Department of Computer Science and Engineering

Understanding and Debugging Concurrent Programs through Visualisation

Jan Lönnberg

Aalto University

DOCTORAL DISSERTATIONS

Aalto University publication series
DOCTORAL DISSERTATIONS 24/2012

Understanding and Debugging Concurrent Programs through Visualisation

Jan Lönnberg

Doctoral dissertation for the degree of Doctor of Science in Technology to be presented with due permission of the School of Science for public examination and debate in Auditorium T2 at the Aalto University School of Science (Espoo, Finland) on the 16th of March 2012 at 12 noon.

Aalto University
School of Science
Department of Computer Science and Engineering
Learning + Technology Group

Supervisor
Professor Lauri Malmi

Instructor
Associate Professor Mordechai Ben-Ari

Preliminary examiners
Professor Jürgen Börstler, Umeå University, Sweden
Professor Jeffrey Magee, Imperial College London, United Kingdom

Opponent
Associate Professor Arnold Pears, Uppsala University, Sweden

Aalto University publication series
DOCTORAL DISSERTATIONS 24/2012

ISBN 978-952-60-4529-0 (printed)
ISBN 978-952-60-4530-6 (pdf)
ISSN-L 1799-4934
ISSN 1799-4934 (printed)
ISSN 1799-4942 (pdf)

Unigrafia Oy
Helsinki 2012

Finland

The dissertation can be read at http://lib.tkk.fi/Diss/

Aalto University

Abstract

Aalto University, P.O. Box 11000, FI-00076 Aalto www.aalto.fi

Author
Jan Lönnberg

Name of the doctoral dissertation
Understanding and Debugging Concurrent Programs through Visualisation

Publisher Aalto University School of Science

Unit Department of Computer Science and Engineering

Series Aalto University publication series DOCTORAL DISSERTATIONS 24/2012

Field of research Software Systems

Manuscript submitted 15 November 2011 **Manuscript revised** 20 January 2012

Date of the defence 16 March 2012 **Language** English

☐ **Monograph** ☒ **Article dissertation (summary + original articles)**

Abstract

In this thesis, the development and evaluation of a visualisation system intended to support students in understanding and debugging concurrent programs is presented. The first phase of development consisted of examining how students understand and develop concurrent programs through phenomenographic research. The resulting outcome spaces included the students' understandings of tuple spaces, the purpose of a programming assignment and what developing, debugging and testing a concurrent program involves. The outcome spaces included categories ranging from simple understandings containing only what is necessary to complete the assignments to understandings that placed the program in a larger context beyond the assignment.

These outcome spaces were used in a classification of defects in students' concurrent programs. The defects found in the students' programs were classified by the underlying human error and by the type of program failure the defect causes. This analysis of defects was used to determine appropriate measures to support students in avoiding such defects. Many of the students' defects were related to misunderstanding the goals of the assignment, so they were rewritten to clarify the goals. To give the students a more concrete demonstration of the situations their programs had to deal with, test packages were provided to them.

Many of the students' defects were related to incorrect use of concurrency. To help students understand and correct these defects and learn from their mistakes, the visualisation system Atropos was developed. Atropos is intended to help students understand concurrent program behaviour in Java. Atropos supports backward debugging of concurrent Java programs through interactive exploration of a dynamic dependence graph. A solution for replay and dynamic dependence analysis of concurrent Java programs that may include data races was devised.

Atropos was evaluated through a mixed-methods analysis of the behaviour of pairs of students using Atropos to debug concurrent programs. The results include a description of the ways in which students successfully made use of Atropos and suggestions for how it could be improved to better support their debugging approaches. While students appear to understand the dependence graph representation and how to apply it in debugging, they need more support from Atropos for eliding the implementation of data structures in order to examine their use.

Keywords software visualisation, concurrent programming, dynamic dependence analysis, computer science education, phenomenography

ISBN (printed) 978-952-60-4529-0 **ISBN (pdf)** 978-952-60-4530-6

ISSN-L 1799-4934 **ISSN (printed)** 1799-4934 **ISSN (pdf)** 1799-4942

Location of publisher Espoo **Location of printing** Helsinki **Year** 2012

Pages 180 **The dissertation can be read at** http://lib.tkk.fi/Diss/

Aalto-universitetet

Sammandrag

Aalto-universitetet, PB 11000, 00076 Aalto www.aalto.fi

Författare
Jan Lönnberg

Doktorsavhandlingens titel
Programvisualisering för att hjälpa studenter förstå och avlusa jämlöpande program

Utgivare Aalto-universitetets högskola för teknikvetenskaper

Enhet Institutionen för datateknik

Seriens namn Aalto University publication series DOCTORAL DISSERTATIONS 24/2012

Forskningsområde Programsystem

Inlämningsdatum för manuskript 15.11.2011 **Datum för disputation** 16.03.2012

Datum för det korrigerade manuskriptet 20.01.2012 **Språk** Engelska

☐ **Monografi** ☒ **Sammanläggningsavhandling (sammandrag plus separata artiklar)**

Sammandrag

I denna avhandling presenteras utvecklingen och utvärderingen av ett visualiseringssystem som skapats för att hjälpa studenter förstå och avlusa jämlöpande program. I den första utvecklingsfasen undersöktes genom en fenomenografisk undersökning hur studenter uppfattar och utvecklar jämlöpande program. Utfallsrummen från denna undersökning handlade om hur studenter uppfattar tupelrymder, avsikten med ett övningsarbete i programmering och vad som ingår i utvecklandet, avlusandet och testandet av ett jämlöpande program. Utfallsrummen innehöll beskrivningskategorier från enkla uppfattningar som bara omfattar det som krävs för att utföra övningarna till uppfattningar som satt programmet i ett sammanhang som sträcker sig bortom övningen.

Utfallsrummen användes för att klassificera buggarna i studenternas jämlöpande program. Buggarna klassificerades enligt det bakomliggande mänskliga felet och enligt det felaktiga uppförandet i programmet buggen ger upphov till. Denna analys av buggarna användes för att finna lämpliga medel för att hjälpa studenter undvika att skapa dylika buggar. Många av buggarna kom från att studenterna missförstått övningsarbetenas mål, så de skrevs om för att förtydliga målen. Studenterna gavs testpaket som gav dem en mer konkret demonstration av övningsarbetenas mål.

Många av studenternas buggar var kopplade till felaktigheter i samverkan mellan exekveringstrådar. För att hjälpa studenter förstå och korrigera dessa buggar och lära sig från sina misstag utvecklades visualiseringssystemet Atropos. Atropos är tänkt att hjälpa studenter förstå hur jämlöpande program i Java uppför sig. Atropos stöder baklänges avlusning av jämlöpande Java-program genom interaktiv utforskning av en dynamisk beroendegraf. En lösning skapades för återuppspelning och dynamisk beroendeanalys av jämlöpande Java-program som kan innehålla datatävlingssituationer.

Atropos utvärderades genom en analys av hur par av studenter använde Atropos för att avlusa jämlöpande program. Analysen gjordes med kvalitativa och kvantitativa metoder. Utvärderingens resultat omfattar bland annat en beskrivning av hur studenterna utnyttjade Atropos och förslag för hur Atropos bättre kunde stöda deras sätt att avlusa. Fastän studenterna verkar förstå beroendegrafsrepresentationen och hur den kan utnyttjas i avlusning, borde Atropos ha bättre stöd för att dölja implementationen av datastrukturer då användningen av dem undersöks.

Nyckelord programvisualisering, jämlöpande programmering, dynamisk beroendeanalys, datadidaktik, fenomenografi

ISBN (tryckt) 978-952-60-4529-0 **ISBN (pdf)** 978-952-60-4530-6

ISSN-L 1799-4934 **ISSN (tryckt)** 1799-4934 **ISSN (pdf)** 1799-4942

Utgivningsort Esbo **Tryckort** Helsingfors **År** 2012

Sidantal 180 **Tillgänglig på nätet:** http://lib.tkk.fi/Diss/

Preface

Looking back on my postgraduate work, it seems that it can be divided into two phases: working backwards and forwards. My master's thesis was based entirely on visualisation technology with only a little discussion of the human part of the human-computer interaction involved. My initial plans for my doctoral thesis were essentially my master's thesis writ large: a complex visual debugging tool comprised of several visualisations of varying degrees of novelty. My supervisor, Lauri Malmi, and my instructor, Mordechai Ben-Ari, convinced me that the scientific and practical contribution of a visualisation tool would be greater if it were based on empirical studies of the needs of its users. This led me to analyse the defects in a set of concurrent programs written by students. While doing this analysis, I found that I lacked the necessary understanding of how students understood and approached concurrent programming. This encouraged me to take a further step back and perform a phenomenographic study on this topic. My licentiate's thesis instructor, Anders Berglund, was particularly helpful during this phase by introducing me to phenomenography and guiding me through the entire study.

Armed with a greater understanding of the human issues involved in the students' programming process, I could start moving forward and finish my defect analysis. Much of this work was based on the assessments made by the teaching assistants of the Concurrent Programming course: Teemu Kiviniemi, Kari Kähkönen, Sampo Niskanen, Pranav Sharma, Yang Lu, Ari Sundholm and Pasi Lahti. I could then move on to design and implement a visualisation of a concurrent program's execution as originally planned. Like my master's thesis work, the visualisation was based on the algorithm visualisation framework Matrix developed by my master's thesis instructor, Ari Korhonen, and other members of the Software Visualization Group. Finally, I did a study of how students use the visualisation in order to evaluate it.

While the process of writing this thesis has been longer and more complex than I expected, it is clear that the result is also much better in many ways because of this, and I am deeply grateful to all the aforementioned people for their guidance and assistance. Furthermore, without taking the scenic route through computing education, I would never have had a chance to become a part of the computing education community; a particularly tight-knit and friendly community. I would like to thank the many people I have had pleasant and productive discussions with, in particular the research groups to which I've belonged at TKK and Aalto: the aforementioned Software Visualization Group (SVG), Computer Science Education Research Group (COMPSER) and the Learning + Technology Group (LeTech) that was formed out of the previous two, as well as the groups that I have had the opportunity to visit: Uppsala Computing Education Research Group and the Computer Science Group of the Department of Science Teaching of Weizmann Institute of Science. Hecse, the Helsinki Doctoral Programme in Computer Science, also arranged for Ilkka Niemelä and Keijo Heljanko to provide additional advice. I'd also like to thank Judy Sheard, Simon and Margaret Hamilton for inviting me to join them in their literature analysis.

I am grateful to the pre-examiners, professors Jürgen Börstler and Jeffrey Magee, for taking the time to check my thesis and for their constructive feedback.

I thank TES, the Finnish Foundation for Technology Promotion, for providing funding for a year of my postgraduate studies. Aalto University and its predecessor TKK provided most of the rest of the funding. Hecse also provided some funds.

Finally, I would like to thank my parents for their continuing support, without which I doubt I could have got this far.

Espoo, February 17, 2012,

Jan Lönnberg

Contents

Preface **7**

Contents **9**

List of Publications **13**

Author's Contribution **15**

1. Introduction **17**

1.1 Consequences of Nondeterministic Execution 17

1.2 Goal and Approach 18

1.3 Evolution of the Research Process 20

2. Background **23**

2.1 Concurrent Programming 23

2.1.1 Java Memory Model 24

2.2 Software Defects 25

2.3 Debugging Strategies 27

2.3.1 Slicing 28

2.4 Debugging Techniques 28

2.4.1 Execution Replay 28

2.4.2 Visual Debugging 30

2.5 Evaluating Visualisations in an Educational Context 31

2.6 How Students and Professional Programmers Debug 32

2.7 How Students Determine Correctness in Concurrent Programming 33

3. Setting **35**

3.1 Programming Assignments 36

3.1.1 Trains 37

3.1.2 Reactor 38
3.1.3 Tuple Space 38
3.2 Weekly Exercises 39

4. Understanding Students Working with Concurrent Programs 41
4.1 Students' Understandings and Approaches 41
4.1.1 Phenomenography 42
4.1.2 Data Collection 43
4.1.3 Analysis 44
4.1.4 Students' Understandings of Tuple Spaces 45
4.1.5 Students' Understandings of the Goal of Program Development 46
4.1.6 Students' Approaches to Developing Programs 47
4.2 Students' Defects in Concurrent Programs 50
4.2.1 Data Collection 50
4.2.2 Analysis 51
4.2.3 Results 52
4.3 Conclusions 58

5. Visualisation System Design and Implementation 61
5.1 Collecting and Replaying Execution Traces with Dynamic Dependence Analysis 63
5.1.1 Instrumentation and Execution Trace Collection . . . 64
5.1.2 Replay and Dynamic Dependence Analysis 66
5.1.3 Technical Evaluation 67
5.1.4 Performance Loss Caused by Instrumentation 68
5.1.5 Effect on Failure Occurrence 68
5.2 Visualising Dynamic Dependence Graphs 69
5.2.1 Visual Representations 69
5.2.2 Navigating the DDG 70
5.3 Discussion 74

6. How Students Make Use of the Visualisation 77
6.1 Data Collection 78
6.2 Analysis 79
6.3 Results 81
6.3.1 What Students Try to Achieve with Atropos 82
6.3.2 Identifying Incorrect Behaviour 82
6.3.3 Debugging Process 84

6.3.4 Successful and Unsuccessful Use of Atropos 84

7. Validity 87

7.1 Descriptive Validity 87

7.1.1 Recordings 87

7.1.2 Code Analysis 88

7.1.3 Affecting the Students 88

7.2 Interpretive Validity 89

7.3 Theoretical Validity 90

7.4 Researcher Bias 90

7.5 Internal Validity and Credibility 91

7.6 External Validity, Transferability and Dependability 92

7.7 Construct validity 92

8. Conclusions 95

8.1 Understanding and Debugging Needs of the Students 95

8.2 Addressing the Students' Needs through Visualisation . . . 95

8.3 Evaluating the Visualisation 96

8.4 Implications for Teaching 97

8.5 Future Work 98

Bibliography 99

A. Assignment Used in Visualisation Evaluation 107

A.1 Tasks 107

A.2 `ConcurrentSelectionSort.java` 107

B. `Second.java` 109

Errata 111

Publications 113

List of Publications

This thesis consists of an overview and of the following publications which are referred to in the text by their Roman numerals.

