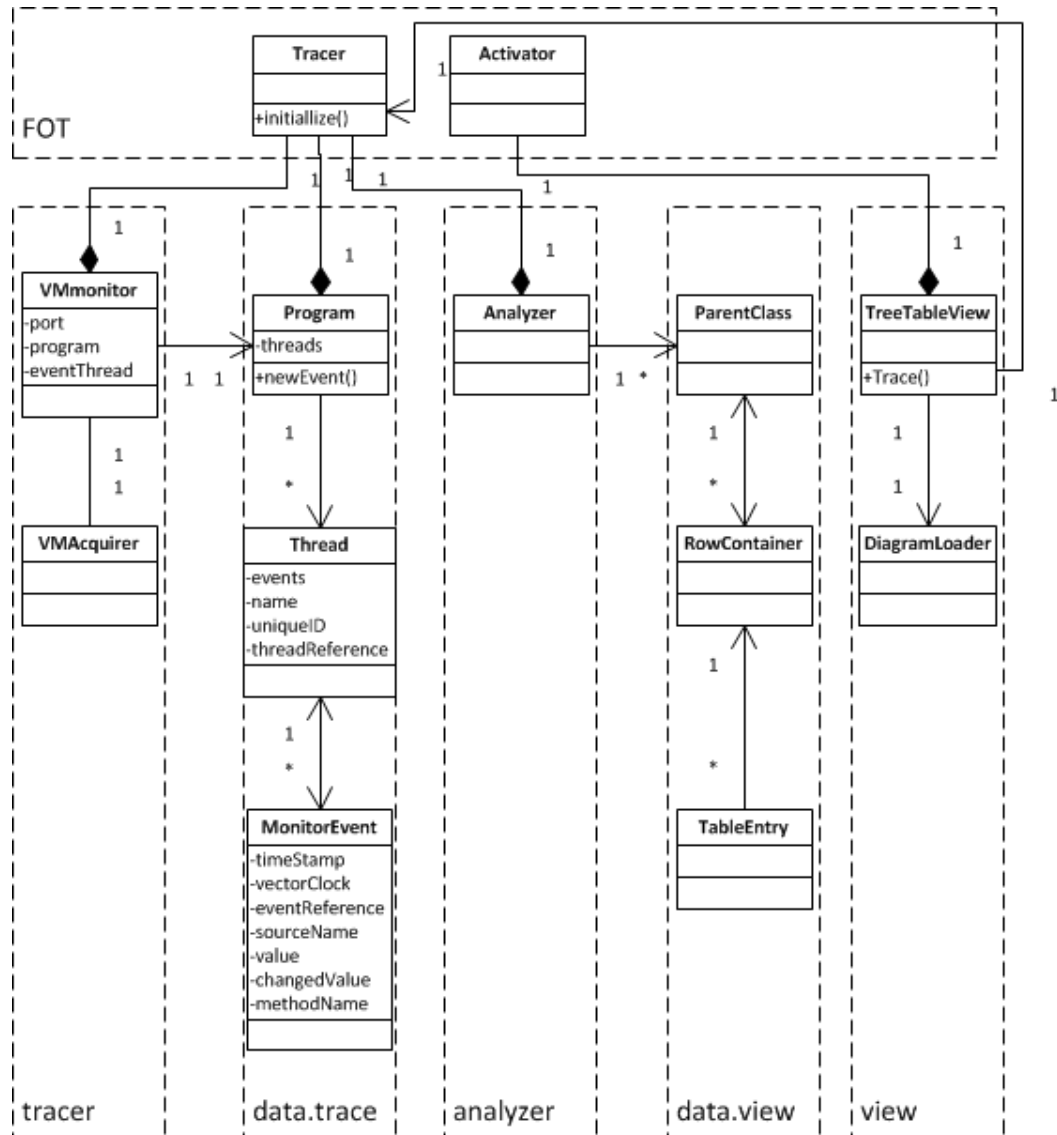


Figure 4.10: FOT Class Diagram.

The class diagram of FOT in figure 4.10, shows the classes in the order of the flow diagram from left to right. On the top are the main FOT activation classes. The *Activator* class starts the plugin and the *Tracer* class handles the main flow of a trace. Starting with the *VMonitor* and the *VMAcquirer*, the *VMAcquirer* connects the tracer to the VM JDI interface on a specified port and hands this connection over to the *VMonitor*. The *VMonitor* runs a separate thread that registers the services, receives and passes on the events. The events are handled and stored in the *Program* class together with all other trace information for this single program trace.

4. FIELD OF THREADS (FOT)

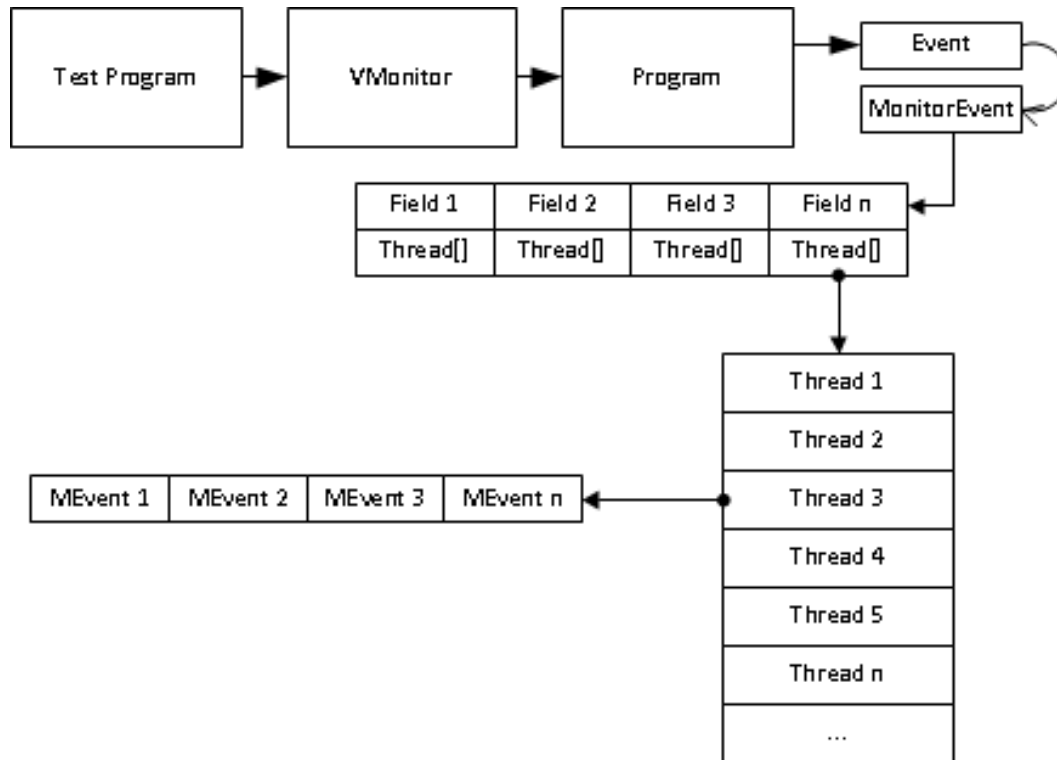


Figure 4.11: FOT Trace Data Structure.

The data structure created is shown in figure 4.11. It shows how Program creates a *MonitorEvent* from a traced event. This *MonitorEvent* is stored in the *threads* object. This object contains a hash map of the fields within the program. These fields are all associated with lists of threads. These threads represent the threads that interact with the fields during the runtime of the test program. Within the threads in the *threadLists* a list of the interacting *MonitorEvents* is kept. Therefore, FOT now knows what read and write events a thread has made with each field.

In the FOT class diagram an Analyzer class is used to take the *threads*-object from the *Program*-class containing the structured event traces and generates parent and *RowContainer* classes. These classes make up a tree-structure visible in the *TreeTableView* abstracted in figure 4.12.

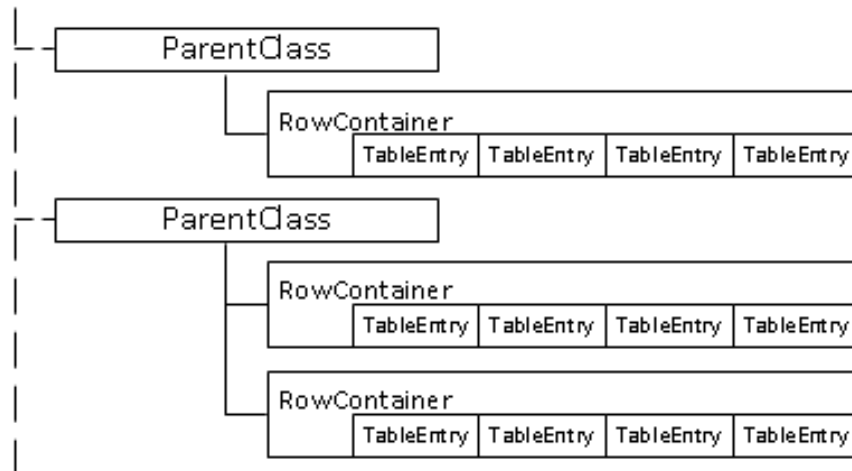


Figure 4.12: FOT View Data Structure.

Every *RowContainer* represents a field. A *ParentClass* represents the class in which that field was defined. To form a table the *RowContainer* class contains a list of *tableEntry* objects that relate to the table elements shown in the *TreeTableView*. The right most columns of classes are the two different views seen in the FOT user interface. The *TreeTableView* displays the data structure generated by the analyzer. This class also initiates the entire flow of execution by calling the *Tracer* class, because it contains the user interface controls. When a row is double-clicked, the associated *RowContainer* is sent to the *DiagramLoader*, which reads the event interleaving and generates a sequence diagram in the *sequence diagram view*.

Chapter 5

Evaluation

In this chapter, we will evaluate the effectiveness of the FOT tool in different ways. The goal is to show how FOT handles some of the problems found in chapter 3. In the next section, we will expose the FOT tool to some test scenarios. Then a visual and functional comparison is made to the tools mentioned in the related work chapter 2.

Table 5.1 shows the case scenario names and which problems are handled where.

	Case 1	Case 2	Comparison
3.1.1 Expressivity	—	—	—
Non-determinism	X		X
Detail and Complexity			X
Context and Environments			X
Information Overload			X
3.1.2 Concurrency	—	—	—
Race Conditions	X		X
Deadlocks		X	X
Starvation and Dormancy		X	X
Synchronization		X	X

Table 5.1: Case Problem Matchup.

5.1 Case Study

We will now show how FOT displays different example scenarios.

The system setup for running all the examples for FOT are the following:

FOT version 1.0 runs on Eclipse IDE, Java Developers edition.

Version: Indigo Service Release 2. Build id: 20120216-1857.

In a plugin resource execution environment with 10Gb ram maximum available for the FOT runtime environment. The JDK used for execution and compilation is jdk1.6.0_25.

5. EVALUATION

The workstation uses two Intel Xenon CPU E5345 @ 2.33GHz, 2327 Mhz, 4 Core(s), 4 Logical Processor(s) with 16Gb of memory. Running on x64 Microsoft Windows 7 professional.

5.1.1 Case 1 Race Condition

Scenario

This case is about the visualization of non-determinism and race conditions. For this demonstration, we consider a simple classic counter problem.