I Jan Lönnberg and Anders Berglund. Students' understandings of concurrent programming. In *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007)*, pp 77–86, Koli, Finland, April 2008.

II Jan Lönnberg, Anders Berglund and Lauri Malmi. How students develop concurrent programs. In *Proceedings of the Eleventh Australasian Computing Education Conference (ACE2009)*, pp 129–138, Wellington, New Zealand, January 2009.

III Jan Lönnberg. Defects in Concurrent Programming Assignments. In *Proceedings of the Ninth Koli Calling International Conference on Computing Education Research (Koli Calling 2009)*, pp 11–20, Koli, Finland, November 2009.

IV Jan Lönnberg, Mordechai Ben-Ari and Lauri Malmi. Java Replay for Dependence-based Debugging. In *Proceedings of PADTAD IX — Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, pp 15–25, Toronto, Ontario, Canada, July 2011.

V Jan Lönnberg, Mordechai Ben-Ari and Lauri Malmi. Visualising Concurrent Programs with Dynamic Dependence Graphs. In *Proceedings of 6th IEEE International Workshop on Visualizing Software for Under-*

standing and Analysis (VISSOFT 2011), 4 pp, Williamsburg, Virginia, USA, September 2011.

VI Jan Lönnberg, Lauri Malmi and Mordechai Ben-Ari. Evaluating a Visualisation of the Execution of a Concurrent Program. *Proceedings of the Eleventh Koli Calling International Conference on Computing Education Research*, pp 39–48, Koli, Finland, November 2011.

Author's Contribution

Publication I: "Students' understandings of concurrent programming"

Lönnberg performed the data collection and analysis himself and wrote most of the paper, with Berglund providing feedback and suggestions throughout the process.

Publication II: "How students develop concurrent programs"

Lönnberg performed the data collection and analysis himself and wrote most of the paper, with Berglund and Malmi providing feedback and suggestions throughout the process.

Publication III: "Defects in Concurrent Programming Assignments"

Lönnberg did the analysis and wrote the paper himself. The analysis used assessments made by teaching assistants of the Concurrent Programming course of students' programs as data; the author was the assistant only one year out of three.

Publication IV: "Java Replay for Dependence-based Debugging"

Lönnberg designed and implemented the system and performed the evaluation himself and wrote the paper, with Ben-Ari and Malmi providing feedback and suggestions throughout the process.

Publication V: "Visualising Concurrent Programs with Dynamic Dependence Graphs"

Lönnberg designed and implemented the system and performed the evaluation himself and wrote the paper, with Ben-Ari and Malmi providing feedback and suggestions throughout the process.

Publication VI: "Evaluating a Visualisation of the Execution of a Concurrent Program"

Lönnberg performed the evaluation himself and wrote the paper, with Malmi reanalysing some of the data and both Malmi and Ben-Ari providing feedback and suggestions throughout the process.

1. Introduction

There are many ways in which writing concurrent programs can be challenging. Dividing the work of a program between different processors or distributed nodes adds complexity to a program. Furthermore, the separate processes or threads of execution[1] must be co-ordinated so that they communicate correctly with each other when necessary, often through shared resources such as memory, yet do not interfere with each other.

1.1 Consequences of Nondeterministic Execution

Co-ordinating processes correctly is particularly difficult, since most concurrent programs and systems involve *nondeterminism*; they are not guaranteed to behave the same way every time they are executed, even if the same input is given to them. This is typically due to unpredictable differences in timing caused by a wide variety of sources, such as variation in time needed to process different data elements, the behaviour of schedulers in operating systems, and, especially in distributed systems, delays in communication.

Nondeterminism has strong implications for the development of concurrent programs. Testing certain inputs (test cases) once is not enough to reliably detect a defect that manifests itself nondeterministically, since only some of the possible interleavings of threads or processes will result in a failure. In order to effectively test for defects that manifest themselves nondeterministically, steps must be taken to ensure coverage of different interleavings in addition to the usual coverage criteria. This may be done using stress testing or by modifying thread scheduling behaviour to

[1]In this thesis, I follow common usage in using the term 'process' for both processes and threads in situations where the distinction is irrelevant, such as when discussing many concurrent algorithms and concepts on a general level. In specific cases where the distinction clearly exists, such as when discussing a program implemented using Java [27], I use the term 'thread'.

increase the chances of concurrency defects being exposed [22, 26, 68, 74].

Deductive proofs (done by hand or with automated theorem proving tools such as the SPARK tools [13]) and *model checking* (using a model checker such as Spin [30] or JPF [77]) are two alternative approaches to checking the correctness of a program that can be used to check that all interleavings lead to the desired behaviour. There are also static analysis tools, such as Jlint [4] and FindBugs [31], that look for specific patterns in code that are commonly associated with defects. None of these approaches is clearly superior to others; they detect different types of defects and make different trade-offs between generating false positives and false negatives [65].

Studying the execution of a program for debugging purposes is also affected. If a program behaves differently when re-executed, debugging techniques that rely on repeatedly re-executing a program to examine its behaviour become difficult to apply. This not only makes debugging concurrent programs harder than debugging sequential programs, it also makes it hard to learn concurrency by examining the behaviour of concurrent programs.

It is obvious that students must learn how concurrency mechanisms are to be used in order to effectively use them in their programs. However, as noted above, the most important way in which concurrent programming differs from sequential programming is in its nondeterministic behaviour. In order to be able to develop correct concurrent programs, a programmer must understand the implications of concurrency described in this section. Hence, helping students understand how to deal with concurrency is an important goal of teachers of concurrent programming.

1.2 Goal and Approach

The long-range goal of the work described in this thesis is to help programmers produce correct concurrent programs. The approach used is to develop methods and tools to help programmers understand what happens during the execution of a concurrent program. This serves two purposes: supporting the debugging of programs, allowing programmers to find the bugs in their programs and correct them, and helping programmers understand the behaviour of concurrent programs.

There are several reasons why students of concurrent programming are an obvious target audience for this type of research. Most obviously, they are still learning about concurrent programming and are likely to need help in understanding what happens when a concurrent program is executed. Introducing them to a new way of debugging is likely to have more of a (hopefully positive) effect on them than on seasoned programmers, since students do not yet have ingrained ways of working and have more of their career ahead of them to apply the new techniques in. Furthermore, they have more difficulties due to incomplete knowledge and lack of experience, so they are in greater need of help.

Price et al. [63] note that while *software visualisation* (SV) "has tremendous potential to aid in the understanding of concurrent programs", few SV systems are actually in production use, especially by professional programmers. They also note that when a SV system is designed, the content to be shown must be selected according to the goals of the system, which in turn are based on the requirements of the users.

Similarly, Hundhausen et al. [32] note that a lot of visualisation research involves exploring new visualisation techniques based on what the researchers feel would be useful or filling a niche in a taxonomy rather than studies of the requirements of programmers.

Based on this reasoning, the work described here consists of three parts:

1. Gaining an understanding of how students understand (and fail to understand) concurrent programming and what they need help with in debugging and understanding concurrent programs. This is described in Chapter 4, Publication I, Publication II and Publication III. This answers the question: What needs do students have with regard to understanding and debugging concurrent programs?

2. Designing and implementing a visualisation tool to support the students in understanding concurrent programs, using empirical studies to determine the students' needs. This is described in Chapter 5, Publication V and Publication IV. The question here is: How can the needs from the previous part best be addressed through visualisation?

3. Evaluating the visualisation tool and finding out what aspects of it are helpful to students by examining how students make use of the visualisation tool. This is described in Chapter 6 and Publication VI. The question here is: In what ways did the visualisation tool in the previous part assist the students?

Before describing my own work, I will summarise the previous work in the relevant fields on which this work builds (Chapter 2) and the setting in which all this work has been done (Chapter 3). I finish by discussing validity issues (Chapter 7 and presenting the conclusions of the thesis (Chapter 8).

1.3 Evolution of the Research Process

My initial goal and plan was to explore the possibilities of visual debugging and testing and create a new debugging tool that integrated all of this, essentially as a continuation of MVT, the tool I developed as part of my master's thesis [49]. In this original plan, the focus would have been on increasing the abstraction level of the data visualisation, visualising execution traces and controlling the execution of the program and manipulating the data through the visualisation tool. The target group was professional programmers, although a controlled experiment using advanced students as an approximation of professionals was planned to provide statistically significant proof of the system's ability to improve debugging.

After extensive discussions with other researchers, it became clear that this extended MVT would have been, much like MVT itself, a solution in search of a problem; an example of *system roulette*. Hence, two major changes were made to the plan: the decision was made to focus on concurrency-related issues in Java and, to avoid system roulette, a study of the defects in students' programs was added. At this point, the planned system would have visualised both execution traces and states in the execution.

Having attempted, as described in Sections 4.1 and 4.2, to analyse the defects in students' programs, I was encouraged to perform a qualitative study to examine how students actually understand and approach concurrent programming so I could use that as a basis for the rest of my work. This work is described in Section 4.1. The revised defects analysis is presented in Section 4.2.

I could finally return to what I had originally planned to focus on, visualisation development. However, at this point, the focus of the development had changed to reflect changes in my values; mostly a shift from a software developer's perspective to an academic one. Specifically, I now saw students as the intended users, rather than professional programmers, and the focus of the research was no longer to build new tools but to examine how visualisation can support students. At this point, the research goals became those described in Section 1.2.

Once the visualisation tool was functional enough to evaluate, I performed a small-scale usability test with a few students. The results were encouraging and suggested a few small changes that would improve usability. This allowed me to proceed with the evaluation described in Chapter 6. The plan for the evaluation involved a comparison between users and non-users of Atropos as well as a broader qualitative analysis. In addition to the operation foci that were examined, the students' activities were also analysed for interaction and transactive discourse. The latter was left out when preliminary analysis suggested that our students had very little transactive discourse. The former was left out since its results overlapped with the operation foci, but seemed less relevant to the research questions.

2. Background

In this chapter, I present background information relevant to understanding this thesis. Section 2.1 is a brief summary of the aspects of concurrent programming relevant to the research described here. Section 2.2 defines terminology for discussing defects in software and presents studies of defects in software.

Section 2.3 is an overview of debugging strategies. Section 2.4 describes several debugging techniques with a focus on visualisation. The remaining sections present empirical studies of the use of visualisation (2.5), debugging (2.6) and how students determine the correctness of a program (2.7).

2.1 Concurrent Programming

This section briefly summarises the aspects of concurrent programming students are expected to understand and make use of when writing concurrent programs and those necessary to understand the explanations in this thesis.

Concurrent programs may be run on anything from a single processor to a distributed system connected only by the Internet. Most of the concurrent programs discussed in this thesis run on one or more processors in shared memory, typically within a single Java Virtual Machine (JVM). Some run in a distributed context.

In order for concurrent programs to be useful, the processes in the programs need to be able to communicate with each other. Hence, mechanisms for interprocess communication (IPC) are necessary. Many of these mechanisms focus on ensuring *synchronisation* of processes; the avoidance of incorrect interleavings of processes. The simplest form is the *lock* or *mutex*, which allows programmers to designate parts of a program as *critical* *sections*. Execution of critical sections protected by a lock is *mutually exclusive*; only one critical section per lock is active at a time. *Monitors* extend locks by including *condition variables*. A condition variable provides a queue in which processes can *wait* until another process *notifies* the condition variable, which allows one of the processes to proceed. In some systems, a process can notify all the processes waiting on a condition variable at once.

Semaphores contain a non-negative integer value and two operations that make use of it: `P` ('prolaag' or 'try-to-decrease'; also known as 'wait' or 'down'), and `V` ('verhoog' or 'increase'; also known as 'signal' or 'up') [3, 5, 18, 53]. `V` increments the value of the semaphore by one. `P` waits until the value of the semaphore is positive and then decrements it by one.

Synchronisation constructs are often used in conjunction with *shared memory*; memory that can be read and written by several processes. Concurrency in Java is based on shared memory and monitors [27, §17].

An alternative approach is *message passing*, in which *messages* are sent between processes. These messages both contain data and can be used as signals for synchronisation. *Tuple spaces* [24] can be seen as an extension of message passing in which messages (*tuples*) can be stored in a shared storage space and accessed by specifying a *pattern* describing the tuples to fetch. Traditionally, a tuple space can be accessed through (at least) three operations: `in()`, `out()` and `read()`. `out()` takes a tuple as an argument and inserts it in the tuple space. `in()` takes as its argument a pattern consisting of a tag and zero or more data values or formal parameters. As soon as a matching tuple is found, `in()` fills all the formal parameters with the corresponding values from the matching tuple, removes the tuple from the space and returns. `read()` behaves like `in()`, but does not remove the tuple from the space.

More detailed explanations of these concepts can be found in almost any textbook on concurrent programming (e.g. [3, 5, 53]).

2.1.1 Java Memory Model

While Java uses shared memory, using it to transfer data between threads is not a straightforward matter of writing in one thread and reading in another. This is something that we expect our students to be able to take into account when writing concurrent Java programs. Also, tools that analyse Java programs must be designed to take the Java memory model into account.

The memory model of Java 1.5 [27, §17.4] describes the conditions that need to be met for inter-thread actions to be guaranteed to be *visible* to each other; i.e. for operations to see the changes to the program state made by each other. All synchronisation actions (acquiring and releasing locks, starting and joining threads and reading or writing `volatile` variables) are totally ordered in each execution. However, many actions that can be affected by other threads (such as reading non-`volatile` variables) do not have a total order. Also, in many cases (e.g. two threads acquiring two different locks at the same time), the order of two synchronisation actions cannot be distinguished, in which case the JVM may, in practice, allow these actions to occur simultaneously. The behaviour of shared memory is defined using the *happens-before* relationship, which is a partial order between inter-thread actions that is consistent with the synchronisation order. If an action happens-before another, the effects of the first are visible to the second. Naturally, operations in a thread happen-before each other in the execution order of the thread. Synchronisation between threads induces happens-before relationships between threads; most importantly, releasing a lock in one thread happens-before it is next acquired. Similarly, starting a thread happens-before its first operation. As long as the results are consistent with this memory model, the actual JVM implementation may reorder and parallelise operations arbitrarily.

If a read and a write or two writes are made to the same variable and neither happens-before the other, a *data race* occurs. In this situation, the result of reading this variable is not defined.

2.2 Software Defects

A programmer may make an *error* [11, 33, 59] (*mistake* [33, 59], *breakdown* [42], *failure* [59])—“a mistake, misconception, or misunderstanding on the part of a software developer” [11]—while writing a program. This may cause a *defect* [11, 59] (*error* [33, 42], *bug* [11, 33, 59, 80], *fault* [11, 33, 59])—a discrepancy between the actual program and a correct program[1]—to be introduced in the program. When the code containing the defect is executed, a *fault* [42] or *failure* [33]—a discrepancy between the actual execution and the behaviour of a correct program—may occur. This may then manifest itself as a *symptom* [59] (*failure* [11, 33, 42, 59],

[1]For the sake of simplicity, it is assumed in this thesis that an unambiguous definition of what constitutes a correct program, such as a specification, exists.

error [33, 80])—incorrect behaviour that someone or something notices, such as incorrect output or an exception.

It is clear that the terminology is somewhat inconsistent; the terminology not in parentheses in the previous paragraph is the one used in this thesis.[2]

In this thesis, the focus is on errors that lead to runtime failures. One can examine the resulting programs in terms of their errors, defects, failures and symptoms. If a program contains a defect that prevents it from being compiled or executed, there can be no failure and symptom, except, arguably, the diagnostic output from the compiler or execution environment. Finding and correcting this type of problem involves different approaches and tools, such as improving compiler error messages (see e.g. [19]).

Knuth [40] made use of introspection to classify his own errors according to the aspect of his programming process he failed in and found the following categories of error:

- Algorithm Awry
- Blunder/Botch
- Data structure Debacle (violation of data structure invariants)
- Forgotten Function (missing code)
- Language Liability
- Mismatch between Modules (conflicting assumptions in modules)
- Reinforcement of Robustness
- Surprising Scenario (unexpected interactions between program parts)
- Trivial Typo

Eisenstadt [21] analysed anecdotal descriptions of bugs that professional programmers found hard to find. The most common reasons for this were related to the symptoms:

- Cause and symptom are separated in space or time;
- The incorrect behaviour does not consistently manifest itself;
- The programmer gets stuck by misinterpreting what he sees.

Several studies have been made of students' errors in programming assignments, such as those by Grandell et al. [28], Ko and Myers [42], Spohrer and Soloway [71]. These studies are intended to help develop an understanding of why students make the mistakes they do, which can then be

[2]See the Errata for cases where I do not follow the above terminology.

used to help develop teaching to help students develop the knowledge or skills that the errors demonstrate are problematic.

The goal/plan analysis of Spohrer and Soloway [71] and Spohrer et al. [72] uses the part of the program design that is incorrect as a basis for analysis; a program has a goal. For every goal, the programmer chooses a plan to solve it, which in turn involves subgoals that must be solved, until the plans are small enough to be translated directly into code.

Grandell et al. [28], Spohrer and Soloway [71], Spohrer et al. [72] used the students' code as data; either the final versions submitted by the students [28] or every syntactically correct version compiled by the students [71]. This code was then (mostly manually) analysed for defects. Grandell et al. found that most students' logic errors could be attributed to accepting erroneous input and incorrect or missing algorithms. Spohrer et al. [72] found that novice programmers who tried to address several goals with the same code often did so incorrectly. Spohrer and Soloway [71] found that few of the bugs produced by novices can be explained as misunderstandings of programming language constructs.

In contrast, Ko and Myers [42] set up experiments with students performing a programming task, who were videotaped while encouraged to think aloud. This allowed Ko and Myers to determine the errors the students made and the cognitive breakdowns that led to these errors, such as not applying rules to construct or modify Boolean expressions correctly.

An overview of studies of what bugs occur and why has been done by McCauley et al. [57].

2.3 Debugging Strategies

In this section, I summarise the different approaches and strategies that can be used to debug programs and justify the choice of strategy underlying the visualisation design described in Chapter 5.

There are two main approaches to finding ways a program can fail: *dynamic* (in which programs are executed and the results analysed) and *static* (in which the analysis is done on program code without executing it). This thesis focuses on dynamic methods, as most work with a focus on concurrency belongs to this category and the focus of static methods is typically on finding patterns in the code that correspond to common mistakes [65].

Metzger [59] describes many different strategies for debugging a program. Most of them are based either on dividing program code into parts, searching through these parts and ruling them out as the site of the bug or on generating hypotheses and disproving them. These strategies do not specify what information is collected, making it hard to develop software tools to help programmers to apply them. However, one strategy, the *slicing* strategy, is different in many ways, and will be discussed in detail in the following subsection.

2.3.1 Slicing

A *(static) slice* of a program with respect to a variable at a specific point in the program is the part of the program that could have affected the value of the variable at this point. A *dynamic slice* similarly is the part of the execution of a program that was involved in determining the value of the variable at a particular point in the execution. By repeatedly using slices, programmers can trace the propagation of incorrect behaviour back through the program or its execution from a symptom to the fault and thus identify the defective code [1, 59, 79, 80].

In order to calculate a dynamic slice, one must first find the corresponding *dynamic dependence graph* (*DDG*). A DDG is a directed graph whose vertices are *operations* (executions of statements) and whose edges are the data and control dependencies between these operations. A dynamic slice of a program is the lines of code that appear as operations in a DDG.

Effectively, a DDG explains why each operation did what it did by explaining why it was executed (how the program branched to this code) and where it got the data it operated on.

2.4 Debugging Techniques

I present in this section an overview of debugging techniques for concurrent programs, with the focus on those that are applied in Chapter 5: *execution replay* and *visual debugging*.

2.4.1 Execution Replay

Since the first debugger, *FLIT* [73], debuggers have been based on breakpoints and single-stepping. The user of the debugger often finds himself having to re-execute a program to examine something that happened earlier in the program's execution. A similar issue is even more common among programmers who debug by adding debug code such as `print` statements to examine program execution. While this is not a problem if the input can easily be replicated and the program is deterministic, the nondeterminism of concurrent programs makes this approach difficult to apply to them.

Since a concurrent execution can be hard to reproduce, collecting a trace of the execution of a program is particularly useful for debugging concurrent programs. The trace is typically used to replay the execution of the program and examine it in another debugging tool. The quality of the replay is determined by its *accuracy* (how closely the replay matches the original) and *precision* (how much the execution is changed by collecting information on it). Accuracy can be ensured by collecting sufficient information on the nondeterministic aspects of the execution to reconstruct the execution completely. Precision is less clear-cut and can be affected by e.g. unnecessary instrumentation [14].

Execution traces can be generated by modifying the execution environment to collect the information needed to reconstruct an execution. For example, *DejaVu* [14] and *jreplay* [67] use modified JVMs to keep track of thread switches and other nondeterministic behaviour. DejaVu can then replay the trace for examination in, for example, a traditional debugger. Precision and compatibility may suffer if one has to use a different JVM for debugging. Also, these systems are limited to single-processor execution, limiting the precision of debugging on multiprocessor systems, and do not collect the information needed for accurate replay of data races.

Instead of modifying the JVM, one can use *instrumentation*: the addition of code to collect information, as done by e.g. *JaRec* [25]. JaRec also uses instrumentation to perform replay by enforcing the original order on synchronisation operations. JaRec cannot handle data races, either.

RetroVue [12] and *ODB (Omniscient Debugger)* [46] use instrumentation to generate execution traces. Instead of replaying the execution, these tools enable the programmer to examine the execution history through a graphical debugger. Both debuggers can also show execution traces as a list of operations and enable the user to examine all the states of the program in the same way a traditional debugger enables the examination of the current program state. The instrumentation used by ODB causes precision problems by adding extensive synchronisation to the program that is being examined. *MVT* [52] is similar, but lacks the list view of RetroVue and ODB.

In short, DejaVu and jreplay use modified JVMs to trace uniprocessor execution, while JaRec, RetroVue, ODB and MVT use instrumentation to trace execution on any JVM. None of these can trace data races accurately.

2.4.2 Visual Debugging

Both traditional debuggers and visual debuggers such as *DDD* [82] typically show only the current state of the program. RetroVue [12] provides a tree view of all executed operations and, as noted in Subsection 2.4.1, the ability to examine all the states of the program. It also provides a thread display that shows the times each thread executed and locking interactions between them.

Query-based debugging (*QBD*) is based on the idea of allowing a programmer to perform queries on the data in the program that is being debugged. Lencevicius et al. [45] have implemented a QBD tool for Java based on instrumenting the program to be debugged with debugger code wherever a change relevant to the query is made.

The *Whyline* [41] uses a DDG-based visualisation (together with the other elements of the Alice IDE, and, in a later version, Java [43]) to answer queries such as “Why was this statement (not) executed?” or “Why does this variable have this value?”. The answer is (part of) a dynamic dependence graph (DDG). In many cases, this DDG enables the programmer to find the reason for incorrect behaviour of a program very quickly by tracing the cause-effect chain from the bug to the symptom backwards along the DDG. The DDG can further be used to quickly navigate to sections of the program and its execution relevant to the bug. Evaluations suggest that the Whyline substantially speeds up the debugging process for novice programmers.