Test program

The test program uses three threads to in sequence read a value, increase the value by one and write the value back. This is of course not done in an atomic way, causing the eventual count to become not $5*3=15$ but some other value. The program itself is listed in appendix C.1. The output of the program is visible in listing 5.1.

```
1 Listening for transport dt_socket at address: 8000
2 Count = 0 Added by: 0
3 Count = 0 Added by: 2
4 Count = 0 Added by: 1
5 Count = 1 Added by: 2
6 Count = 1 Added by: 0
7 Count = 2 Added by: 2
8 Count = 1 Added by: 1
9 Count = 3 Added by: 2
10 Count = 2 Added by: 0
11 Count = 4 Added by: 2
12 Count = 2 Added by: 1
13 Count = 3 Added by: 0
14 Count = 3 Added by: 1
15 Count = 4 Added by: 0
16 Count = 4 Added by: 1
```

Listing 5.1: Case 1 test program Race Conditions output.

It shows 15 lines generated by the threads with a count indicating the current number "count = #" and a thread number to see which thread produced it "Added by: #". As you can see the count reaches 4 in the last line, where it should have been 15.

Applying FOT

Of course, this effect is caused by the interleaving of the commands: *read*, *increase* and *write*. Because of the non-deterministic interleaving if the different threads the order can be changed in to *read*, *read*, *increase*, *increase*, *write*, *write*, or any order of these, resulting in unpredictable end totals. In figure 5.1 we see the sequence diagram output of the count field containing the counted total.

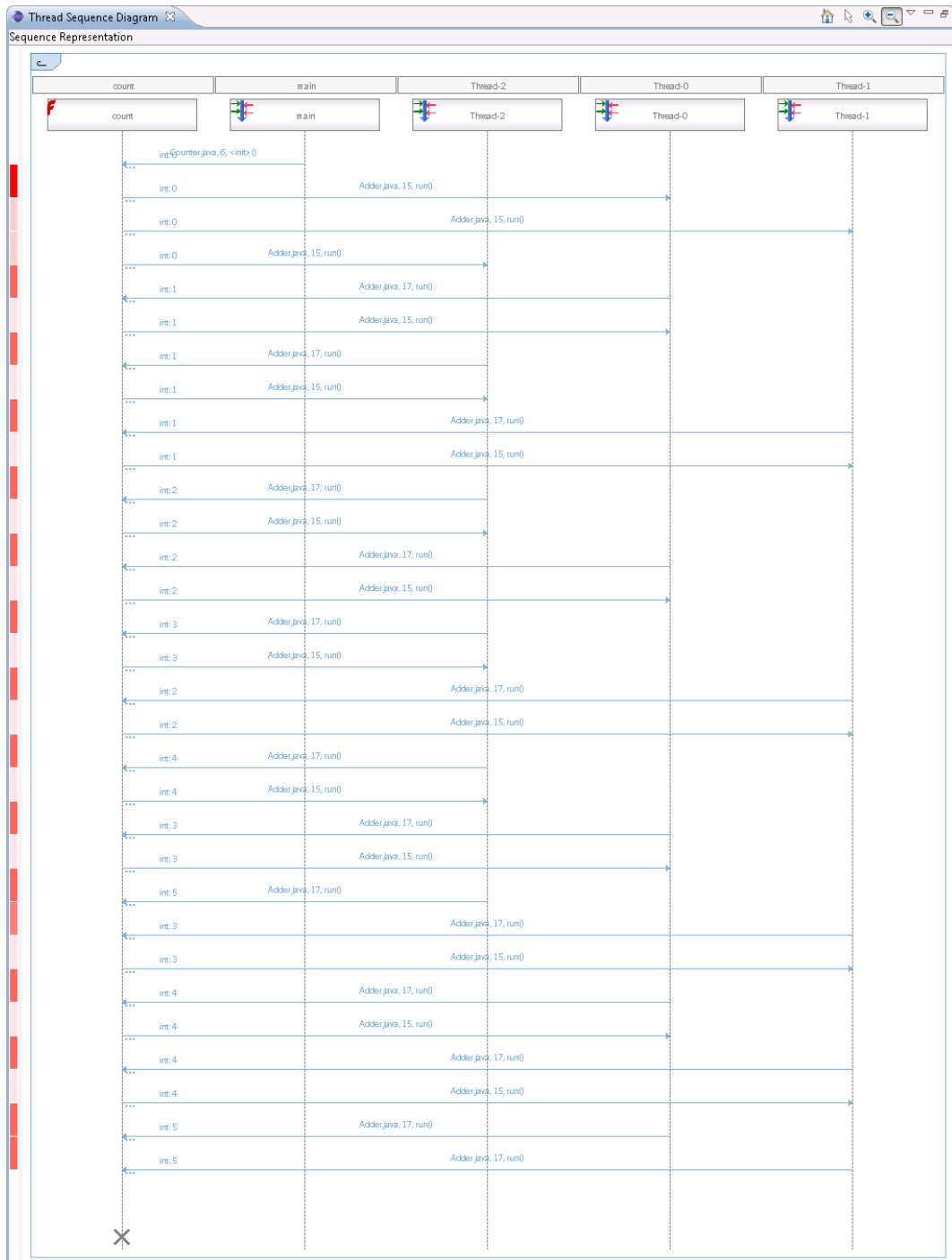


Figure 5.1: FOT sequence diagram of Case 1 Race Conditions.

We can see exactly what read and write interleaving the test program has executed to arrive at the final number. FOT offers an understandable overview of non-deterministic operations and a different perspective on inspecting executions for some atomicity problems and race conditions.

5.1.2 Case 2 Dining Philosophers

The second case is about the visualization of the problems: deadlock, starvation and dormancy.

Scenario

For this demonstration, we consider the famous Dining Philosophers problem. Scenario: A table holds five ohashi (chopsticks) and has food in the center.

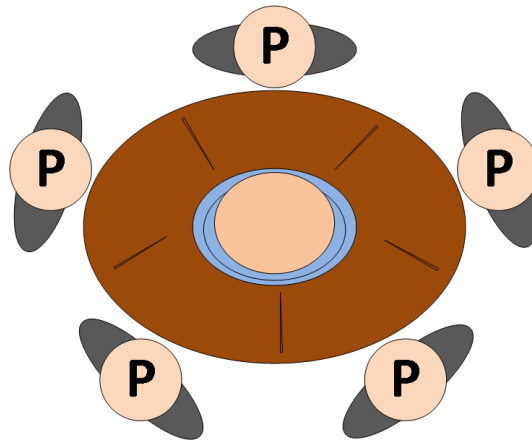


Figure 5.2: Dining philosophers scenario.

Five philosophers are seated around the table. Each philosopher eats for a while, leaves the table to think and comes back to eat, and so on. There are five philosophers and just as many hashi, but each philosopher needs two hashi to be able to eat, one from his left and right. The main problem is that without some coordination or protocol the philosophers could starve.

- All philosophers could hold a chopstick to their left creating a deadlock situation in which all philosophers will starve.
- When one (or more) of the philosophers gets an unfairly small amount of opportunities to eat, causing starvation due to dormancy.

Test Program

Here our aim is not to solve the dining philosopher problem by modifying it so the problems will not occur, but to show how the symptoms can be seen in the execution of an implemen-

tation of this problem by use of the FOT tool. The complete implementation can be found in appendix C.2. A section of the code is shown in listing 5.2. The dining philosophers test program we will use has three classes: a *Table* class which is the main class that creates all objects and sets the scenario. An *Ohashi* class representing each of the five chopsticks on the table.

```

1  class Philosopher extends Thread {
2      private int identity;
3      private Table table;
4      private Ohashi left;
5      private Ohashi right;
6
7      Philosopher(Table table, int id, Ohashi l, Ohashi r) {
8          this.table = table;
9          identity = id;
10         left = l;
11         right = r;
12         System.out.println("Reservation for ID: "+id);
13     }
14
15     public void run() {
16         try {
17             while (true) {
18                 // thinking
19                 table.setPhil(identity, Table.THINKING);
20                 sleep(25 * (int) (100 * Math.random()));
21                 // hungry
22                 table.setPhil(identity, Table.HUNGRY);
23                 right.get();
24                 // got right hashi
25                 table.setPhil(identity, Table.GOTRIGHT);
26                 sleep(250);
27                 left.get();
28                 // eating
29                 table.setPhil(identity, Table.EATING);
30                 sleep(20 * (int) (50 * Math.random()));
31                 right.put();
32                 left.put();
33             }
34         } catch (java.lang.InterruptedException e) {}
35     }
36 }

1  class Ohashi {
2
3      private boolean taken=false;
4      private Table table;
5      private int id;
6      int tst;
7
8      Ohashi(Table table, int id) {
9          this.table = table;
10         this.id = id;
11     }
12
13     synchronized void put() {
14         taken = false;
15         table.setFork(id, taken);
16         notify();
17     }
18
19     synchronized void get()
20     throws java.lang.InterruptedException {
21         while (taken) {
22             wait();
23         }
24         taken = true;
25         table.setFork(id, taken);
26     }
27
28     public boolean isTaken(){
29         return taken;
30     }
31 }

```

Listing 5.2: Case 2 test program sections Class Ohashi and Philosopher

And a *Philosopher* class representing the philosophers and containing the threads performing the task of a philosopher in different phases. These phases are; *thinking* (phase 0); *hungry* (phase 1); *got right hashi* (phase 2); *eating* (phase 3); *got left hashi* (phase 4). The phases are used in the output of the program. An example of the output is given in the listing 5.3.

5. EVALUATION

```
1 Starting Diner
2 Reservation for ID: 0
3 Reservation for ID: 1
4 Reservation for ID: 2
5 0 0 0 0 0
6 Reservation for ID: 3
7 0 0 0 0 0
8 Reservation for ID: 4
9 0 0 0 0 0
10 0 0 0 0 0
11 0 0 0 0 0
12 0 0 0 0 1
13 0 0 0 0 2
14 0 0 0 0 3
15 0 1 0 0 3
16 0 2 0 0 3
17 ~~~~~
18
19 ~~~~~
20 0 3 2 0 3
21 0 0 2 0 3
22 0 0 3 0 3
23 0 0 0 0 3
24 0 0 0 0 0
25 0 0 0 1 0
26 0 0 0 2 0
27 0 0 0 3 0
28 0 0 1 3 0
29 0 1 1 3 0
30 0 2 1 3 0
31 0 2 1 0 0
32 0 2 2 0 0
33 1 2 2 0 0
34 2 2 2 0 0
35 2 2 2 0 1
36 2 2 2 0 2
37 2 2 2 1 2
38 2 2 2 2 2
39 blocked
40 blocked
```

Listing 5.3: Case 2 test program Dining Philosophers output.

Each thread starts announcing "Reservation for ID: #" including their id number. The output gives on one line a simple status number for every philosopher, every time a philosopher changes states. At line 18, lines were emitted. The last four lines show where the program enters a deadlock state. All the philosophers always end up in the *got right hashi* state. Therefore, the last numbered row will show all twos. This is the deadlock state that we wish to visualize. To spot this deadlock state more easily, a "blocked" announcement was added instead of the rows of all twos.

Applying FOT

The tree table diagram figure 5.3, from the trace output from FOT shows all the fields from the Dining Philosophers example.