A few debuggers and program visualisation systems have been designed with concurrency in mind. Most of them (e.g. JAVAVIS [62] and JAN [48]) use sequence diagrams or message sequence charts to display method calls and object diagrams to show program state; JaVis [58] uses collaboration diagrams to show interactions between objects. These diagrams have a level of detail suitable for debugging, but become cumbersome for complex executions. Kraemer [44] describes many visualisations for specific aspects of concurrent programs such as call graphs and time-process diagrams for message traffic; these visualisations are primarily designed to give an overview of thread interactions and thus have too little detail to readily identify the code or data involved in individual interactions.

Name	Data shown	Data collection	Visualisation type
DDD (Java)	Program state	JDI	State view, object graph
RetroVue	Program execution	Instrumentation	Execution history tree, state view
Lencevicius	Program execution	Instrumentation	Query responses
Whyline (Java)	Program execution	Instrumentation	DDG
JAVAVIS	Program execution	JDI	Sequence diagram, object diagram
JAN	Program execution	Instrumentation	Sequence diagram, object diagram
JaVis	Program execution	JDI	Sequence diagram, collaboration diagram
Bogor	Counterexample	Model checker	Execution history tree, object graph
Spin	Counterexample	Model checker	MSC

Table 2.1. Comparison of visual debuggers

DDD, JAVAVIS and JAN use the Java Debug Interface (JDI; part of the Java Platform Debugger Architecture [75]) built into Sun's JVM to collect information on program execution. The others use instrumentation of Java bytecode.

Finding a defect from a model checker counterexample is similar to debugging in that the program is known to behave incorrectly and the programmer seeks to find the underlying defect. Assuming the program one wishes to debug can be effectively analysed by a model checker and a similar visualisation can be used to examine the model checker's output, a model checker can also be used for visual debugging. *Bogor* [64], for example, enables the user to examine the counterexample using visualisations similar to those of DDD [82] and RetroVue). *Spin* [30] can produce message sequence charts that show interactions between processes.

A full summary of all visual debuggers is beyond the scope of this thesis, see [49] for more. This subsection is summarised in Table 2.1.

2.5 Evaluating Visualisations in an Educational Context

Many empirical evaluations have been made of software visualisation [32, 69], and it is beyond the scope of this thesis to present them all. The focus is on the methods used in the evaluation described in Chapter 6. Hundhausen et al. [32] describe in their meta-study many different evaluations of algorithm visualisations based on empirical techniques. Many of these, such as [60], are controlled experiments that attempt to determine whether changing e.g. the learning or debugging medium affects a measure of success such as post-test accuracy or debugging time by comparing different groups of students. However, in order to help identify how the visualisation helps students and support future development, it would be useful to examine how the students make use of the visualisation in more detail.

Kiesmüller [39] and Yehezkel et al. [81] have examined how the use of visualisation tools in an educational context affects students' activities when defining and testing a program. In both cases, the activities of the students were recorded and the focus of the students' operations (what activities the students performed) and—in the latter study—the students' conversation (what the students talked about) was determined. Yehezkel et al. found that students working without the EasyCPU visualisation primarily used the data input and instant run operations with a strategy of trial and error, while the students using EasyCPU tended to run the program step by step and investigated the program's execution more. Kiesmüller identified different problem-solving strategies, such as bottom-up, trial and error and hill climbing.

Isohanni and Knobelsdorf [34] examined how students make use of the program visualisation tool VIP by interviewing them and observing and video recording them completing a short programming assignment. They found that when instructed to use VIP, the students ran their program in VIP. However, not all students used VIP to examine the program execution, and only one used VIP as intended (single stepping).

2.6 How Students and Professional Programmers Debug

Several studies have been made of students' debugging strategies [69]. As in many subfields of computer science education, most of the work on how students debug focuses on novice programmers [57].

Eisenstadt [21] found that what the programmers did to track down the difficult bugs they reported was to examine what happened when the program was executed through traditional debugging techniques such as single-stepping, adding `print` statements, adding conditional breakpoints to the program and inspecting the data when the breakpoint is triggered and comparing dumps of the program's state.

Both von Mayrhauser and Vans [78] and Eisenstadt [21] have shown that programmers spend a lot of time tracing the data and control flow of programs in order to find causes for bugs. They also show that programmers often require information on the causes of an event and connections between parts of a program or its execution when looking for hard-to-find defects. Eisenstadt in particular emphasises that complex cause-effect relations that can be computed (e.g. data flow links) should be computed by the debugger rather than forcing the programmer to work them out. He also points out that the information needed is often at a higher level of abstraction and granularity than the values of individual variables. This suggests that tool support for tracing data and control flow could be very useful in debugging, which is one of the justifications for the design presented in Chapter 5.

Fitzgerald et al. [23] used interviews including both a programming exercise and a debugging exercise to investigate the debugging skills and approaches of novice programmers. Strategies used by the students included mental tracing (with and without `print` statements), hand tracing and tracing using the debugger. The students exhibited both *forward* (tracing) and *backward* (causal) reasoning.

Ahmadzadeh et al. [2] examined programs compiled by students working on a similar debugging task and a programming task. They compared students' work in the roles of *programmer* and *debugger*. They found that weak debuggers had difficulties applying their knowledge of programming and debugging unless they were working with programs with a familiar structure. While most good programmers were better at debugging than weak programmers, the good programmers who were weak debuggers did not appear to understand the program to debug well enough to find or fix the bugs.

Murphy et al. [61] studied the debugging strategies of students who had completed 15–20 weeks of Java instruction using a debugging exercise and a semi-structured interview. They found that students made extensive use of testing, tracing and print statements. However, they often did not use these techniques rigorously or efficiently. They also sometimes used counterproductive approaches such as rewriting code they did not understand.

2.7 How Students Determine Correctness in Concurrent Programming

When determining whether a program is correct or not, the definition of 'correct' being used obviously has a strong effect on the approach used.

Indeed, as will be demonstrated later in this thesis, many defects in students' programs can be explained as consequences of students having different understandings of correctness. In such cases, explaining to the student how a program behaves incorrectly may be of secondary importance to explaining why the behaviour is incorrect. Ben-David Kolikant [7] writes that students define a "correct program" as a program that exhibits "reasonable I/O for many legal inputs" and that roughly a third of the students did not even run their programs to check whether they worked; if it compiled, they were satisfied that it was correct.

Ben-David Kolikant [6] describes learning concurrent programming as entering a community of computer science practitioners. Students initially approach the concurrent programming assignment from a user's perspective, since that is what they are familiar with. Only one of the two students in the study was able to switch to a programmer's perspective and reason correctly about synchronisation goals and interleavings.

3. Setting

The work described in this thesis revolves around the Concurrent Programming course (T-106.5600, before 2006 T-106.420) at Aalto University (before 2010 Helsinki University of Technology).[1] The goal of this course is to teach students:

- The principles of concurrent programming;
- Synchronisation and communication mechanisms;
- Concurrent and distributed algorithms;
- Concurrent and distributed systems.

The course is for many students their first exposure to writing concurrent programs, which, as explained in Section 1.2, makes it a good target for improvements to teaching and tools based on empirical research.

The course is arranged once every year, in the autumn. Roughly 50–80 students, mostly students who have completed a bachelor's degree or an equivalent part of a master's degree, enrol each year. The research described here started with the 2005 instance of the course, so earlier versions are not described here.

The following description of the course uses the normal numeric grades 0 (fail) and the integers 1–5 (passing, from worst to best). A numeric equivalent has been provided for nonstandard grades (e.g. "pass with honours").

The course grade is, as of 2007, a weighted average of the exam (60 %), a programming assignment (2007–2009) or a set of weekly exercises (2010–2011) (10 %) and two programming assignments (15 % each). A passing grade is required in all parts to get a passing grade in the course. In 2005 and 2006, the course grade was the exam grade and there were

[1]Course web site at: `https://noppa.aalto.fi/noppa/kurssi/t-106.5600/`

	2005	2006	2007	2008	2009	2010	2011
Exam grade (e)	0–5						
Assignment 1	Trains					Weekly exercises	
Report	Always	Resubmit	Always			N/A	
Grade (a_1)	0/3/5		0/1 (late)/3/5	0–5			
Resubmissions	∞	1		0			
Assignment 2	Reactor						
Report	Always	Resubmit	Always				
Grade (a_2)	0/3/5		0/1 (late)/3/5	0–5			
Resubmissions	∞	1		0			
Assignment 3	Tuple space						
Report	Always	Resubmit	Always				
Grade (a_3)	0/3/5		0/1 (late)/3/5	0–5			
Resubmissions	∞	1		0			
Course grade	Exam grade		$0.6e + 0.1a_1 + 0.15a_2 + 0.15a_3$				

Table 3.1. Parts of the Concurrent Programming course

three mandatory programming assignments from which students could also get bonus points for the exam by getting the best grade (5) for the assignment. Students could also pass the assignment without bonus points (3) or fail it (0). In 2007, an additional grade of 1 was introduced for late and resubmitted assignments. Since 2008, all six integers from 0 to 5 (inclusive) have been used as assignment grades.

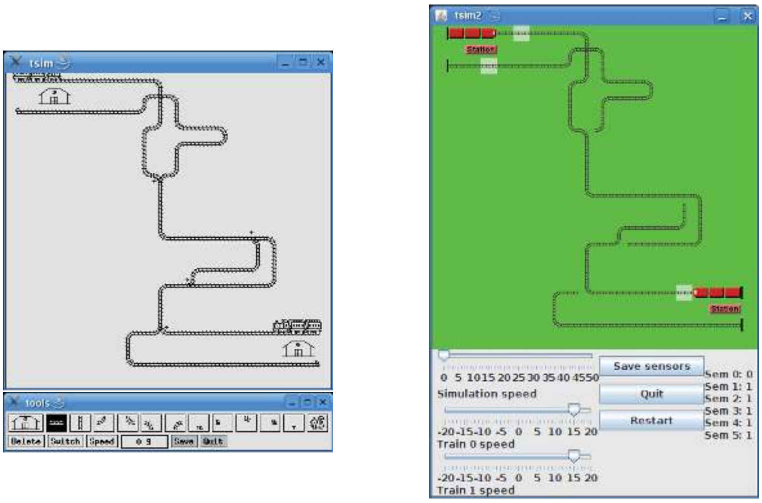
The parts of the course are summarised in Table 3.1.

The course is based on a textbook by Ben-Ari [5] (before 2006 by Andrews [3]).

3.1 Programming Assignments

Since 2005, the course has contained programming assignments which students did alone or in pairs. Initially, they were three; the first one was replaced in 2010 by a set of programming exercises. Students are required to implement a specification as a Java program and write a brief report (roughly two pages) describing their solution; the report was not required in 2006, except in the form of an explanation of errors corrected when resubmitting.

Initially, students were allowed to submit corrected versions of incorrect assignments. Since 2008, students are only given one attempt at each assignment. To compensate, they are given access to the test packages previously used only by the course staff for assessment. The test package for each assignment contains test cases for each assignment intended both to check that the specified interface has been followed and stress



Sem 0: 0
Sem 1: 1
Sem 2: 1
Sem 3: 1
Sem 4: 1
Sem 5: 1

Figure 3.1. tsim (left) and tsim2 (right) showing track used in Assignment 1

tests intended to help expose concurrency-related problems. Giving the students access to the test packages was, in particular, intended to make the requirements of the assignment more concrete and allow students to check that, for example, they implemented the Reactor API correctly. The introduction of the test packages is described in further detail in Section 4.2. The implementation of the test packages is described in Section 5.1.

3.1.1 Trains

Trains was the first assignment from 2005 to 2009. In this assignment, the students are given a simulated train track with two trains and two stations. The simulator, *tsim*, was created in 1990 for a similar assignment in a similar concurrent programming course at Chalmers University of Technology. A simplified version written in Java, *tsim2*, was introduced in 2008. The students' task is to write code that drives the simulated trains from one station to another by receiving sensor events and setting the speed of the trains and the direction of the switches on the track. The trains are to communicate with each other only through semaphores provided by the simulator. The track used in the assignment, as displayed by the simulator, is shown in Figure 3.1.

A typical solution to this assignment uses binary semaphores to ensure that segments of the track bounded by switches are used by only one train at a time. When a train has left a track segment, the corresponding semaphore is released, and before passing the point at which it must start braking to avoid entering a segment, it stops until it can acquire (one of) the following segment(s).

3.1.2 Reactor

Reactor was the second assignment from 2005 to 2009 and became the first in 2010, due to the removal of Trains. The assignment concentrates on the Reactor design pattern [66] and its application to a simple multi-player Hangman game, in which multiple players (using TCP to communicate) collaborate in trying to guess a word known to the server, letter by letter. The students' task is to, using the synchronisation primitives built into the Java language, implement a dispatcher and demultiplexer that can read several handles that have blocking read operations at the same time and sequentially dispatch the events read from these handles to event handlers and to implement a server program for a simple networked Hangman game that uses this Reactor pattern implementation.

This assignment is intended to give students an understanding of a pattern commonly used to cope with concurrency in a simple and reliable way and let them practise writing a multithreaded server program.

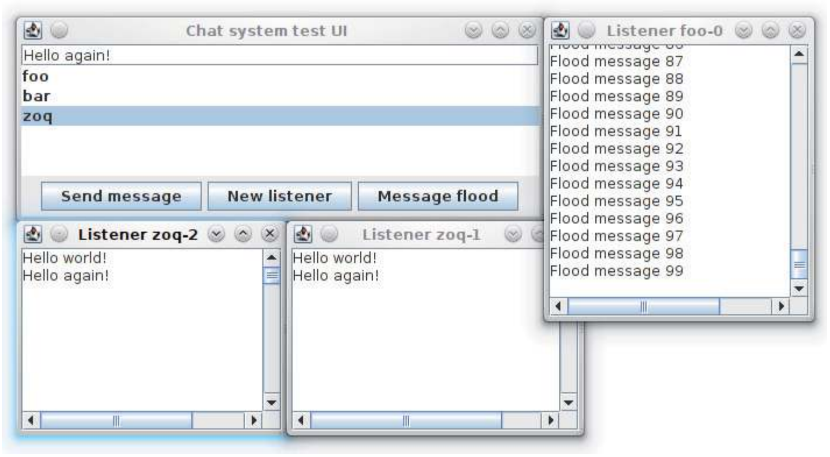
Since blocking read operations are used, the Reactor implementation must create helper threads to wait for data and then collect this information for sequential dispatching in the main thread. The Hangman implementation is single-threaded except for the read handles.

3.1.3 Tuple Space

Tuple space was the third assignment from 2005 to 2009 and is, as of 2010, the second assignment. The student implements a simple tuple space (see Subsection 2.1) containing only blocking `get` and `put` operations on tuples implemented as `String` arrays. He or she is to do this using Java synchronisation primitives and use this tuple space implementation to construct the message passing section of a distributed chat server. The student's message passing code communicates with the rest of the chat system using method calls; a simple GUI front-end to the system (shown in Figure 3.2) is provided to the students for testing purposes. This GUI contains a main window from which messages can be sent and new listeners created as well as windows showing the messages received by each listener.

This assignment lets students familiarise themselves with writing distributed systems using message passing.

Most students implement the tuple space as a collection of tuples (often



```
Hello again!
foo
bar
zoq
```

```
Hello world!
Hello again!
```

```
Hello world!
Hello again!
```

```
Flood message 87
Flood message 88
Flood message 89
Flood message 90
Flood message 91
Flood message 92
Flood message 93
Flood message 94
Flood message 95
Flood message 96
Flood message 97
Flood message 98
Flood message 99
```

Figure 3.2. Chat UI for testing of Assignment 3

without any indexing). Their `get` operation typically repeatedly checks whether a matching tuple is found and waits if not. The `put` operation wakes all the waiting threads. Only a few students wrote solutions that, when a tuple is added, wake only waiting threads with a matching pattern.

The chat system has been implemented either using the tuple space for message passing, in which case the amount of listeners is tracked and a copy of each message sent to them, or using a shared buffer in the tuple space in which all unread messages are kept and removed when they can no longer be accessed.

3.2 Weekly Exercises

The weekly exercises focus on examining the possible behaviour of concurrent programs and on writing or modifying programs to solve simple concurrency-related tasks. To make the exercises more concrete, Java is used in addition to the more abstract concurrency model used by Ben-Ari.

In the 2010 instance of the course, there were five rounds of exercises. Two or three sessions with at least one teaching assistant present were arranged for each round. Students were allowed to choose which sessions to participate in. The topics of the rounds were:

1. Concurrency models and critical sections
2. Semaphores
3. Monitors
4. Channels

5. Tuple spaces

Each exercise session consisted of four tasks, most of which were exercises from (or adapted from) the textbook [5].

In each session, the assistants presented the problem and any applicable tools, and then left the students to work on the exercises in pairs; when called on by the students, they assisted the students and evaluated their work. The students presented their solutions to the assistants who then provided feedback on the students' solutions which the students could use to improve their solutions.

4. Understanding Students Working with Concurrent Programs

As described in Section 1.2, the first phase of the work behind this thesis was motivated by a desire to understand the problem before constructing a solution. It consisted of gaining an understanding of how students approach concurrent programming and what difficulties they have in creating correct concurrent programs. This work was previously summarised in [51].

The research questions of this part of the research were:

1. What kind of defects do programmers inexperienced in concurrent programming introduce in their concurrent programs, and why?

2. Which of these defects are hard to find or understand and why?

This part of the research answers the question: "What needs do students have with regard to understanding and debugging concurrent programs?", which is used as a basis for the visualisation design in Chapter 5.

The answer to that question, in the form of the problems students have with our concurrent programming assignments, as well as solutions to some of the students' problems, are described at the end of this chapter, in Subsection 4.2.3. Those that are addressed by the visualisation are discussed in Chapter 5.

4.1 Students' Understandings and Approaches

Originally, my intent was to simply perform a quantitative analysis of the defects in our students' concurrent programming assignments and use this directly to determine in what area the students' skills or knowledge was insufficient. Unfortunately, when I did this quantitative analysis, less than half of the defects could be classified by the area in which the student made an error, due to my lack of understanding of the students' understandings of and approaches to developing concurrent programs. Since I only used the students' explanations of how their programs work as a basis for determining their errors, I had limited information on many errors. In particular, I had almost no explanations of errors of omission, such as not taking a requirement into account. Also, I attempted to distinguish between errors made in constructing an algorithm and implementing it without any indication that students make this distinction [50].

In order to gain the necessary understanding, a qualitative analysis was added to construct a model of how students approach programming assignments. This model was then used as a basis for the revised classification of the defects described in Section 4.2. More details on this study, such as illustrative quotes, can be found in Publication I and Publication II.

4.1.1 Phenomenography

The qualitative analysis is based on *phenomenography*, an approach which aims to reveal the different ways in which something is understood in a cohort [55]. Interest in phenomenographic research has increased in the computing education research community in the last decade, since phenomenography produces qualitative results that focus both on the learners and what they learn about and these results have been shown to be useful in computing education [9, 10, 69].

An experience involves distinguishing a phenomenon from its context, which involves a *focus* on an aspect of or a viewpoint on the phenomenon grounded in a *framework* that describes the context in which the phenomenon is experienced. Different people experience the world in different ways, none of which are complete understandings [55]. There is no effective way to deduce how different people think about the world from what we know about it [54].

Phenomenography is "research which aims at description, analysis, and understanding of experiences" and its focus is on understanding the variation in these experiences [54]. The outcome of a phenomenographic research project is a set of *categories of description* grouped into *outcome spaces*, where each category describes a qualitatively different way in which a phenomenon is understood or experienced. Each outcome space contains the different understandings or experiences found for a phenomenon or an aspect of it, typically organised in a hierarchy of complexity; each individual may have zero or more of these understandings [55]. Sorva [70] extends this organisation by adding branches out from the hierarchy to represent incorrect extensions of understandings. In other words, phenomenography is used to understand the qualitative and collective variation of understandings or experiences of a phenomenon.

According to Berglund [9], phenomenographic research in computer science education consists of a data collection phase and an analysis phase. In the data collection phase, the researcher interviews students about a phenomenon or a set of phenomena. A diverse sample of students (10–15 is an adequate amount [76]) is selected in order to get a rich variation of experiences. The researcher transcribes the interviews and looks for quotes that illuminate the students' various understandings and classifies the quotes into categories of description. The tentative categories usually change repeatedly as the researcher refines his analysis.

4.1.2 Data Collection

At the end of the 2006 instance of the course, I interviewed eight students regarding the Tuple space assignment (see Subsection 3.1.3). The interviews were conducted after the results of the initial submissions were published, but before the deadline for submitting correct solutions. The focus of the interviews was on how the students approached the programming assignment, especially the reasoning behind their design, with a focus on the defects found in their programs.

Twelve groups of students (eight students who did the assignment alone and four pairs who collaborated on the assignment) were selected for interview based on the defects found in their solutions for the Tuple space assignment. In order to maximise the variation of experiences, I chose groups with different types of problems with their code, as determined by the teaching assistant who graded the assignments. Ten out of 31 groups that failed the (initial submission of the) assignment and two out of 24 that had passed the assignment were invited to an interview. Out of these groups, seven of the failing groups (six single students and one pair; a total of eight students) agreed to participate and were interviewed. The pair of students was interviewed together. This sampling process is summarised in Tables 4.1 and 4.2. Although no students who had passed the assignment in their first attempt participated (and the sample was smaller than intended), the results in Subsections 4.1.4, 4.1.5 and 4.1.6 show that even the students from the failing groups had a diverse set

Groups	Fail			Pass			Total		
Group size	1	2	Total	1	2	Total	1	2	Total
Did assignment	19	12	31	12	12	24	31	24	55
Selected	7	3	10	1	1	2	8	4	12
Participated	6	1	7	0	0	0	6	1	7

Table 4.1. Overview of student groups selected for interviews

Students	Fail			Pass			Total		
Group size	1	2	Total	1	2	Total	1	2	Total
Did assignment	19	24	43	12	24	36	31	48	79
Selected	7	6	13	1	2	3	8	8	16
Participated	6	2	8	0	0	0	6	2	8

Table 4.2. Overview of students selected for interviews

of understandings, including advanced ones (which is hardly surprising since more than half of the groups had failed the assignment). In other words, the sample appears to have been adequate despite having fewer participants (especially successful ones) than intended.

The interviews were in the form of a free-form conversation based on a set of prepared questions that were used to open up topics for discussion, and lasted roughly 30–60 minutes. The first questions in the interviews established the students' backgrounds in programming, particularly concurrent programming. The rest of the questions were about tuple spaces, the design decisions made by the students in solving the assignment, their approach in determining whether their solution was satisfactory, and problems found by the students or the teaching assistant.

I recorded the interviews using a single microphone and transcribed them. I also wrote down the main points of the interview directly after the interview. To make it easier to discuss the students' development process and programs, the students were, before the interview, given printed copies of the code they submitted. Before the interview, I also explained the purpose of the interview and confirmed that the student accepted that I recorded the interview, allowed collaborating researchers access to anonymised transcripts and published selected anonymised quotations.

4.1.3 Analysis

I did the analysis in discussion with Anders Berglund and, in a later stage, Lauri Malmi. After discussing the contents of two interview transcripts, the iterative phase of the analysis was performed. In each iteration, I read through the transcripts looking for relevant quotes and formed categories based on these, building on the results of the previous iteration. I grouped the categories into outcome spaces by the issue they describe. Berglund and Malmi then examined these categories and made suggestions on how to improve them. The resulting categories from the last iteration are presented in the following section.

In the first iterations, the analysis focused on finding as many quotes as possible that illustrated ways in which the interviewees understood concurrent programming and approached the assignment. First, quotes were grouped together if they appeared to express similar standpoints. They were then grouped together into tentative categories representing similar understandings. These understandings were then grouped by phenomenon into tentative outcome spaces.

The categories changed in many ways during the analysis process. Starting from the third iteration, the emphasis of the analysis shifted to refining the preliminary categories. Quotes we deemed not to fit the research approach or area of interest of this particular study were left out. Phenomena with only a few quotes were left out due to insufficient data.

Most of the quotes fit into a phenomenographic framework. However, the sources of failures taken into account by students are several related phenomena and are therefore described in a different fashion.

The results of this study are in the form of phenomenographic outcome spaces describing how students understand tuple spaces (Subsection 4.1.4), how they understand the goal of the programming task (Subsection 4.1.5) and how they understand developing a program (Subsection 4.1.6).

4.1.4 Students' Understandings of Tuple Spaces

Tuple spaces were described by the interviewees in a number of different ways that all describe the same data structure from different viewpoints. Tuple spaces can be seen as a 'black box' with operations (a specification of an interface), an implementation, something that can be used in a program to achieve a goal and as one of many possible ways to achieve something in a program. The categories are summarised in Table 4.3. For details, see Publication I.

The first three categories are more or less required by the assignment, as a specification is provided for the students to implement and use. The fourth category was not required and is an encouraging sign of students going beyond the immediate requirements of the assignment.