Fields	main	Thread-0	Thread-1	Thread-2	Thread-4	Thread-3
Ohashi.java						
id	r:0 w:5 c:0	r:1 w:0 c:0	r:1 w:0 c:0	r:1 w:0 c:0	r:1 w:0 c:0	r:1 w:0 c:0
table	r:0 w:5 c:0	r:1 w:0 c:0	r:1 w:0 c:0	r:1 w:0 c:0	r:1 w:0 c:0	r:1 w:0 c:0
taken	r:120 w:5 c:36	r:3 w:1 c:0	r:3 w:1 c:0	r:3 w:1 c:0	r:3 w:1 c:0	r:3 w:1 c:0
Philosopher.java						
identity	r:0 w:5 c:0	r:3 w:0 c:0	r:3 w:0 c:0	r:3 w:0 c:0	r:3 w:0 c:0	r:3 w:0 c:0
left	r:0 w:5 c:0	r:1 w:0 c:0	r:1 w:0 c:0	r:1 w:0 c:0	r:1 w:0 c:0	r:1 w:0 c:0
right	r:0 w:5 c:0	r:1 w:0 c:0	r:1 w:0 c:0	r:1 w:0 c:0	r:1 w:0 c:0	r:1 w:0 c:0
table	r:0 w:5 c:0	r:3 w:0 c:0	r:3 w:0 c:0	r:3 w:0 c:0	r:3 w:0 c:0	r:3 w:0 c:0
Table.java						
ohashi	r:39 w:1 c:23					
out	r:11 w:0 c:0	r:3 w:0 c:0	r:3 w:0 c:0	r:3 w:0 c:0	r:3 w:0 c:0	r:3 w:0 c:0
phil	r:10 w:1 c:3					
state	r:61 w:1 c:36	r:6 w:0 c:4	r:6 w:0 c:4	r:6 w:0 c:6	r:6 w:0 c:6	r:6 w:0 c:4
tst	r:0 w:85 c:36					
untable	r:0 w:1 c:0	r:1 w:0 c:1	r:1 w:0 c:1	r:1 w:0 c:1	r:1 w:0 c:1	r:1 w:0 c:1
Throwable.java						
Unnamed						

Figure 5.3: Case 2 FOT Tree Table.

The reason why the threads 3 and 4 indicated above are not shown in order, even though they were created in order, is due to the non-deterministic nature of the execution and therefore also the tracer. Since each philosopher contains a thread, the threads *Thread-0* through *Thread-4* can be thought of as the five philosophers synonymously.

Deadlock. The test program always enters the deadlock state where all philosophers report state 2. You may expect the field *left* to be highlighted as a collision field due to all the philosophers having their right hashi and waiting on their left. However, the waiting thread is inactive it will read nor write to this value. Therefore, we will not see any collision here. Normally the lack of activity in the sequence diagrams should indicate the problem, and any collision can be seen during interaction with other fields.

To show the deadlock more easily a line was added to the main thread to read and write from every thread: `for(Ohashi h :ohashi)if(h.isTaken() == true)h.tst=0; .` This causes the waiting threads to be shown in the sequence diagram in figure 5.4, in the form of activation bars on each thread's lifeline in sequence.

5. EVALUATION

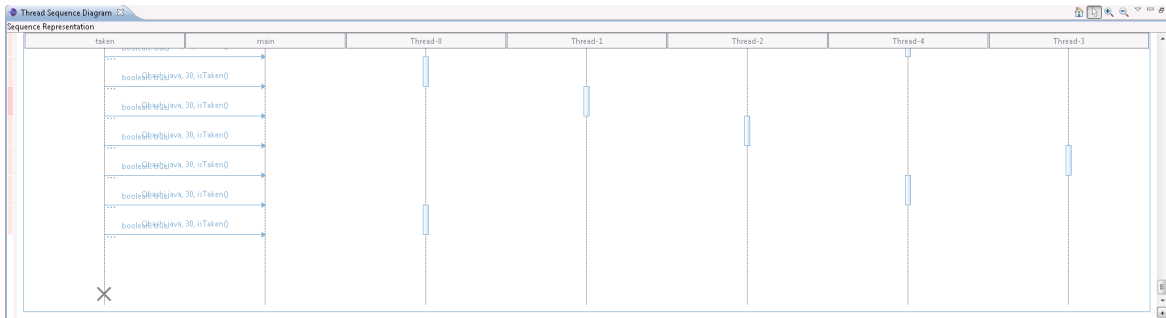


Figure 5.4: Case 2 deadlock scenario FOT sequence diagram "taken" field.

This figure 5.4 shows the *taken* field reading from every thread in sequence and revealing the waiting threads. This effect can also be observed if you were to display all writes to the "ts" field in the FOT sequence diagram view.

Dormancy. Deadlock is spotted in a sequence diagram by lack of interaction, indicating dormancy of the execution. Lack of interaction in this case means, a lack of arrows from or to the dormant or deadlocked thread, indicating that the thread does not read from or write to the observed field. Dormancy or starvation of a single thread is also easy to spot, by lack of interaction with that specific thread. Though were to see this depends upon the program of course. Figure 5.5 shows the *state* field from *Table.java*.

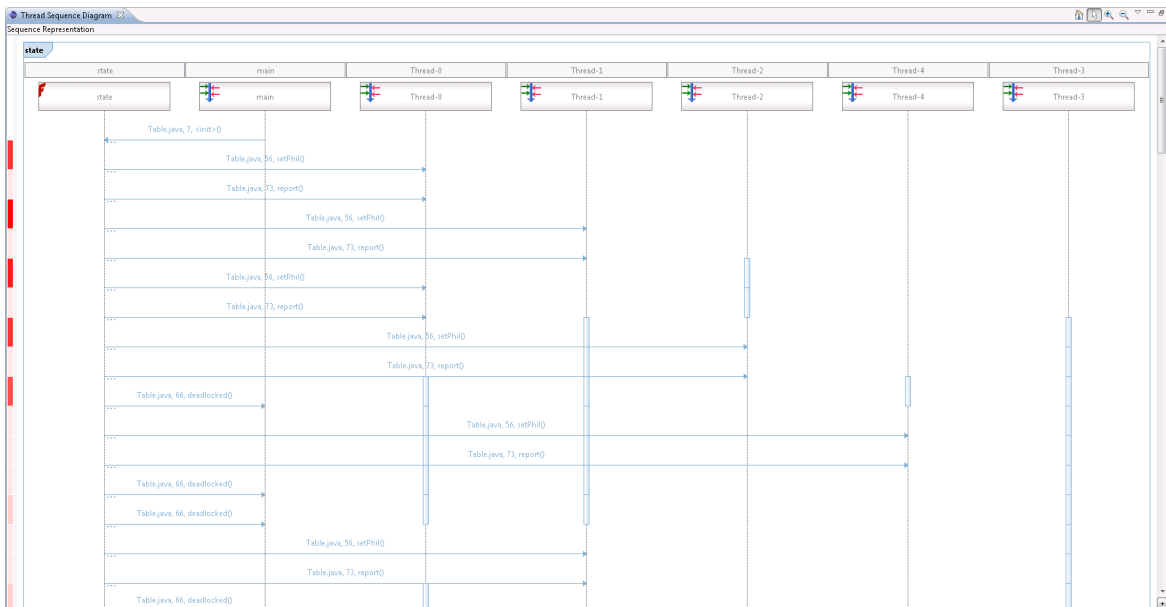


Figure 5.5: Case 2 dormancy scenario FOT sequence diagram "state" field.

Field *state* is updated every time a philosopher goes through and reports a status change.

Observing this field gives an overview of the status changes of every philosopher and their waiting. We can see generally two kinds of messages: *setPhil()* and *report*. *setPhil* updates the status and *report* prints these changes onto the console. *setphil* is shown as a read (a message from the field to the thread) because it might return an exception. *Thread 3*, the thread for philosopher 3, waits for a long time for its turn to change states (eat). So if this situation persists it indicates starvation in all senses of the word and might warrant further investigation if this is not normal behavior. Of course, in our example if we would have scrolled further down we would have seen this thread getting its turn.

5.2 Functional Comparison

We evaluate the tools we mentioned in *Releated Work*, section 2. We take the problems mentioned in the *Analysis*, section 3 and evaluate them next to the FOT tool for their solution or lack thereof. First in table 5.2 we show an overview of the tools and their evaluations. The evaluation for each tool is represented by an O, A or X.

The O indicates that the tool supports a solution for the posed problem.

The A indicates that though there is a solution it is supported by introducing a diagram or visualization other than the sequence diagram.

The X indicates that the tool does not solve or support the posed problem according to the set of criteria.

	FOT	UML	JThreadSpy	JACOT	JIVE	JRET	TPTP	Others
3.1.1 Expressivity	—	—	—	—	—	—	—	—
Non-Determinism	O	X	O	O	A	O	O	O
Detail and Complexity	O	X	O	A	A	X	A	X
Context and Environments	A	A	X	X	A	X	A	X
Information Overload	O	X	X	X	A	O	O	X
3.1.2 Concurrency	—	—	—	—	—	—	—	—
Race Conditions	O	X	O	O	O	X	O	O
Deadlocks	O	X	O	A	A	X	X	O
Starvation and Dormancy	O	X	O	O	O	X	O	X
Synchronization	O	X	O	A	X	X	X	X

Table 5.2: Functional Comparison table.

The reason why we differentiate between the O and the A is: The solution with an O indicates that the criteria were solved by enhancing the sequence diagram. The A indicates that it was solved in a different visualization. As we have posed earlier, a sequence diagram is a low-level high detail tool. Without alternative diagrams, it only provides a limited but clear part of a larger puzzle. This is why "*Context and Environments*" are only solved by

alternative visualization solutions (A). The alternative visualizations mentioned are important for they may lead to superior solutions to problems a sequence diagram is not yet built to solve and show concurrency information not yet integrated in FOT.

Below we discuss the solutions, how FOT solves these problems and the criteria for evaluation. We compare this to UML, which serves as our baseline.

Non-Determinism

Non-deterministic effects of a multithreaded program can only be predicted up to a certain point in a static way. The only way to simulate the actual effects of a program is to trace and execute the program and see what it does, in other words, through use of dynamic analysis [5]. But seeing one execution of the target program does not show non-determinism. The only way to see this is to compare multiple executions and spot relevant differences. This can often be achieved by re-running the execution. Proper use of a sequence diagram can already show a lot about the method execution interleaving and by extension non-determinism. Non-determinism can be noticed by comparing the sequence diagrams generated by different executions. The differences in execution order and or method interaction while executing the same scenario show the non-deterministic nature of the executions. To show enough differences between the executions to solve non-determinism problems a diagram needs to show a certain level of detail.

Criteria:

- The tool must perform dynamic analysis.
- The tool must allow the execution to be re-run.
- The tool must show enough detail so that differences between executions can be spotted.

FOT has detail comparable on the field level. It also shows how the execution comes to each of the values. Since the UML specification was just meant for static design, it has no support for non-determinism. JThreadSpy, JRET, Others, JACOT, TPTP and the others show enough detail to be comparable between the different executions, even though they were not all built for this type of comparison. Though JIVE works like the aforementioned tools, its focus on state stepping and variation in views, makes the sequence diagram simpler by comparison. This makes it more difficult to compare executions at the sequence diagram level, but makes it easier to compare at the object and variable level.

Detail

The tool must provide enough detail about the execution environment to understand what has occurred during execution. The UML1.0 specification is only designed as a high-level design specification and as such does not support any information that pertains to the execution. More added low-level concurrency specific or trace-oriented information could picture the scenario in better detail. The provided information cannot be too transient because this would prevent one execution from being compared to another.

By transient information, we mean for example that the tracing information is not retained

for longer than the state that produces this information persists in the VM, like JACOT's state diagram. Criteria:

- More detail needs to be shown than the standard UML 1.0 sequence diagram.
- Insight into lower level details of the programs execution useful for understanding or debugging of the program.
- Providing more than transient information during the execution tracing that cannot be compared to other executions.

FOT and JThreadSpy add thread-status and concurrency related information to the sequence diagram. JACOT, JIVE and TPTP add various alternative views to show extra detailed information. Particularly JThreadSpy adds a lot of concurrency details to one sequence diagram in the form of concurrency oriented indication icons. TPTP shows a lot of different detail information through multiple visualizations.

Context

The sequence diagram is a powerful but high detail tool, and because of that high detail, the information needs to be put into context by another view to be interpreted correctly. Referenceable naming must be used to be able to link the high level, low detail, abstract diagram to the low level, high detail execution and the other representations.

Criteria:

- Does the other view provide the references, naming and scoping needed to put the visualized elements in a proper context?
- Can the information provided by sequence diagrams and other views be linked to the implemented code?

The UML specification defines many diagrams that can be used to define context on different levels. Nevertheless, in the context of execution and automatic generation they are rarely useful for understanding concurrency. JThreadSpy focusses specifically on smaller scenarios and analyzing them with higher detail, choosing not to use any alternative visualization. FOT strives to work on much the same level of detail, but uses in the interest of the limiting of information the *Table View*. JACOT does use an alternative view but chooses to go more in detail with it instead of less, probably with the same intention as JThreadSpy. JIVE and TPTP give many different views that can provide integrated on-demand information on different levels ensuring good context awareness.

Information Overload

Much data can be generated by tracing a program execution. Displaying all this information will not improve understanding of what is happening, unless only the necessary information is presented. Information must be managed so not to overwhelm the user, it must be structured or provided when requested [25].

Criteria:

- Is information provided as needed?

- Can excess information be hidden?
- Can the visualization be shown on different levels of granularity?
- Is the information structured in a human readable way?

JRET was specifically designed to limit information and to display sequence diagrams as concise as possible, but more concise is not always better. JRET's compatibility with concurrent analyses is limited by the automations that make the condensing of information possible. FOT hides the information in a structure to ensure that the information can be requested when needed. JIVE and TPTP prevent information overload by spreading it over different views and linking them together.

Race Conditions

By use of a sequence diagram, race conditions can be observed, when using multiple executions. Like with non-determinism, when you compare generated sequence diagrams between different executions, you can spot the difference in interleavings. If these interleavings cause the output of the execution to change, you have a race condition. The diagram should enable deducing the problem causing the race condition. The sequence diagram must differentiate between the different threads of execution to discern which threads partake in the race condition.

Criteria:

- Must see the difference between the threads of operation and what they are doing.
- The order of their operations must be visible.

All tools except for UML and JRET can show sequence diagrams that can differentiate between threads enough to show the effects of race conditions on the operation interleavings. To successfully diagnose a race condition, checking the in-and-output of the field in question may also be necessary. This function is included in FOT in the form of labels showing the value of simple types along the field's lifeline. This functionality is not a requirement in our evaluation because there are other ways to monitor in-and-outputs.

Deadlocks

Dormancy can be seen by use of a sequence diagram that includes thread information, but relating this to a deadlock requires more information.

Criteria:

- Can deadlocks be properly traced?
- Can deadlocks be visualized?
- Can the cause behind the deadlocks be discovered and diagnosed?

JThreadSpy features the most extensive augmentation to the sequence diagram for showing synchronization and locking mechanisms. It shows the most verbose way to display a deadlock situation in a sequence diagram. FOT can show the threads that are waiting for the same objects thus showing the deadlock. The JaVis [21] tool in the *others* category is specifically designed to detect deadlocks and display the so-called *path of crime* leading

up to the deadlock in a sequence diagram. JACOT does much the same as FOT, indicating the status of the threads through the life of the sequence diagram. Jive records every action pertaining to the execution for forward and backward stepping analyses. Though Jive's visualizations were not built to specifically show deadlocks, diagnosing the problem when a deadlock occurs, is possible in a debugging kind of way.

Starvation and Dormancy

Starvation and dormancy can be visualized by a sequence diagram with thread information. It is important that the balance between threads can be seen and for sections of the sequence diagram, where the balance is disturbed a reference, back to the situation in the execution can be made.

Criteria:

- Must see the difference between the threads of operation and what they are doing.
- Must see if the threads are operational.
- Must see frequency of operations of each thread.
- Can the cause behind the lack of operation be discovered and diagnosed?

FOT, JThreadSpy, JACOT and JIVE all support starvation and dormancy detection in much the same way through thread support. TPTP has an extra sequence diagram with a thread-centric perspective, and a thread visualizer to help with detecting starvation and dormancy. The addition of timing information in FOT, TPTP and JThreadSpy helps with finding and gauging starvation.

Synchronization

For solving deadlocks, dormancy and other problems added functionality pertaining to concurrency can be useful.

Criteria:

- Must add concurrency or synchronization related information to the diagram.

JThreadSpy adds the most extensive monitor contestation information to its sequence diagram. FOT adds only information about threads in waiting for object monitors. JACOT adds more general thread status information to a separate state diagram.

5.3 Limitations

5.3.1 Related Works Selection

The included related works form the basis for the thesis and the resulting FOT tool. If different works were selected for the analysis, the thesis could perhaps have had a different result. It was not possible to include all available works in the related work, excluding all possibility for bias or oversight, so a selection was made based on criteria ranking all the works and selecting the top eight. The related work mentioned in this paper was selected on basis of the following criteria:

- The work had to involve or mention dynamic analysis as a method for acquiring information.
- The work had to be oriented on or as closely related to concurrency as possible.
- The work had to be referenced by other work as much as possible, indicating the importance of its content.
- Older works may have been included due to their importance, serving as a good basis and comparison. Examples are UML and the works included in the "Others" section.
- Newer works were selected on the basis of their innovations not only in the field of concurrency but also in the general use of sequence diagrams.
- Some works may have been excluded on the basis difference in focus or resemblance to an already included solution.
- Some works were excluded because the posed method or solution seemed impossible to mimic, integrate or reconstruct. In other words, they were overly complicated for our aim to create a simple solution.

5.3.2 Problems Selection

The bias that could have been present with the selection of the related works could influence the problems found in the analysis in the same way. Most of the problems distilled from the related works were problems they posed, or aimed to solve. However, the problems posed in the analysis section were not all problems that were mentioned. Our goal was not to solve all the problems mentioned in the related work, just the ones that lay within our focus and objectives. To show our methods for problem selection we mention JIVE. One of the reasons for including JIVE in the related work is because it has well defined "properties" to which successful interactive visualization has to adhere, that can be related to the problems they try to solve:

- **Depict Objects as Environments.** This was incorporated as the problem *Context and Environments*.
- **Provide Multiple Views of Execution States.** This was incorporated in the *Detail and Complexity* and alternative visualization research.
- **Capture History of Execution and Method Interaction.** This was already cleared by the basic requirement of the use of sequence diagrams.
- **The support for forward and backward execution.** This was one of the requirements that could solve many problems but was incompatible with our concurrency focus. As previously mentioned in [4.1] when using multicore systems backward execution can cause too many problems.
- **Support Queries on the Runtime State.** This property we perceived as an attempt to minimize information storage by requesting details on-demand. Though this is partly addressed in the information overload section, this also reaches outside our focus by trying to let the user decide what information is relevant for understanding. We are trying to find a way to provide this information off-hand as digestible as possible.
- **Produce Clear and Legible Drawings.** This is handled by the information overload problem. In JIVE the focus is put on the drawings/diagrams for JIVE as many of

those. But since we focus on existing sequence diagrams as a pre-defined concept we only consider use of added information.

- **Use Existing Java Virtual Machine.** Tough this is a good requirement that was used in the production of FOT, it does not negate the advantages of other methods featured in other tools.

In this manner, through the evaluation of all features and/or problems of the related works we had come to the problems shown in the *Analysis* [3] section. As this example shows, we cannot eliminate the bias for this selection completely, because of the subjective interpretation of the focus. Therefore, we emphasize here that this are not the only, correct or most important problems involving concurrency visualization, but they were instrumental in the conception of FOT.

5.3.3 Evaluation

As the related work selection and the problem refining is open to bias the evaluation criteria can be seen as biased, unfair or inaccurate. To minimize bias when evaluating the different tools we took the following steps:

- We used the problems from the problems section [3.1] as a basis, even though this leaves open the risk of confounding the bias problem.
- We incorporated as much of the requirements pertaining to these problems from the related works as possible.
- We tried to make the criteria as crisp as possible so even if the reason for a criterion is in question its use in the evaluation is clear.
- Even if its use in evaluation might be unclear we tried to evaluate all tools in exactly the same manner.

It needs to be said that because it is an evaluation it remains partially subjective. Though we used all attainable information on these works in the research and evaluation, due to ageing of the technologies or lack of program code, we could not actually run and compare all tools described in the related works. Ideally, we would have preferred to run all tools under the same conditions on the same test code and compare the results.

5.3.4 Threat to external validity

In section 5 FOT was applied to two cases that show the usefulness of the FOT tool in those situations. It can be argued that the usefulness in these situations is not always as high outside these situations. The functionality of FOT can be argued to be limited to the cases. Though not many cases have been provided in section 5. We have chosen cases that are as representable as possible for the problems that have been described in section 3.1. The cases represent almost the definition of the problems they describe. Though they are left simple to keep them understandable, they are as comparable to a real problem as we can make them within the performance confines of FOT.

5.4 Discussion/Reflection

Extensibility Originally the FOT visualization was intended to show in addition to the method causing the trace event a trace of the methods call stack. This feature is still partially implemented in FOT but has not been disabled because the TPTP sequence diagram generator [1] and visualization is not equipped to provide references to the messages plotted that could point to the method call stack. Leaving this function enabled could cause extra delay while tracing, because it extends the tracer to record every method entry and exit as we have seen done in JThreadSpy [17]. Instead, FOT currently makes due with naming the message with the defining filename, the line number and message name. If the sequence diagram visualization is extended or changed in the future a reference to a generated method call stack may still be a worthy addition for added understandability.

Generalizations and Distinctions FOT makes some very specific generalizations and distinctions in its analysis phase. One simple generalization is that fields are generalized to the same class, as can be seen in the table tree view. So FOT makes its distinction on class level. Ok, now for a more problematic generalization of values to fields. In the sequence diagram is shown what value a Field has. The problem is that the Field is a definition of one or more instances (values) in the execution, so multiple values can in some cases be shown as one field.

We also make distinctions between the threads that interact with each field. Now we do not know for sure if one event instigated by a thread and another event instigated by the same thread shown sequentially in a sequence diagram are in fact about the same value. We can only be sure that they share a definition and therefore functionality with each other. If there are two fields that share a name but have a different definition, both are displayed. This means that in the sequence diagram view there can be variance in the value display that does not coincide with the change events when there are more than one instance of a field being traced. It also makes, distinguishing multiple instances in different threads in general, still problematic.

This distinction can still be created by adding another layer to the table tree. If list up all instances of a field and list those as children of a field in the tree table, the instance-difference can be shown. This could be very verbose and confusing for first time users if implemented but, can be a nice addition for future versions.

The waiting threads, shown as activations in the sequence diagram, only distinguish on class basis. So even if the shown field is not the reason for the lock causing the wait. If the thread waits for something in the entire class, an activation bar is shown. This is caused by the way JDI addresses the waiting thread pool. On the upside, this makes it easier to analyze blocking situations from different angles or field perspectives. On the downside, it makes it harder to pinpoint the culprit causing the block within a class. Because this feature was intended as one of many possible concurrency indicators addable to the sequence diagram view it suffices as a demonstration.