Label	**What is the tuple space described as?**	**What is in focus?**	**Framework**
Specification	Operations on tuples	The properties of the operations	-
Implementation	Data structures and code	How a tuple space implementation works or could work	Part of a program
Usage	A tool to achieve a specific subgoal in a program	What a tuple space can be used for in a program	A program
Evaluation	A better way of coordinating distributed systems	The advantages of using the tuple space	Other communication and distributed data storage mechanisms

Table 4.3. Categories of tuple spaces

Label	**The purpose of the programming task**	**What is in focus?**	**Framework**
Assignment	To meet the requirements of the university setting	The university setting's requirements	University setting
Ideal problem	To produce a program that functions within the university setting's requirements	The program itself	University setting
Working solution	To produce a solution to a problem beyond the university setting	The program itself	An environment beyond the university setting
Possibilities	To solve a problem with potential for future development	Possibilities for future development	An environment beyond the university setting

Table 4.4. Purposes of the programming task

4.1.5 Students' Understandings of the Goal of Program Development

The interviewees express the purpose of a programming assignment in different ways shown in Table 4.4, reflecting different perspectives on both learning and programming. The different purposes can be seen as a progression that starts with a focus solely on meeting the university's requirements. In the second category, the program's functioning becomes the focus. In the third, the program is seen beyond the university setting. Finally, the focus shifts from the program itself to the possibilities for future development it raises.

The students mentioned three types of failure sources: other systems, the user and the programmer. They are summarised in Table 4.5. This list of failure sources is closely related to the outcome space of purposes of

Source	**Effect on program design**
Systems	Tolerate other systems' failures
Programmer	Minimise chances and/or consequences of programmer error
User	Tolerate user error

Table 4.5. Sources of failure

Label	**What is developing and debugging described as?**	**What is in focus?**	**Framework**
Implementation	Writing and debugging code	The code and its execution	Relevant programming language constructs
Solving technical problems	Finding solutions to a series of technical problems	Central ideas of concurrent programming	The program, seen as a technical entity
Producing an application	Finding solutions to real-life problems	What users need from the program	Context in which program is used

Table 4.6. Categories of developing and debugging

the programming task in that they both describe the context in which the students' programs are expected to function.

In order to keep the assignment simple, students were told that they could ignore the possibility of network failure in the assignment. Some students took the possibility into account anyway. Similarly, students were told not to bother with checking for user error in these assignments. Nevertheless, user error was used by one student as a reason to ignore the specification in the handling of a special case (empty messages in the chat system). In both cases, students have gone beyond the assignment's requirements and designed their code to deal with 'real-world' problems. I did, however, expect that students would try to minimise the consequences of programmer error by sticking to simple solutions, since this also decreases development time and improves the chances of meeting the assignment deadline.

These understandings are described in more detail in Publication II.

4.1.6 Students' Approaches to Developing Programs

The interviewees understand the process of developing and debugging their program in many different ways: as simply writing and debugging code, as solving a technical problem and as producing an application. Each category differs from the previous in that it moves further from the actual program code toward a user-oriented 'big picture'. This is summarised in Table 4.6. For details, see Publication I.

Software engineering emphasises ways of managing complexity and quality that rely on different perspectives on the software that is being developed. The categories of developing seen here are similar to several of the different views needed in many common software development processes.

The implementation categories of both developing and tuple spaces are obviously necessary for practical software development and in an assignment where students are required to implement a tuple space. The solving technical problems category can be seen as design. The specification the students are provided with is more or less a finished architecture design. Hence, students need not do any design other than determining the appropriate data structures and algorithms to use.

The change in perspective between solving technical problems and producing an application parallels the change in perspective to include a usage context outside the university seen in Subsection 4.1.5. The first two categories of developing can also be considered facets of what Ben-David Kolikant [6] calls the programmer's perspective, which includes reasoning in terms of both the concurrency model and the implementation.

The application production category, which is suited for requirements analysis, is another unexpected example of going beyond the assignment's requirements into the real world. In assignments like this one, it can distract students from the intended goal of producing a reliable implementation of a specification. The student may run into trouble when programming professionally if he chooses to ignore specifications in favour of his own interpretation of the requirements for the program. In this case, the expectation that the student or programmer does not deviate from the specification should be made clear. However, it can be argued that students should be given opportunities in programming courses to practise determining system requirements, as this is a useful skill for them.

The students showed a wide range of understandings of how a program is developed, which was reflected in their development processes. The process understandings are summarised in Table 4.7. For details, see Publication II.

The six categories of development process models can be seen as a progression from an unstructured or informal development process to a structured one. In the first category, the students saw no need for a structured process. In the next two, they saw their lack of a structured process as a problem. In the following category, a solution from the student's earlier work is used but found problematic. The two final categories reflect well-known and accepted ways to find a solution to a programming problem.

The students had approaches to testing that reflected different understandings of the intent of testing. These are shown in Table 4.8. The testing approaches range from the superficial and, by the students' own admissions, inadequate, to testing with increasing degrees of purposeful-

Label	**What is the process understood as?**	**What is in focus?**	**Framework**
No design needed	Writing code directly based on requirements	Writing code	Requirements and code
Trial and error	Writing code to find a solution that meets requirements	What code works?	Requirements and code
Coding to understand	Writing code to understand the requirements	Understanding the requirements	Requirements and code
Inertia from previous work	Writing code based on own previous work	Writing code	Requirements, code, own experiences
Apply known technique	Using a known technique to structure the solution before implementing it in code	Structuring the solution	Requirements, code, ways to structure code
Adapt known solution	Writing code based on others' previous work	Structuring the solution	Requirements, code, solution archetypes

Table 4.7. Software development process models

Label	**What is testing understood as?**	**What is in focus?**	**Framework**
Unplanned	Trying out the program to see if it works	How program reacts to input	Features, test inputs and outputs
Breaking the system	Trying to get defects to manifest as failures	Finding inputs that make the system fail	Features, test inputs and outputs
Covering different cases	Trying to show the program can not fail	Finding a set of inputs that gives sufficient reassurance the program will not fail	Features, test inputs, outputs and coverage
External testing support needed	Trying to show the program can not fail, which a programmer cannot reliably do alone	Getting someone else to find a set of inputs that gives sufficient reassurance the program will not fail	Own testing ability and others'
Testing inadequate	Part of ensuring the program is correct	Limitations of testing	Own testing ability and others'
Proof necessary	A complement to a correctness proof	Limitations of testing	Testing and proving correctness

Table 4.8. Testing approaches

ness, awareness of the limitations of testing and, finally, complementary approaches to determining program correctness. Especially the last few categories show an understanding of how nondeterministic program behaviour affects testing. For details, see Publication II.

4.2 Students' Defects in Concurrent Programs

In order to determine what types of defects exist in students' programs and then work out how to effectively address them through e.g. changes to the assignments or teaching or visualisation tools, I analysed the programs written by students for all of the programming assignments in the 2005, 2007 and 2008 instances of the Concurrent Programming course.

As described in Section 4.1, this part was originally intended to be a self-contained empirical study, but the classification of defects was found to be difficult to perform and the phenomenographic study was done to help understand how students approach the assignment in order to form a meaningful classification of defects. For further details, see Publication III. How the classification changed is described in more detail in Subsection 4.2.3.

4.2.1 Data Collection

The obvious source of information on defects in students' programs is the programs themselves. Furthermore, since students' programming assignments are graded by checking them for defects, the grading process already incorporates much of the necessary defect detection work. This work was done primarily by hand by myself and the other teaching assistants in the Concurrent Programming course working according to specifications I provided. I checked the other assistants' work and helped them as needed. For the 2005 course, I did all the assessment myself. In 2006, another teaching assistant did most of the assessment, in part using his own classification. The results for this year are therefore omitted. Since 2007, assessment of assignments in the course has been divided between several teaching assistants. To ensure consistency, a defect classification I made based on the 2005 results has been used as the basis for assessment.

Only the initial submissions of each assignment were considered, since resubmissions would (hopefully) not add defects and would have complicated the quantitative analysis.

4.2.2 Analysis

I have classified the defects found in the students' programs using two separate classifications. One classification is by the underlying error (to the extent it can be determined), which helps determine what understanding or skill the student lacks. In the other classification, defects are divided based on whether the failures they cause occur deterministically.

Non-functional requirements (such as using a mechanism that is not allowed) can be interpreted as resulting in failure by considering the execution of a call to a forbidden feature as a failure or by considering the operation to behave incorrectly. For example, threads that are supposedly running on different machines would see separate locks instead of one. Since many of the non-functional requirements in programming assignments are based on a notional execution environment, it is natural to use the failure induced by this type of error in the notional environment for classification purposes. This also makes this classification by failure consistent when notional limitations are made real, as in our Concurrent Programming course.

Defects and failure are defined here with respect to the written assignment specification, as interpreted by the person assessing the assignment.

Classifying Defects by Error

Errors can be classified by the task the programmer was performing when he made the error. This allows one to easily determine the knowledge and skills involved and provide feedback to the student to help him understand his error.

Inadequate testing can be considered a separate problem as it does not introduce defects into the code, although it (by definition) may prevent defects from being found.

I initially formed this classification by grouping together defects based on similarities in how they deviate from the corresponding correct solution; this is conceptually similar to the goal/plan analysis of Spohrer and Soloway [71]. With some minor refinements and additional defect classes, this classification was used as a basis for assessment in 2007 and 2008. I combined these defect classes into larger classes based on the distinctions I wanted to make when assessing the students' programs. As this classification proved to be impractical, I revised this classification based on the refined classification of the defects and the results of the phenomenographic study.

These classifications are presented in Subsection 4.2.3.

Classifying Defects by Failure
An alternative classification is by the type of failure; this is relevant for testing and debugging.

Deterministic failures occur consistently with certain inputs and are thus easy to reproduce. This allows traditional debugging, based on repeated executions, single-stepping and breakpoints and examining program states, to be used.

Nondeterministic failures are hard to duplicate. Debugging from execution traces is thus easier than traditional debugging, since the failure only needs to occur once while logging is being done.

This classification was done by examining the effect of each defect class on program execution through testing and by reasoning about the effect of the defect on the program's behaviour.

4.2.3 Results

Initially, I constructed a classification based on the assessment criteria of the Concurrent Programming course and on the classification of Eisenstadt [21]. The results of this analysis can be found in [50]. Errors were divided into:

Concurrency errors Concurrency-related misconceptions or design errors

General programming errors Misconceptions or errors related to the programming language or non-concurrent algorithms

Environment errors Errors related to the environment in which the assignment was performed

Goal misunderstandings Misunderstandings of the requirements of the assignment

Slips Slips or other careless errors

Unfortunately, only a small amount of the students' errors could be unambiguously placed in one of the above categories; asking students to explain the reasoning behind their entire solution in a written report did not give enough information to reconstruct their errors. Another problem was that some errors can fit into many classes.

As described in Section 4.1, one of the goals of the phenomenographic analysis was to provide an understanding of how students understand concurrent programming in order to analyse their defects meaningfully. Hence, the outcome spaces described earlier in this chapter led to some changes to the classification. While it would be possible to distinguish between errors made in designing an algorithm to solve a problem and implementing it, students did not seem to make this distinction in the way they worked, as seen in Table 4.7. Hence, this distinction was removed. Similarly, since no evidence was found that students divide programs or the development thereof into sequential and concurrent parts, that distinction was also removed. Also, in a concurrent programming assignment, most programming errors are in some way related to concurrency; the question of where to draw the line has no clear answer. Some students did, however, show an awareness of the difference between deterministic and nondeterministic failures, as seen in Table 4.8. Table 4.7 also shows that students may find understanding the requirements of the assignment to be a source of difficulties that is great enough to structure their work around. In Tables 4.4 and 4.5, examples of alternative understandings of the goal of an assignment, which lead to understanding the requirements differently, can be seen.

The distinction between the programming and the assignment environments is made in order to determine which errors are irrelevant in assessing the students' concurrent programming knowledge and skill and could be reduced or eliminated by changing the assignment.

The resulting categories are:

Requirement-related error A part of a specification has not been understood correctly or not been taken into account properly when designing or implementing. Some understandings of the goals of a programming task can lead to this. Explaining the requirement and a failure in which it is violated should be enough to explain this type of error to the programmer. Communicating requirements as tests with a clear pass/fail indication can help programmers detect these. Eliminating this type of error should be a priority when designing programming assignments.

Programming environment-related error Misconceptions of the goals of a programming task that relate to the target environment, such as considering unbounded memory usage to not be a problem, can result in this type of errors. Alternatively, there may be something about the language, API or other aspect of the execution environment the programmer has not understood, in which case explaining the relevant aspect (e.g. by referencing a specification) may help. Finding problems in students' knowledge of a programming environment in general can be helpful to them, but secondary in many advanced courses to the actual topic of the course.