Event driven tracing The FOT tool only shows status information at the times events occur. In some situations it can be preferable to show status information, because otherwise you'll lack pieces of the general view. Reference events, big problems or other things might be lacking because no read or write events are taking place at that time. This is something to keep in mind while using the tool. This property can be viewed as much as a feature (against information overload) and a flaw (information loss), and is the result of the analysis approach and the JDI feature set.

Chapter 6

Conclusions and Future Work

This chapter gives an overview of the project's contributions. After this overview, we will reflect on the results and draw some conclusions. Finally, some ideas for future work will be discussed.

6.1 Contributions

The key contributions of this thesis are an evaluation of current dynamic analysis based tools with concurrent execution visualization. We have made an analysis of the useful functions of these tools and used these to develop a new field centric concept for visualizing multi-threaded behavior. With this concept, we introduce an alternative sequence diagram design incorporating the field centric concept. This new concept is meant to add a new low-level perspective to concurrency oriented dynamic analyses for visualizing shared memory based thread interaction. Many of the best concepts and functions of previous tools, combined with the field centric design have been made into a new tool called Field Of Threads. We also include a functional comparison of all tools including Field Of Threads, of their capability to trace, analyze and visualize the execution of multithreaded concurrent programs.

6.2 Conclusions

In this thesis we try to improve the way running threads, thread's functionality and interaction are represented and visualized so that they can be understood more easily. We have done this by answering the following questions and creating a tool demonstrating the results named Field Of Threads (FOT).

RQ1 How are threads visualized and analyzed now?

Answer 1 Threads can be visualized through tracing the execution of test programs. These dynamic analysis tools analyze the execution and generate collaboration, sequence and other kinds of diagrams based on the UML specification. The more advanced tools incorporated new functions on top of the UML specifications extending them to better show concurrency information.

RQ2 What are the major problems impeding thread comprehension?

Answer 2 The found problems ranged from:

- The **expressivity** of the visualization.
 - * **Non-Determinism** causes differences between executions confusing the programmer.
 - * **Detail and Complexity** need to be balanced, to provide enough detail about the execution, to understand timing communication and interleaving without making the visualization too complex to understand.
 - * **Context and Environments** need to be made clear to the user so any information produced by diagrams and visualizations can be linked to implementation and design.
 - * **Information Overload** occurs when the large amount of information generated by tracing an execution is not properly filtered or structured.
- **Concurrency's** inherent problems.
 - * **Race conditions** may cause the outcome of executions to depend entirely on execution order.
 - * **Deadlocks** can occur if multiple threads are waiting for each other to finish.
 - * **Starvation and Dormancy** can occur when concurrency causes threads' execution to become unevenly balanced or fall dormant for too long.
 - * **Synchronization** needs to be used properly in concurrent programs. If used improperly this can create complex problems that are hard to analyze.

RQ3 What kinds of operations are the current representations of threads useful for?

Answer 3 The most useful elements that make up a successful concurrent execution visualization are:

- The **UML Sequence Diagram** is a good extensible basis for showing interaction between objects. It also gives the most comprehensive timeline for interleavings and execution order.
- **Colored threads** offer a way to distinguish different entities of execution and scopes. These create a better reference to the execution environment.
- **Timing info** creates a discrete timeline to allow analysis of time related problems like dormancy and starvation. It better relates the execution events to the output of the execution.
- **Variable granularity** allows the user to both get a grasp of the larger context as well as the specific details while retaining a good relation to the execution environment and the implementation.
- **Alternative visualization** provides a different dimension, usually a better overview, to the provided information, creating references and links between the sequence diagram, execution environment and implementation.
- **Added indicators and icons** add extra useful expressions of synchronizations and other concurrency concepts, making debugging and analysis more intuitive.

RQ4 What elements provide the best way to view, analyze and comprehend thread execution?

Answer 4 The best way to view, analyze and comprehend thread execution consists of:

- A tracer that can observe and record the execution of a test program.
- An analyzer that can refine thread behavior from raw execution data.
- A concurrency augmented sequence diagram visualization.
- An added generalized visualization to place the sequence within the whole structure.

RQ5 How can the representation of threads be improved further?

Answer 5 Thread representations can be improved by shifting away the focus from the thread, even the objects within the threads and focusing on how the different threads manipulate their common fields as defined in the code. Simplifying the gathered data and bringing the representation closer to the definition in the code.

The Java tool FOT incorporates the field-centered approach resulting from the 5th answer. Its objective is to trace the execution of a test program, analyze its inter-thread communication and visualize the execution in a field-centric way for detection and diagnosing problems. FOT uses the Java Debug Interface to trace small Java programs' execution, analyses and displays this in an Eclipse Platform visualization plugin.

6.3 Future work

6.3.1 Concurrency is more complicated.

FOT is attempting to simplify the visualization of concurrent executions. However, the visualizations of FOT were devised with more potential in mind. The sequence diagram's features provided by the TPTP visualization library are almost saturated by the field centered visualization in FOT. The alternative sequence diagram was envisioned to become more extensible than this. For instance, the waiting thread feature that is shown now as activation bars is only one of many possible concurrency indicators possible for this view. If we look back at JThreadSpy, we see that there is more information that can be collected and displayed to show concurrency and the monitors used to solve concurrency problems. Adding more of these is something for future work. The reason these functions were not added earlier is that this would probably mean that the tracer would need to register many more types of events besides reading and writing fields. Causing further slowdown and making it harder to combine the two message types. The problem would be that a registered event would occur before, after (or coincide with) a field event. In which case, it would become hard to manage where a status change would belong. The causality of events would become harder to visualize.

6.3.2 Table Tree Features

Initially the Table Tree visualization was not meant to only change color and indicate collision fields but also to detect and indicate more concurrency-based problems. So they can quickly be analyzed with the sequence diagram view. An example is detection of dirty reads and writes, deadlocks [12], lack of atomicity [8] etc. Nevertheless, the addition of more intelligent analyses is something for future work.

6.3.3 Sequence Diagram Comparison

To diagnose non-determinism problems like for instance race conditions it is better to take two executions and evaluate the differences. This way allows you to see the effective variation caused by non-determinism instead of just the static (expected) difference. FOT still lacks the functionality to retain and compare sequence diagrams. FOT only supports one sequence diagram at a time. Supporting multiple sequence diagrams and automatically comparing commonalities and differences between generated sequence diagrams from different executions can be solved in future work.

6.3.4 FOT time and memory (in) efficiency

FOT causes slowdown, which is inherent to the field-centric approach of FOT. Where normally information about fields and threads is requested from a running VM, FOT collects and stores all the data in advance. For every field instance and for every field access or mutation, a lot more memory space and processing time are being used to observe that field.

This causes the slowdown and the use of memory space. Some of the problems and possible solutions are discussed here for possible future work.

Memory space

Large (complicated) programs cannot be traced by FOT. The memory use of FOT is related to the number of events that have occurred in the execution. No matter how many fields or threads are active in the execution the number of read and writes decides the most of the memory use. However, this is relatively unrelated to why FOT does not support large programs. When the JDI tracer starts, it sets watch points to all loaded and to-load fields. This operation is very slow and causes the tracer to disconnect from the VM if it takes too long. Even if this phase completes safely, JDI has the knack of losing track of the main class in the process, when the suspend option is used in the execution arguments of the test program. It is possible to trace a test program without use of the suspend option, but it causes FOT to start tracing halfway through the execution. Because the test program has already created and used many fields and is changing them constantly while executing, the initialization phase of FOT usually takes even longer to set the watch points and often fails anyway. These problems are a result of JDI operation and might be circumvented or fixed in the future.

Disk storage

Even small programs can need a lot of memory to run if a lot of activity is generated. But disk based storage is not yet necessary because this type of use would cause any sequence diagram in FOT to contain so much data that it will cause an information overload. Something we were trying to avoid to begin with (section 3.1.1). Though it may be possible to split of the monitor event data from the field and thread data (which are necessary in memory for speed and lookup purposes) and write only the monitor event data to disk, thus saving memory in future versions of FOT.

Slowdown

Batching Because FOT is monitoring and storing during the execution of a test program the execution slows down. This was expected when the choice for JDI as a tracer was made. Some techniques could be employed to minimize the slowdown. Batching of trace events could cause grouping of delays and speed up tracing. FOT is not yet prepared for this type of optimization. Because right now the timings are being measured as soon as possible, after events occur. If at the same time other values are collected, it will cause delays and skewing of timing information. Batching before this process would mean that these timings would be influenced adversely and the value references could become obsolete. Batching afterwards would result in minimal speedup.

Multithreading Further multithreading of the tracer could possibly speed up the tracer but this creates problems because multiple threads would be using one trace-data storage,

6. CONCLUSIONS AND FUTURE WORK

it would become difficult to determine timing, the order and thus causality between events [27]. But further experimentation could yield faster tracing.

JVMTI Using the knowledge from FOTs JDI tracer it might be possible to create a faster JVMTI-based tracer in future work. Because they both are based on the JNI interface swapping out the JDI tracer for a JVM-TI tracer should be possible and faster. For more (dis)advantages, see section [3.2].

Focusing A possibility for a faster tracer is focusing the tracer in advance on one class. It is possible in JDI to only place watch points on one class within the test program, thus filtering away the other classes. Whether this approach speeds up tracing depends on the way JDI is implemented. An added advantage of this approach is the amount of unnecessary information in the FOT visualization is greatly diminished. A disadvantage is that the problematic class must be entered in advance of the trace, which is not a big problem when multiple traces are made. A sneaky problem is the fact that only tracing one class can have an adverse effect on the fairness of the test programs. Slowing down the threads that use the specified class only, may cause balance problems like starvation and other interleavings that would not occur otherwise. This can cause problems to be depicted where there is none. The timings would also become skewed.

Bibliography

- [1] Linux tools project.
- [2] Luay Alawneh and Abdelwahab Hamou-Lhadj. Execution traces: A new domain that requires the creation of a standard metamodel. *Advances in Software Engineering*, pages 253–263, 2009.
- [3] D. Brock. *Understanding Moore’s law: four decades of innovation*. Chemical Heritage Foundation, 2006.
- [4] Y. Cheon and G. Leavens. A simple and practical approach to unit testing: The jml and junit way. *ECOOP 2002 Object-Oriented Programming*, pages 1789–1901, 2006.
- [5] Bas Cornelissen, Andy Zaidman, Arie van Deursen, Leon Moonen, and Rainer Koschke. A systematic survey of program comprehension through dynamic analysis. *Software Engineering, IEEE Transactions on*, 35(5):684–702, 2009.
- [6] J. desRivieres and J. Wiegand. Eclipse: A platform for integrating development tools. *IBM Systems Journal*, 43(2):371–383, 2004.
- [7] T. Eclipse. Eclipse test & performance tools platform project, 2007.
- [8] Cormac Flanagan and Stephen N Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. *ACM SIGPLAN Notices*, 39(1):256–267, 2004.
- [9] Martin Fowler and Kendall Scott. *UML distilled (2nd ed.): a brief guide to the standard object modeling language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.
- [10] P. Gestwicki and B. Jayaraman. Methodology and architecture of jive. In *Proceedings of the 2005 ACM symposium on Software visualization*, pages 95–104. ACM, 2005.
- [11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of aspectj. *ECOOP 2001 Object-Oriented Programming*, pages 327–354, 2001.