Assignment environment-related error Misconceptions about the framework provided for a programming assignment can also result in errors. These are distinguished from errors in the previous category in that they relate to systems that are only used in this particular programming assignment. Therefore, these errors, like the requirement-related errors above, can be seen as indications that the assignment is confusing. This type of error is avoided if no framework is provided (as in the Reactor assignment); large amounts of this error suggest that the framework is confusing and should be simplified.

Incorrect algorithm or implementation Programmers may introduce errors when creating or implementing an algorithm. These errors vary from creating an algorithm that does not work in all necessary cases to forgetting to handle a case. Showing a programmer how his code fails is enough if the error is not due to insufficient or incorrect knowledge. A programming assignment should allow students to make errors of this type, as they provide valuable indications of deficiencies in the students' knowledge or skill.

In each assignment, different subtypes of the aforementioned errors can be distinguished. They are described in the following to the extent they merit interest either by being common, surprising or because they have consequences for the teaching or visualisation development. A more detailed list of defects can be found in [51].

Table 4.9 shows, for the three yearly instances of the course that I have analysed, the total amount of submitted programs and the amount of defects found in each class in both the error- and failure-based classifications. Note that the amount of students decreases each year between the assignments; this is due to students dropping out of the course.

	Trains			Reactor			Tuple space		
	2005	**2007**	**2008**	**2005**	**2007**	**2008**	**2005**	**2007**	**2008**
Submissions	128	60	52	107	51	40	84	49	39
Requirement	53	10	11	93	112	38	93	49	21
Programming	3	0	0	15	11	1	0	0	0
Assignment	70	20	10	0	0	0	3	0	0
Incorrect	28	16	5	51	56	17	70	51	36
Deterministic	39	2	0	94	102	8	98	58	28
Nondeterministic	115	44	26	65	77	49	68	42	29
Total	154	46	26	159	179	57	166	100	57

Table 4.9. Defects found in assignments

The large amount of nondeterministically manifesting defects in students' programs demonstrates a clear need for debugging tools that do not rely on repeated execution and stepping as is the traditional approach. Instead, the information needed for debugging should be captured for post-mortem examination from a failing execution when it occurs.

Understanding Requirements

Initially, students had problems understanding the requirements of the tasks correctly. Many of these problems were resolved by making it easier for students to notice that they had misunderstood the requirements by providing them with a testing environment in which following the requirements was necessary to make the program work correctly. In other words, notional limitations were made real. In the Trains assignment, students' code could easily access information about the simulated trains that was not supposed to be available and communicate with each other in ways that students were not allowed to use in the assignment. These problems were eliminated in the 2006 version of the assignment through a redesign of the simulator API. Similarly, in the Chat assignment, about half of the requirement-related errors in 2005 were due to the requirement to pretend that the chat system was running in a distributed environment. In later years, the example code provided to the students set up a system 'distributed' over several processes, making this error much less common.

The intended semantics of the methods the students were to implement were hard for some students to understand. These problems were easily mitigated by clarifying the assignments and related material. A few of the submitted Reactor implementations in 2005 submitted all events to all event handlers. It was found that Schmidt's pseudo-code for the Reactor implementation [66] can also be interpreted this way; for the 2006 course, I wrote a simpler explanation of the Reactor pattern that eliminated this ambiguity. A similar ambiguity involved the amount of events to dispatch for each call to `handleEvents()`.

Until the students were provided with test packages in 2008, many made changes to the Reactor API or the way it uses threads to simplify the Reactor or the Hangman server. These errors account for roughly a third of the requirement-related errors. Similarly, problems with input and output formats and the rules of the Hangman game were common until the test packages were introduced. The semantics of the tuple space also caused problems. Most of these errors involved limiting the tuples in some way, such as considering the first element in a tuple to be a `String` used as a key as in the textbook. Some solutions changed the blocking, matching or copying semantics of the `get` operation. The most commonly ignored requirement of the chat system's functionality was that messages stay in order. Again, the test packages helped students discover their misunderstandings.

Many problems were related to keeping memory use under control. The most commonly ignored requirement was to ensure that the Reactor does not buffer an arbitrary amount of data if it cannot handle events quickly enough. In 2005 and 2006, this was not considered a problem, but in 2007 and 2008 it was found to occur in the majority of submitted solutions. The fact that it remained common in 2008 is probably due to the fact that the test package did not include a test case for this scenario, which it does now. The increase in defects between 2005 and 2007 can be mostly ascribed to this change in requirements.

Cleaning up after a handle is removed from use also appears to often have problems, as does ensuring memory use stays within reasonable limits. Similarly, getting rid of unused tuples is a difficult area, accounting for roughly a third of the errors in this category. In some cases (especially those where no cleaning up is done at all), this could be because cleanup is not considered by the student to be relevant to the assignment (i.e. the intended execution environment is not understood to have limited memory). However, most of the reports of students with this error suggest an awareness of memory limitations and a choice to use a simple algorithm that wastes memory rather than a complex one that conserves it, suggesting this is a compromise to save time and/or decrease chances of a programming error. This and the understandings of goals in Table 4.6 suggest that one could encourage students to be more careful with memory management by demonstrating how failure to clean up unused data can lead to failure and that it will be considered an important factor when their assignments are assessed.

In the Trains assignment, a deterministic failure would occur in every possible execution, making it easy to detect. It is therefore not surprising that all the deterministic failures are due to misunderstandings of the requirements.

Understanding the Programming Environment

In line with the results of Spohrer and Soloway [71], very few of the errors were programming environment-related. Example code was provided to illustrate the use of problematic constructs (opening sockets on free TCP ports, the `break` statement in Java).

Understanding the Assignment Environment

The train simulator used in the first assignment proved to pose problems of its own by introducing issues of train length, speed and timing that cause problems for students unrelated to the learning goals of the assignment and hence distract the student from the concurrent programming the assignment is about. This was one reason why the Trains assignment was replaced.

Incorrect Algorithms or Implementations

Errors involving incorrect algorithms or implementations are of particular interest for several reasons. Unlike the other categories of error described above, it is not desirable to modify the assignments to eliminate these errors. On the contrary, the assignments can be seen as a test of the students' understandings and skills; errors of this type are symptoms of insufficient or incorrect understandings or lack of skill. These are the errors we want students to understand and learn from. Hence, the visualisation design in Section 5 is strongly affected by these.

A large part of the errors in the Train assignment were in the train segment reservation code. Some solutions consisted of subsolutions that did not combine properly (Mismatch between Modules as described by Knuth [40]) or relied on train events happening in a specific order. Others had more localised problems. A few unnecessarily complex solutions introduced the possibility of deadlock by making segment reservation or release involve a sequence of operations that could be interrupted by the other train.

Similarly, many Reactor solutions, especially in 2007, failed to correctly handle events that were left undispatched after handle removal or received after handle removal. Some failed in other ways to correctly remove handles from use. The increase in 2007 may be due to improved assessment guidelines. Again, the testing package makes this type of error easier to detect but not necessarily understand or correct.

Several different cases were found of incorrect buffer management algorithms in the Reactor implementation, such as misusing status variables, circular locking dependencies, notifying the wrong thread or at the wrong point, or overwriting or losing messages. Only a few cases of using collections or variables without necessary synchronisation were, however, found.

The tuple space proved to be unproblematic to implement. Only a few cases of critical sections having the wrong extent and `notify()` being used instead of `notifyAll()` were found. More common was for the tuple space to match patterns against tuples incorrectly. A few solutions also corrupted their own data structures while executing.

Initialisation proved to be surprisingly problematic, especially, interestingly enough, the `ChatServer` constructor for connecting to an existing chat system, which often did not replace all the tuples it got. This invariably causes the system to deadlock when the third server node is connected. Outside this method, forgetting to replace tuples was uncommon.

The buffer of messages that the chat system has to maintain for each channel proved to be problematic, with failure to handle a full buffer or simultaneous writes, insufficient locking of the buffer or related sequence numbers and indices being common in 2005 and 2007. The reason for the sudden decrease in 2008 is unknown. Circular locking dependencies, on the other hand, became much more common in 2008.

Only a few errors were obvious implementation slips, such as forgetting a `break` or `else`, parenthesising a logical expression wrong, making an array one element too small or accidentally duplicating or commenting out code, starting a thread twice, and using a stack instead of a queue.

4.3 Conclusions

Students often misunderstand requirements. Without a way to easily check that their program meets all the requirements, it is hard for students to notice that they have made an error in understanding the requirements. Giving the students test code helps them discover such errors. However, they may still decide to ignore a requirement if doing so makes the assignment simpler. In particular, students are often accepting of memory leaks.

The Trains assignment introduced several needless complications, such as inertia and sensor positioning. Errors related to these complications resulted in several defects that were hard to detect and diagnose. This contributed to the decision to replace this assignment.

Incorrect interactions between different parts of students' solutions accounted for many of the defects in students' programs. This was in part because many such defects do not result in failure unless different features of the program are combined in a certain way. Also, such failures often involve causes and effects that are in separate parts of the program. Both of these are reasons that Eisenstadt [21] presents for bugs being hard to find. Many defects involve incorrect algorithms.

Many of these defects do not consistently result in failure, making them hard to detect and analyse. Also, the symptoms of many failures are deadlocks and involve incorrect interaction between threads. In the following chapter, I describe the program visualisation tool I constructed to help students deal with these defects.

5. Visualisation System Design and Implementation

Based on the empirical results in Chapter 4, I designed and implemented a visualisation system called *Atropos* to help students understand what happens in a concurrent program, especially when it does not behave as expected.

The research questions for this part of the research were:

3. What is a ‘good’ visualisation of incorrect executions of concurrent programs, where ‘good’ means a visualisation that can enable students to find and to understand the underlying defects in their programs?

4. How can such a visualisation be created algorithmically from an execution of a program?

This answers the question: “How can the needs from the previous part best be addressed through visualisation?”

In this chapter, I present the reasoning why it seems that Atropos produces a ‘good’ visualisation in this sense and how it does it. Whether Atropos actually is ‘good’ is addressed in Section 5.3 and Chapter 6.

The design is described in more detail in Publication V and the implementation in Publication IV.

The visualisation is primarily intended for errors related to incorrect algorithms or implementation or the programming environment, as explained in Subsection 4.2.3, since showing students how their programs behave in this context is most likely to help them. In contrast, changes to the assignments were made and students provided with test packages in order to eliminate requirement-related and assignment environment-related errors.

Atropos is primarily intended to be used by students independently to

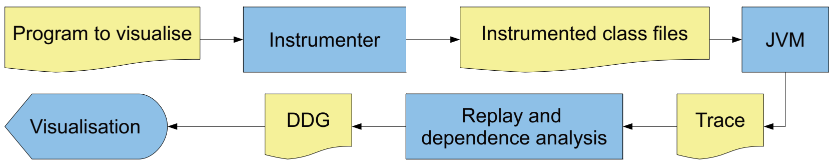


Figure 5.1. Information flow between parts of Atropos

find and correct defects in their programs and learn from their mistakes. In addition, it can be used by teaching staff to explain how concurrent programs work or why a student's program fails to work correctly, or by students to explore the behaviour of concurrent programs.

As noted in Section 2.3.1, the slicing strategy provides a clearly-defined systematic way to trace backwards through a program from a symptom to the failure that caused it and hence identify the underlying defect. It also requires tool support to be effective, and, as noted in Section 2.4.2, it has been shown to help novice programmers at least.

DDGs are of particular interest for concurrent programs, as interactions between threads are clearly shown as edges. This means that a DDG can help students identify unexpected interactions between threads. By showing how different parts of a program interact, they can also help find causes for program behaviour even if they are in another part of the program (which would be useful, as explained in Section 4.3) and isolate relevant information from a large trace. Using execution traces also removes the need to re-execute the program when examining defects that nondeterministically manifest as failure; as noted in Subsection 4.2.3 they constitute a large part of the defects students fail to eliminate from their programs. Also, many defects involve insufficient or incorrect synchronisation and can result in data races.

For these reasons, Atropos was designed to enable its users to explore a DDG and help them effectively apply a backward debugging strategy to a concurrent program. In essence, for every line of code that was executed in a program, Atropos can show why it was executed and where the data it used came from.

Atropos, like the replay systems described in Subsection 2.4.1, generates an execution trace file that can be replayed at a later date. This is particularly useful in our context: a teaching assistant can send a student an execution trace file that demonstrates defects that the assistant found in the student's code, allowing the student to examine the failing execution in more detail.