- [12] Byung-Chul Kim, Sang-Woo Jun, Dae Hwang, and Yong-Kee Jun. Visualizing potential deadlocks in multithreaded programs. *Parallel Computing Technologies*, pages 321–330, 2009.
- [13] E. Kuleshov. Using the asm framework to implement common java bytecode transformation patterns. *Aspect-Oriented Software Development*, 2007.
- [14] H. Leroux, A. Réquilé-Romanczuk, and C. Mingins. Jacot: a tool to dynamically visualise the execution of concurrent java programs. In *Proceedings of the 2nd international conference on Principles and practice of programming in Java*, pages 201–206. Computer Science Press, Inc., 2003.
- [15] Bil Lewis. Debugging backwards in time. *arXiv preprint cs/0310016*, 2003.
- [16] G. Malnati, C.M. Cuva, and C. Barberis. Jthreadspy: teaching multithreading programming by analyzing execution traces. In *Proceedings of the 2007 ACM workshop on Parallel and distributed systems: testing and debugging*, pages 3–13. ACM, 2007.
- [17] G. Malnati, C.M. Cuva, and C. Barberis. Jthreadspy: A tool for improving the effectiveness of concurrent system teaching and learning. In *Computer Science and Software Engineering, 2008 International Conference on*, volume 5, pages 549–552. IEEE, 2008.
- [18] K. Mehner. Javis: A uml-based visualization and debugging environment for concurrent java programs. *Software Visualization*, pages 643–646, 2002.
- [19] A Moffat. Sequence itymbi ..., 2004.
- [20] Inc. Object Management Group. *Superstructure specification*, October 2011.
- [21] R. Oechsle and T. Schmitt. Javavis: Automatic program visualization with object and sequence diagrams using the java debug interface (jdi). *Software Visualization*, pages 672–675, 2002.
- [22] K. OHair and J.J. Heiss. The jvm tool interface (jvm ti): How vm agents work. *Web page, Dec*, 2006.
- [23] Oracle. Java platform debugger architecture structure overview, 2011.
- [24] CK Prasad, R. Ramchandani, G. Rao, and K. Levesque. Creating a debugging and profiling agent with jymti, 2004.
- [25] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Visual Languages, 1996. Proceedings., IEEE Symposium on*, pages 336–343. IEEE, 1996.
- [26] M Strauch. Quick sequence diagram editor, 2008.
- [27] Eleni Stroulia and Tarja Syst. Dynamic analysis for reverse engineering and program understanding. *ACM SIGAPP Applied Computing Review*, 10(1):8–17, 2002.

-
- [28] R. Voets. *JRET: A tool for the reconstruction of sequence diagrams from program executions*. PhD thesis, Masters thesis, Delft University of Technology, 2008.
- [29] S. Xie, E. Kraemer, REK Stirewalt, L.K. Dillon, and S.D. Fleming. Design and evaluation of extensions to uml sequence diagrams for modeling multithreaded interactions. *Information Visualization*, 8(2):120–136, 2009.
- [30] Kwok Yeung, Paul HJ Kelly, and Sarah Bennett. Dynamic instrumentation for java using a virtual jvm. *Performance analysis and grid computing*, pages 175–187, 2004.
- [31] Andreas Zeller and Dorothea Lutkehaus. Ddd a free graphical front-end for unix debuggers. *SIGPLAN Not.*, 31(1):22–27, January 1996.

Appendix A

Glossary

In this appendix we give an overview of frequently used terms and abbreviations.

Agent: A library added to an application for registering its execution.

Argument: An object passed to a method when the method is called.

Atomicity: Whether or not a set of operations are executed in sequence without being subject to non-determinism. An atomic operation will appear to be executed instantaneously.

Collision: A collision or clash is a situation that occurs when two distinct references have the same hash value or abbreviated name.

Concurrency: A property of systems and programs in which several computations are executed simultaneously while interacting with each other.

Delta (states): Delta indicates difference between one and the other. In the case of states this indicates the amount of difference between one state and the other.

Eclipse: A project aiming to provide a universal tool set for development.

Exception: An anomalous or exceptional situation requiring special processing by a handler.

Field: A definition of an object, class or variable within a class definition.

Getter / Setter: Accessor and mutator methods for referencing protected variables encapsulated in classes.

Granularity: The extent to which a system is subdivided, the level of detail shown.

GUI: Graphical User Interface.

Hash: A semi unique numerical representation of an object or value.

HashMap: Data structure containing key-value pairs.

Instrumenting: Inserting tooling code into a program to change or add to the programs functionality.

Interleaving: The order in which operations are executed.

Lifeline: A part of an UML sequence diagram indicating the life of a single class.

Message: A part of an UML sequence diagram shown as an arrow between lifelines indicating the use of a message in a class from another class.

Method call: The execution of a method of another object or environment.

Method call stack: A call stack is a data structure storing information about active program subroutines.

Monitor: In concurrent programming, a monitor is an object or module intended to be used safely by more than one thread. The defining characteristic of a monitor is that its methods are executed with mutual exclusion.

Mutex: Mutex is short for mutual exclusion. Meaning that, at each point in time, at most one thread may be executing this operation.

Profiler: A programming tool that can track and quantify the performance of a computer program.

Stepping: A debugging operation which continues execution for one operation. Stepping forward means execution the next operation, stepping backward means returning to the previous one.

Tracer: A program that is specialized in logging or recording information about a programs execution.

VM: Virtual Machine, A virtual environment that can execute byte code programs. JVM stands for the Java Virtual Machine but called VM here.

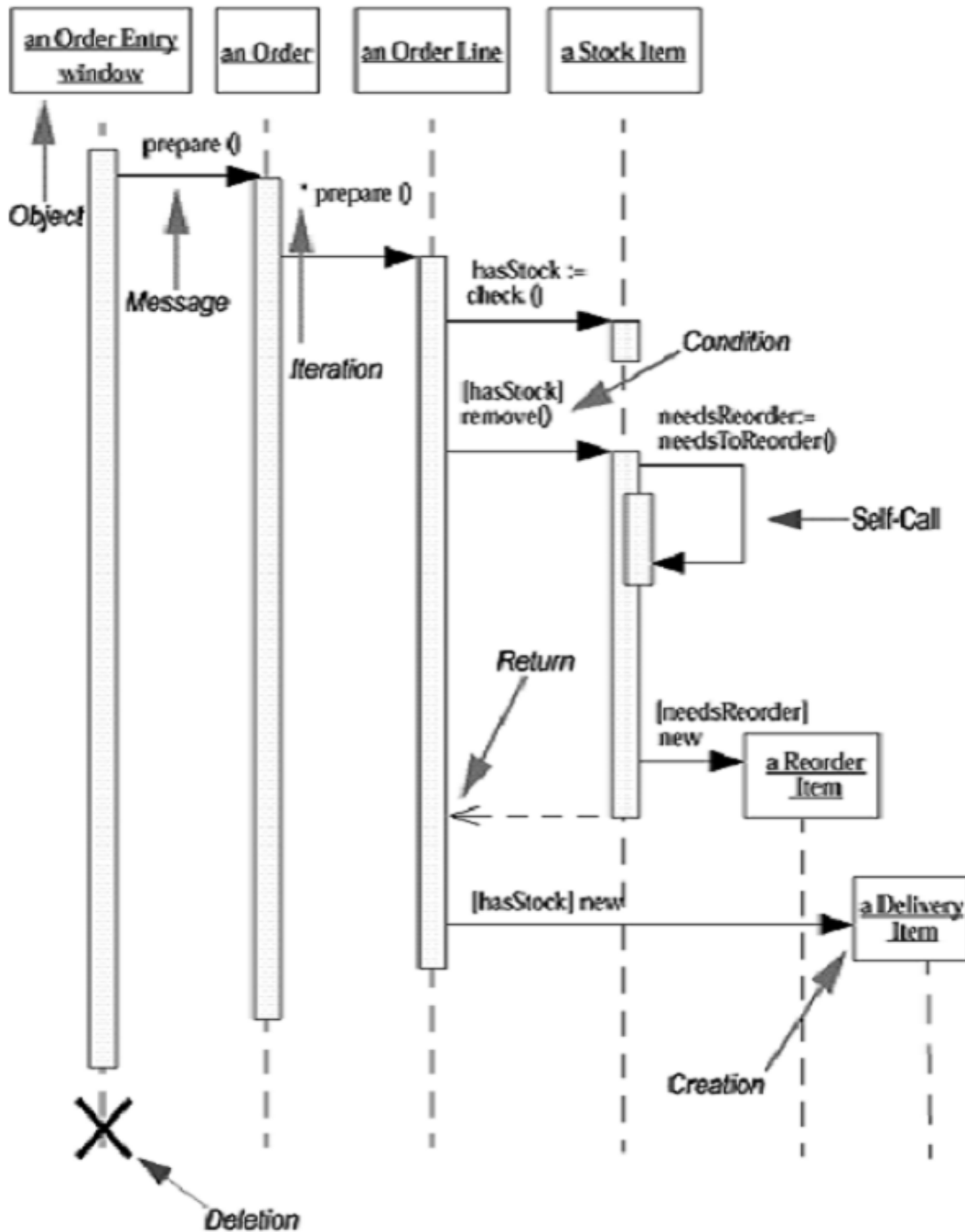
Watch-point: The registration of the monitoring of a field or object.

XML: Extensible Markup Language is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable.

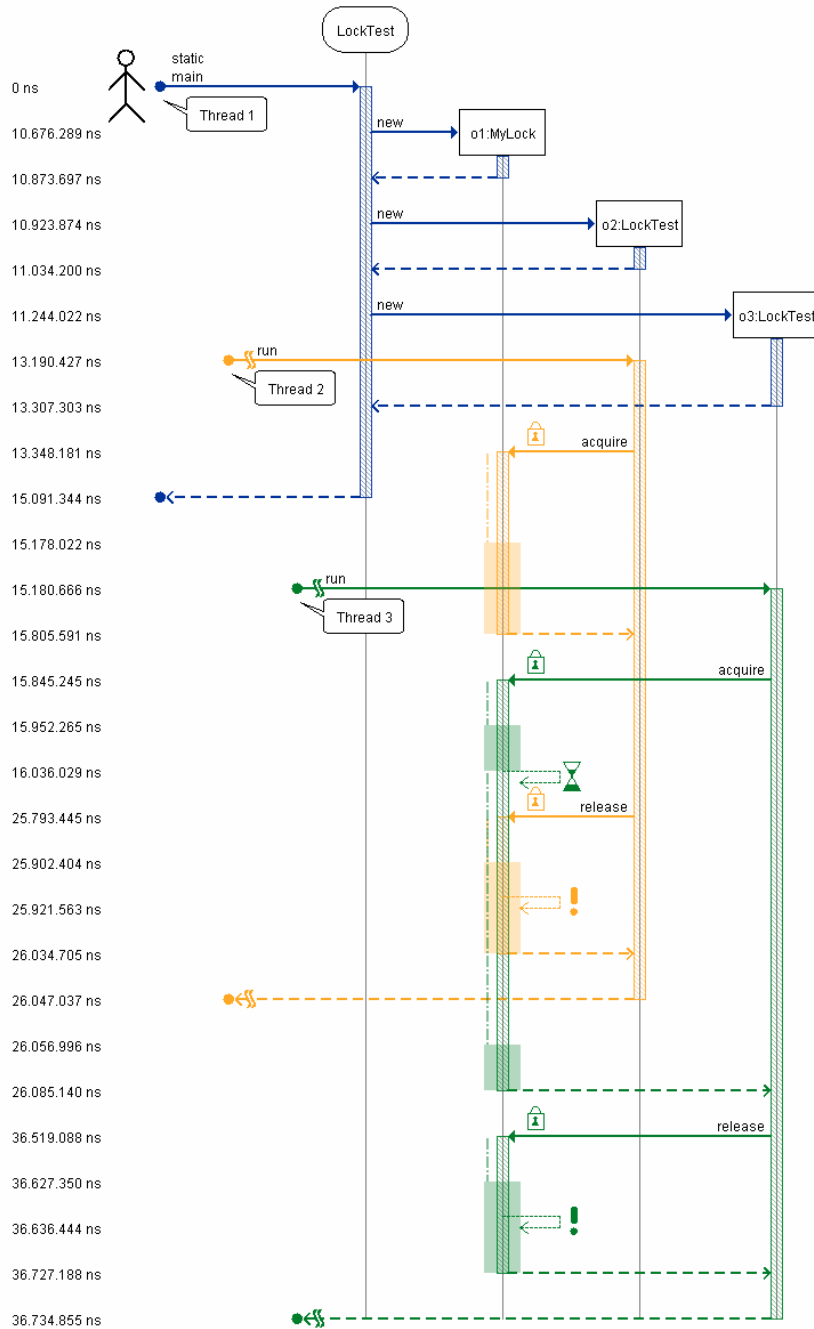
Appendix B

Related Works Visualizations

B.1 UML1.0 Sequence Diagram



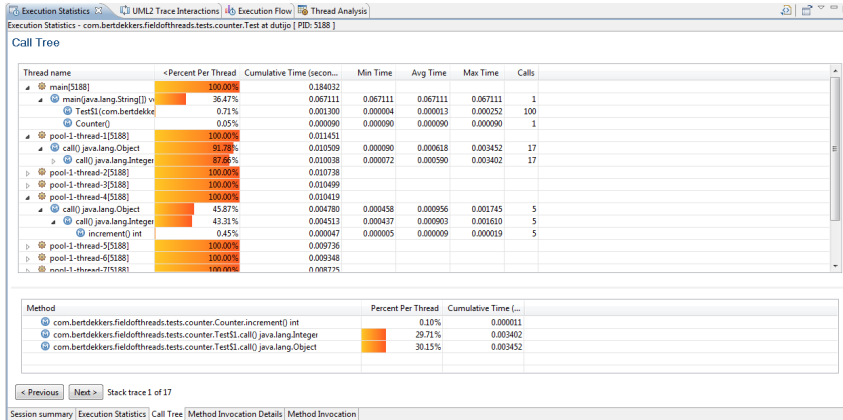
B.2 JThreadSpy Sequence Diagram of Threads competing for a lock.



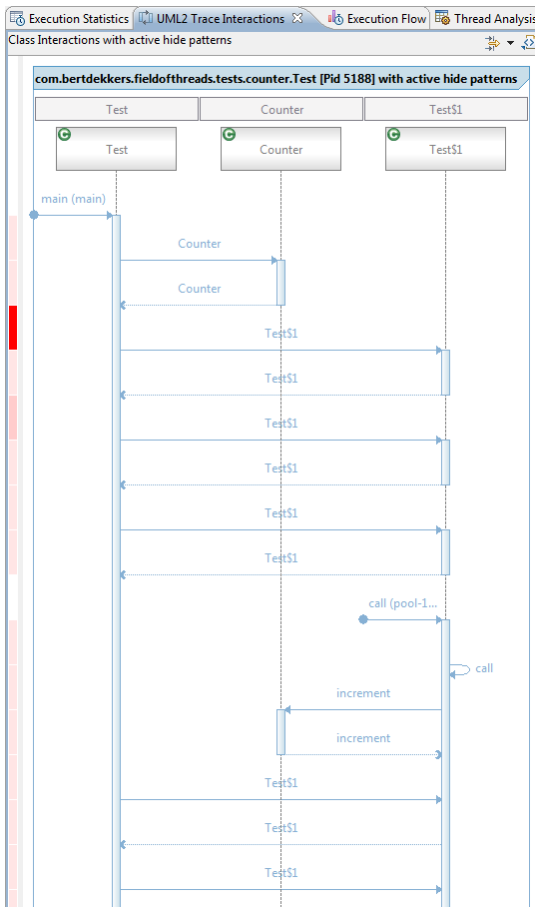
B. RELATED WORKS VISUALIZATIONS

B.3 TPTP Visualizations

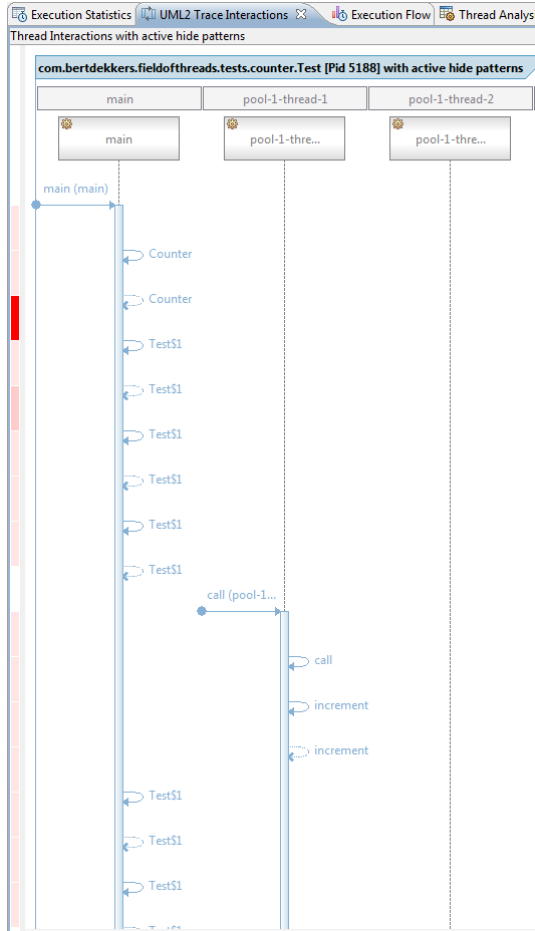
B.3.1 TPTP Call Tree



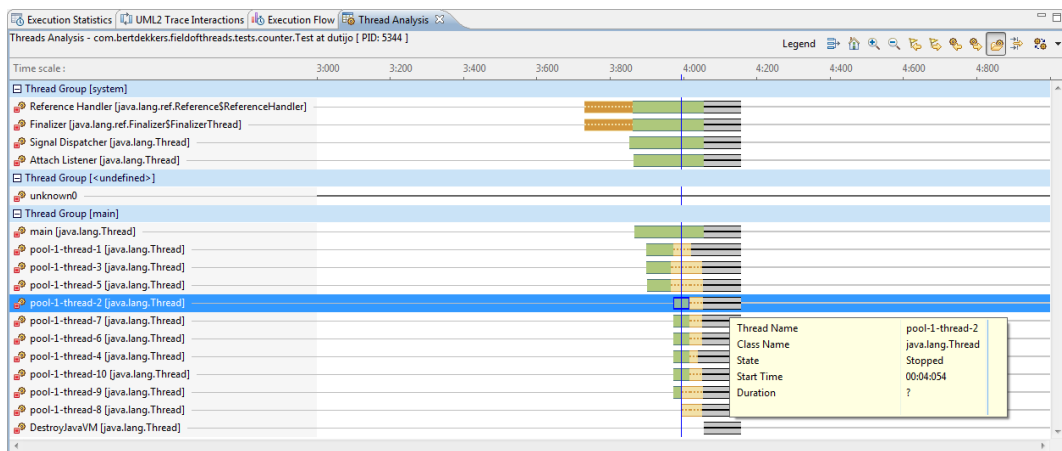
B.3.2 TPTP Class Interactions



B.3.3 TPTP Thread Interactions

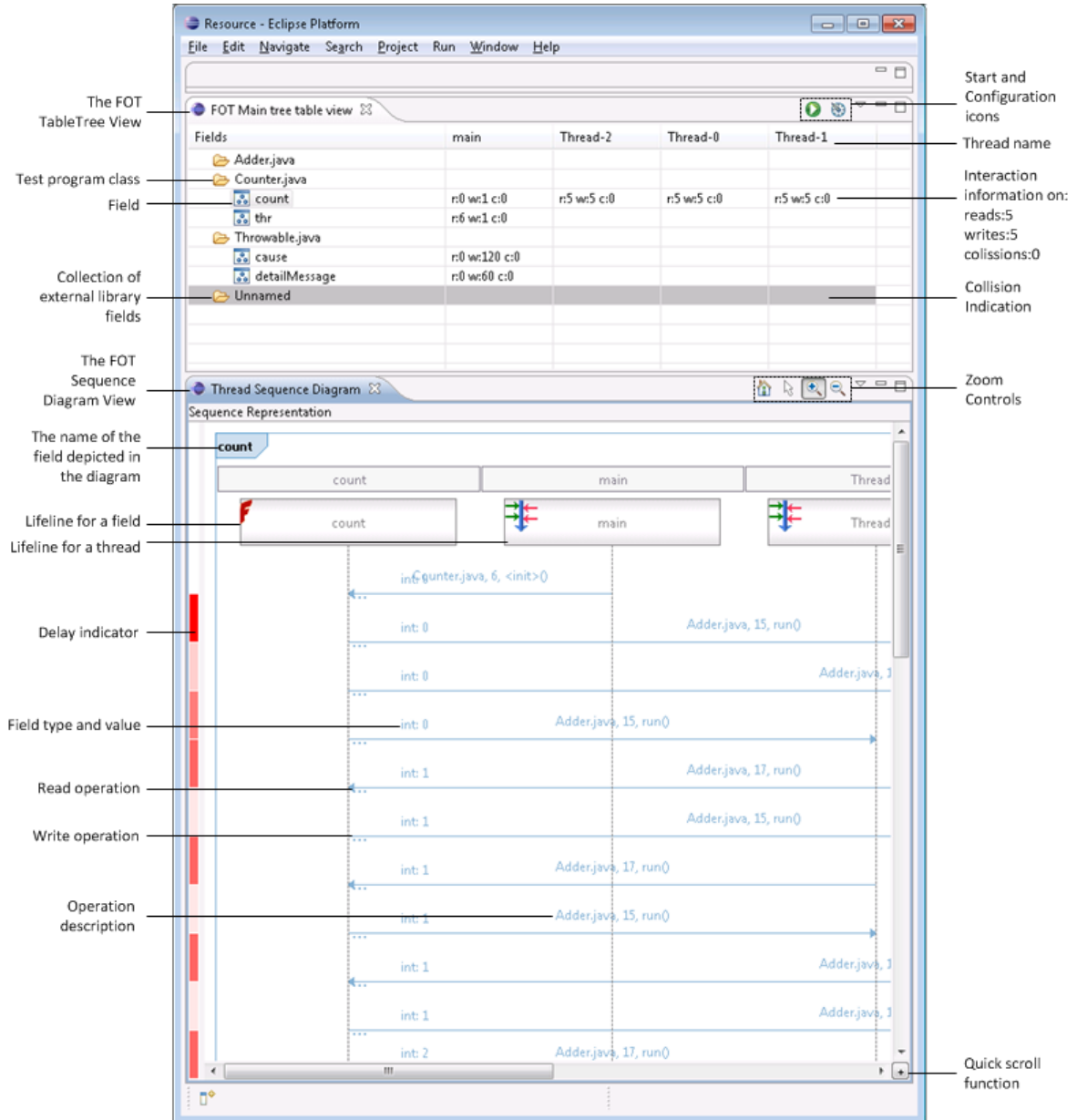


B.3.4 TPTP Thread Visualizer



B.4 FOT Appendices

B.4.1 Overview of FOT Visualizations and Controls



B.4.2 TMF Sequence Diagram Generation Tool code Example

The following code can be used to generate a sequence diagram in the TMF tool:

```
private void createFrame() {
    Frame testFrame = new Frame();
    testFrame.setName("Example Frame");

    // Create lifelines
    Lifeline lifeLine1 = new Lifeline();
    lifeLine1.setName("Object1");
    testFrame.addLifeLine(lifeLine1);
    Lifeline lifeLine2 = new Lifeline();
    lifeLine2.setName("Object2");
    testFrame.addLifeLine(lifeLine2);

    // Create Sync Message
    // Get new occurrence on lifelines
    //(extending the diagram one level down)
    lifeLine1.getNewEventOccurrence();
    lifeLine2.getNewEventOccurrence();

    // Get Sync message instance
    SyncMessage syn1 = new SyncMessage();
    syn1.setName("Sync Message 1");
    syn1.setStartLifeline(lifeLine1);
    syn1.setEndLifeline(lifeLine2);
    testFrame.addMessage(syn1);

    // Create Activations (Execution Occurrence)
    ExecutionOccurrence occ1 = new ExecutionOccurrence();
    occ1.setStartOccurrence(0);
    occ1.setEndOccurrence(syn1.getEventOccurrence());
    lifeLine1.addExecution(occ1);
    occ1.setName("Activation 1");

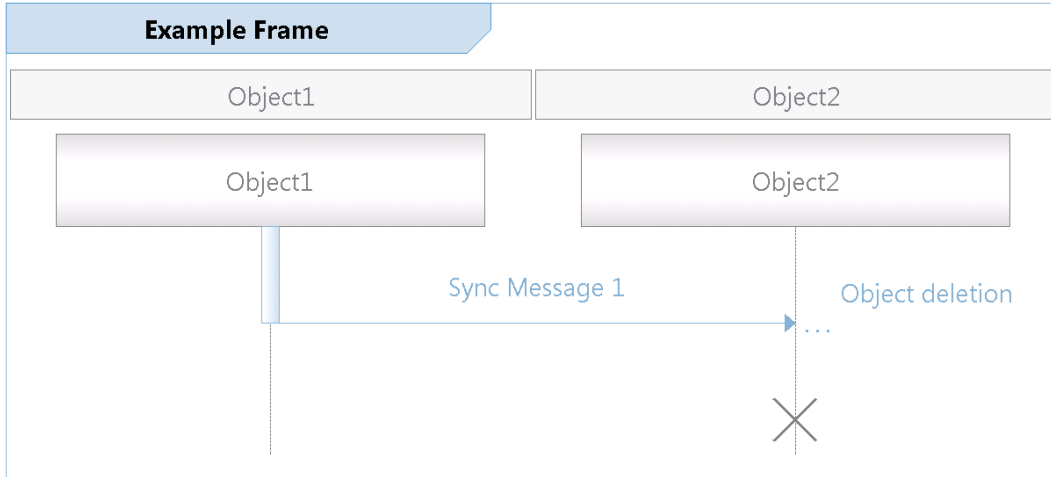
    // Create a note
    EllipsisMessage info = new EllipsisMessage();
    info.setName("Object deletion");
    info.setStartLifeline(lifeLine2);
    testFrame.addNode(info);

    // Create a Stop
    Stop stop = new Stop();
    stop.setLifeline(lifeLine2);
    stop.setEventOccurrence(lifeLine2.getNewEventOccurrence());
    lifeLine2.addNode(stop);

    fSdView.setFrame(testFrame);
}
```

B. RELATED WORKS VISUALIZATIONS

This code would generate the following diagram:



B.4.3 FOT Counter Concurrent Test Program

B.4.4 FOT Table View Unnamed example

Fields	pool-1-thread-1	pool-1-thread-3	pool-1-thread-2	main
Unnamed				
\$assertionsDisabled				r:5 w:0 c:0
\$assertionsDisabled	r:24 w:0 c:0	r:32 w:0 c:0	r:24 w:0 c:0	
\$assertionsDisabled				r:6 w:0 c:0
\$assertionsDisabled				r:12 w:0 c:0
\$assertionsDisabled	r:12 w:0 c:0	r:16 w:0 c:0	r:12 w:0 c:0	
\$assertionsDisabled				r:4 w:0 c:0
A				r:349 w:0 c:0
acc				r:18 w:0 c:0
accessOrder				r:4 w:0 c:0
after				r:2 w:2 c:0
allowArraySyntax	r:6 w:0 c:0			r:57 w:0 c:0
altSlash				r:24 w:0 c:0
append	r:6 w:0 c:0	r:8 w:0 c:0	r:6 w:0 c:0	
args				r:3 w:0 c:0
authority				r:44 w:17 c:0
autoFlush	r:12 w:0 c:12	r:16 w:0 c:16	r:12 w:0 c:8	
averageCharsPerByte				r:0 w:3 c:0
bb	r:48 w:0 c:0	r:64 w:0 c:0	r:48 w:0 c:0	
before				r:2 w:2 c:0
BIG_ENDIAN				r:1 w:0 c:0
bigEndian				r:0 w:2 c:0
blocker	r:0 w:3 c:0	r:0 w:3 c:0	r:0 w:3 c:0	r:0 w:3 c:0
blockerLock				r:0 w:15 c:0
booted	r:6 w:0 c:0			r:57 w:0 c:0
buf	r:24 w:0 c:0	r:32 w:0 c:0	r:24 w:0 c:0	
bugLevel				r:4 w:0 c:0
byteOrder				r:2 w:0 c:0
byteToCharTable				r:196 w:2 c:0
cache				r:8 w:0 c:0
cache1				r:8 w:2 c:0
cache2				r:2 w:2 c:0
capacity	r:60 w:30 c:0	r:80 w:40 c:0	r:60 w:30 c:0	r:10 w:10 c:0
CASE_INSENSITIVE_ORI				r:5 w:0 c:0
cb	r:12 w:0 c:0	r:16 w:0 c:0	r:12 w:0 c:0	
ch	r:6 w:0 c:0	r:8 w:0 c:0	r:6 w:0 c:0	
channel				r:3 w:3 c:0
charOut	r:6 w:0 c:6	r:8 w:0 c:8	r:6 w:0 c:4	
charset				r:0 w:3 c:0
classes				r:15 w:0 c:0
clock				r:1 w:0 c:0
combiner				r:5 w:0 c:0
complete	r:2 w:0 c:0			r:21 w:0 c:0
contents				r:15 w:0 c:0
context				r:13 w:0 c:0

Appendix C

Case Study

C.1 Test Program Case 1 Race Conditions

```
1 package com.bertdekkers.fieldofthreads.tests.race;
2
3 class Counter {
4
5     static final int THREADS = 3;
6     int count = 0;
7
8     Thread[] thr = new Thread[Counter.THREADS];
9
10    public static void main(String[] args){
11        Counter counter = new Counter();
12        counter.start();
13    }
14
15    public void start() {
16        for (int i = 0; i < Counter.THREADS; i++) {
17            thr[i] = new Adder(i, this);
18            thr[i].start();
19        }
20    }
21 }

```

```
1 package com.bertdekkers.fieldofthreads.tests.race;
2
3 class Adder extends Thread {
4     private int identity;
5     private Counter c;
6     int tempCount = 0;
7
8     Adder(int id, Counter c) {
9         identity = id;
10        this.c = c;
11    }
12
13    public void run() {
14        for(int i=0;i<5;i++){
15            tempCount = c.count;
16            System.out.println("Count = " + this.tempCount++ + " Added by: " + identity);
17            c.count = tempCount;
18        }
19    }
20 }

```

C.2 Test Program Case 2 Dining Philosophers

```

1  package com.bertdekkers.fieldofthreads.tests.dining.ic;
2
3  class Table {
4
5      Thread[] phil = new Thread[Table.NUMPHILS];
6      Ohashi[] ohashi = new Ohashi[Table.NUMPHILS];
7      int [] state = new int [NUMPHILS];
8      boolean[] untable = new boolean[NUMPHILS];
9
10     static final int NUMPHILS = 5;
11     static final int THINKING = 0;
12     static final int HUNGRY = 1;
13     static final int GOTRIGHT = 2;
14     static final int EATING = 3;
15     static final int GOTLEFT = 4;
16
17     public static void main(String[] args){
18         Table table = new Table();
19         table.start();
20     }
21
22     public void start() {
23         System.out.println("Starting Diner");
24         for (int i = 0; i < Table.NUMPHILS; ++i) {
25             ohashi[i] = new Ohashi(this, i);
26         }
27         for (int i = 0; i < Table.NUMPHILS; ++i) {
28             phil[i] = makePhilosopher(this, i, ohashi[(i - 1 + Table.NUMPHILS)% Table.NUMPHILS], ohashi[i]);
29             phil[i].start();
30         }
31         while (true) {
32             try {
33                 Thread.sleep(500);
34             } catch (InterruptedException e) {}
35             if (deadlocked() == true) {
36                 System.out.println("blocked");
37             }
38         }
39     }
40     public void stop() {
41         for (int i = 0; i < Table.NUMPHILS; ++i) {
42             phil[i].interrupt();
43         }
44     }
45     Thread makePhilosopher(Table d, int id, Ohashi left, Ohashi right) {
46         return new Philosopher(d, id, left, right);
47     }
48     synchronized void setPhil(int id, int s) throws java.lang.InterruptedException {
49         state[id] = s;
50         report();
51     }
52
53     synchronized void setFork(int id, boolean taken) {
54         untable[id] = !taken;
55     }
56
57     boolean deadlocked() {
58         int i = 0;
59         while (i < NUMPHILS && state[i] == GOTRIGHT){++i;}
60         for(Ohashi h :ohashi){if(h.isTaken() == true)h.tst=0;{}}
61         return i == NUMPHILS;
62     }
63
64     private void report(){
65         String out = "";
66         for(int s :state){
67             out+=" "+s;
68         }
69         System.out.println(out);
70     }
71 }

```

```
1 package com.bertdekkers.fieldofthreads.tests.dining.ic;
2
3 class Philosopher extends Thread {
4     private int identity;
5     private Table table;
6     private Ohashi left;
7     private Ohashi right;
8
9     Philosopher(Table table, int id, Ohashi l, Ohashi r) {
10        this.table = table;
11        identity = id;
12        left = l;
13        right = r;
14        System.out.println("Reservation for ID: "+id);
15    }
16
17    public void run() {
18        try {
19            while (true) {
20                // thinking
21                table.setPhil(identity, Table.THINKING);
22                sleep(25 * (int) (100 * Math.random()));
23                // hungry
24                table.setPhil(identity, Table.HUNGRY);
25                right.get();
26                // got right hashi
27                table.setPhil(identity, Table.GOTRIGHT);
28                sleep(250);
29                left.get();
30                // eating
31                table.setPhil(identity, Table.EATING);
32                sleep(20 * (int) (50 * Math.random()));
33                right.put();
34                left.put();
35            }
36        } catch (java.lang.InterruptedException e) {}
37    }
38 }
```

C. CASE STUDY

```
1 package com.bertdekkers.fieldofthreads.tests.dining.ic;
2
3 class Ohashi {
4
5     private boolean taken=false;
6     private Table table;
7     private int id;
8     int tst;
9
10    Ohashi(Table table, int id) {
11        this.table = table;
12        this.id = id;
13    }
14
15    synchronized void put() {
16        taken = false;
17        table.setFork(id, taken);
18        notify();
19    }
20
21    synchronized void get() throws java.lang.InterruptedException {
22        while (taken) {
23            wait();
24        }
25        taken = true;
26        table.setFork(id, taken);
27    }
28
29    public boolean isTaken(){
30        return taken;
31    }
32 }
```