The execution traces are collected through bytecode instrumentation of class files. For simplicity, compatibility and ease of debugging the instrumenter, this is done by creating instrumented copies of the class files before execution.

The trace files are then used to direct the replay and dependence analysis of the program; the result of this is a dynamic dependence graph of the execution. This graph can then be explored through the visualisation.

This process is summarised in Figure 5.1. In the following sections, I explain the parts of the process in detail. As a running example, I use the program in Appendix B.

5.1 Collecting and Replaying Execution Traces with Dynamic Dependence Analysis

In this section, I describe how Atropos collects execution traces and produces a dynamic dependence graph from them. Details can be found in Publication IV.

The replay and runtime analysis-based tools mentioned in Subsection 2.4.1 proved problematic due to their modifications to the JVM, instrumentation or their assumptions on the programs they collect information on, especially their inability to handle data races. These issues limit the possible executions in such a way that many educationally relevant types of failure can not be examined. The instrumentation used by ODB [46] effectively prevents data races through the synchronisation it uses. RetroVue was ruled out because its source code is not available.

Another problem with most of the existing replay software is that it is geared toward reproducing execution to allow it to be examined in a traditional, state-oriented, debugger. This is impractical for more complex analysis of data flow or inter-thread interaction.

Since bytecode instrumentation is a more appropriate approach, it makes sense to extend an existing bytecode-based execution tracing system. Although ODB would also have been appropriate as the basis for implementing the tool we need, I chose to extend the instrumenter used in my previous tool MVT. In addition to the advantage of familiarity, it has two additional advantages. Its instrumenter was already in use in the test packages in our Concurrent Programming course, which makes it easier to integrate Atropos with the tests used in the course. The test packages primarily consist of JUnit [38] tests using the GroboUtils [29] extensions for multithreading. In order to improve the chances of interleavings that result in failure occurring, a bytecode instrumenter derived from MVT is used to add random delays in a manner similar to that described by Goetz et al. [26] or Stoller [74]. Also, MVT was developed with the intention of extending it to generate DDGs [49].

5.1.1 Instrumentation and Execution Trace Collection

Atropos uses BCEL [15] to instrument Java programs to collect execution traces. The execution trace consists of a partially ordered sequence of executed JVM operations and any data manipulated by these other than operand stack values and local variables (i.e. any data that cannot be easily reconstructed by executing simple and deterministic operations). The sequence is partially ordered because, as noted in Subsection 2.1.1, the behaviour of shared memory in Java is based on the partial order of the happens-before relationship, and some operations can occur simultaneously.

Also, it is desirable for the instrumentation to not induce additional happens-before relationships between existing actions, as this could eliminate data races that we wish to examine. To do this, the instrumentation must use thread-local data structures to collect information on thread-local operations. Obviously, shared data structures are necessary to collect information on inter-thread interaction. If a happens-before b, the instrumentation can record this by storing an identifier for a in a variable associated with the mechanism used to induce the happens-before relationship with b. The instrumentation can then, after b is executed, safely read this identifier, thanks to the happens-before relationship, at which point the instrumentation for b has identified the happens-before relationship. Atropos adds fields to objects to track who last held a lock and when try released it. Similarly, `volatile` variables are replaced with objects containing a value of the original variable and which operation wrote the value.

In the example in Appendix B, the instrumenter replaces `volatile boolean` variables `wantp` and `wantq` with `volatile VolatileBoolean` variables called `$_$volwantp` and `$_$volwantq`. The `VolatileBoolean` objects contain `long` fields identifying the thread and operation that wrote the last value to the original variable as well as a `boolean` field containing the value itself. Read and write operations are also replaced. For example, in line 19, the original `wantp = true` compiles to the following (simplified from the output of javap):

```
12:  iconst_1
13:  putstatic boolean Second.wantp
```

After instrumentation, this becomes:

```
49:  iconst_1
50:  invokestatic void DebugCalls.preNoInputs()
53:  new VolatileBoolean
56:  dup_x1
57:  dup_x1
58:  pop
59:  invokestatic long DebugCalls.getId()
62:  invokestatic long DebugCalls.getOp()
65:  invokespecial VolatileBoolean(boolean, long, long)
68:  putstatic VolatileBoolean Second.$_$volwantp
71:  invokestatic void DebugCalls.postNoValue()
```

Methods `DebugCalls.preNoInputs()` and `DebugCalls.postNoValue()` are used to log that an operation was started and finished (in order to determine whether the thread reached and executed this instruction), but what instruction was executed and the value that was written can be reconstructed from the results of earlier operations.

The read of `wantq` is similarly converted from:

```
0:   getstatic boolean Second.wantq
```

to:

```
9:   invokestatic void DebugCalls.preNoInputs()
12:  getstatic VolatileBoolean Second.$_$volwantq
15:  dup
16:  ifnonnull 26
19:  pop
20:  getstatic boolean Second.wantq
23:  goto 29
26:  invokevirtual boolean VolatileBoolean.readObject()
29:  nop
30:  dup
31:  invokestatic void DebugCalls.postIntOut(int)
```

Note that the original `volatile` variables are used in the read operation if the replacement does not have a value; this allows the default values of these variables to be handled as usual. `VolatileBoolean.readObject()` not only returns the value encapsulated in the `VolatileBoolean`, it logs the thread that wrote it. `DebugCalls.postIntOut(int)` is used to log an `int` value that cannot be reconstructed.

When a thread's list of operations exceeds the size limit or the thread or JVM terminates, the thread's list of operations is dumped to disk. At this point, object references are resolved to unique id numbers using a global table of weak references, allowing objects that are no longer in use to be garbage collected. This table is protected by a `synchronized` lock; in order to minimise the effect of the instrumentation on how threads interleave, it is desirable to minimise accesses to it while a thread is still running. After the JVM terminates, the lists of operations are compressed in a ZIP file.

5.1.2 Replay and Dynamic Dependence Analysis

Rather than performing straightforward replay of a concurrent execution, Atropos replays the program in its own interpreter, which constructs a dynamic dependence graph of the execution. Any thread is allowed to execute for which everything that should have happened-before the current operation has already been executed; in other words, everything is executed in an order consistent with the happens-before order. In the absence of data races, this ensures that whenever a variable is read, it has an unambiguous value and the last write that was performed is the value that was read. In the example, when the log entry generated by `VolatileBoolean.readObject()` is reached, the replay engine only continues the execution after the write operation indicated in the log entry has been executed.

The replay in happens-before order allows a vector timestamp for each operation to be created as described by Mattern [56], with happens-before relationships forming the messages of Mattern's vector clock algorithm.

However, since we allow data races to occur, many different values for a variable may be available for reading at a time. When a read is performed, the corresponding write operation must be found from the set of writes that it is *allowed to observe* [27, §17.4]. This set of observable writes can easily be determined by using the vector timestamps described above to find the last operations in each thread that happened-before the read.

Re-ordering of operations may cause writes that are later than reads [27,

§17.4.5]; the data dependency must then be determined at a later time. This does not affect the replay itself; it merely means a placeholder must be left for replacement with the correct data dependency after the right write operation has been performed. In this case, it becomes necessary to make sure the write does not happen-after the read, but otherwise the process is the same as above.

Control dependencies are traditionally calculated statically, and in any case, as explained in the following section, they are not relevant to our visualisation.

5.1.3 Technical Evaluation

In order to determine whether using Atropos is technically feasible in our context, I collected traces from stress tests of all 40 of the students' solutions to the Reactor assignment of our Concurrent Programming course in 2010. The stress test (from the test package provided to the students) feeds 1000 messages from each of 20 threads into one of 20 handles in an attempt to find synchronisation problems. This is repeated 50 times. The issues examined were the sizes of the traces, how much slower the testing was when collecting execution traces and whether collecting the traces prevented the tests from failing in the presence of defects.

The tests described in this section were all done on a 64-bit Ubuntu 10.04 workstation with an Intel Core 2 Quad Q9400 CPU and 4 GB of RAM.

Size of Traces

The execution traces were, unsurprisingly, very large in the case of stress tests. The traces had an uncompressed mean size of 1.7 GB, a maximum size of 13 GB and a median size of 0.93 GB. When compressed, the mean was 110 MB, the maximum 460 MB and the median 85 MB. Although this means a large chunk (limited by how much data can be written before the test times out) of temporary disk space is necessary, the final file size is manageable. Also, most of the executions that were aborted due to incorrect output had traces of only 2–3 MB (uncompressed), since the stress test caused a failure early in the execution. This suggests that trace size is unlikely to be a problem if stress tests are divided into smaller parts to avoid having to store and replay several minutes of correct behaviour that is of little relevance when debugging.

Performing the dependence analysis is often problematic since Atropos constructs the entire DDG in memory. In practice, this means that traces may not be larger than a few megabytes. Again, this can be mitigated by finding test cases in which failures occur and are detected as quickly as possible.

5.1.4 Performance Loss Caused by Instrumentation

Considering only the executions that completed successfully (since the failures were not necessarily in the same place), instrumenting the programs to trace the execution caused the time used by the stress test to increase on average to 10.2 times the original execution time; the median slowdown factor was 5.27. One test was only 25% slower; the Reactor implementation in question is otherwise efficient but creates a new thread for each event, causing an overhead that dwarfs that of the instrumentation. The two worst slowdowns were by factors of 69.5 and 35.4, both with Reactor implementations that allow each thread to feed unlimited amounts of data into the buffer without ever waiting for it to be processed. While there is very little overhead from switching between threads in this type of solution, they run the risk of running out of memory.

While the instrumentation introduces noticeable overhead, much of this is masked by overhead from e.g. creating or switching between threads.

The mean time for the instrumented stress tests was 127 seconds, the median 75.5 seconds, the minimum 44.9 seconds and the maximum 739 seconds. Without instrumentation, the mean was 17.9 seconds, the median 15.0 seconds, the minimum 3.8 seconds and the maximum 59.9 seconds. This means that for most students, the stress test will be completed within a reasonable time despite the instrumentation.

5.1.5 Effect on Failure Occurrence

To evaluate whether the execution tracing prevents failures from manifesting in incorrect programs, I reran the stress test 10 times with and without instrumentation on the Reactor implementations in which a failure caused by a race condition was detected by the test package (without the instrumentation). All three programs exhibited race conditions that consistently caused the stress test to fail both with and without the tracing. One of the programs, a very inefficient implementation, was slow enough that the overhead from the instrumentation consistently caused the test to time out before a failure occurred. This can be remedied by adjusting the timeouts to compensate for the instrumentation overhead.

5.2 Visualising Dynamic Dependence Graphs

When Atropos has replayed the execution of the program and constructed its DDG, it can be explored using the visualisation facilities of Atropos. As the full DDG of a program execution is likely to be very large, only a small portion that the user has explicitly requested is shown. Essentially, the visualisation explains what happened in an operation in terms of previous operations. Once the user has found an operation that does the wrong thing despite being executed at the right time and operating on the right data, he has found code involved in a defect.

5.2.1 Visual Representations

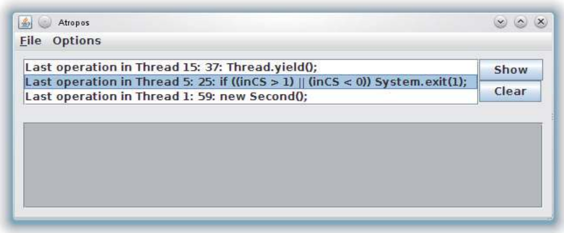
Since a DDG is a graph, one can make use of well-known visual representations for graphs in visualising a DDG. Operations and dependencies must be labelled so that the user can identify them. Directed labelled graphs are often represented as labels (often surrounded by a rectangle or similar container) representing vertices connected by arrows representing edges.

Operations are by default grouped together by lines, as is traditional in debuggers. Each operation is labelled with the number of the line of code and the line itself. The shape of the line of code makes a rectangle a natural choice of container.

In order to show execution order in an intuitive fashion, vertices are arranged chronologically from top to bottom as a *layered graph* [16]. If one operation happened before another, it will be above it. Two vertices being next to each other means that neither is known to have preceded the other. The vertical layout uses space more efficiently than a horizontal one when there are many vertices in a thread and labels are long.

Vertices are horizontally positioned by thread. Above the vertices that belong to the execution of a method call, the name of the method and the object or class on which it was called is shown, together with the arguments to the call. Indentation is used to indicate nesting of method calls.

Dependencies between operations are shown as arrows between vertices. Data produced by one operation and used by another are labelled with the variable name (where applicable) and the value.



```
Last operation in Thread 15: 37: Thread.yield();
Last operation in Thread 5: 25: if ((inCS > 1) || (inCS < 0)) System.exit(1);
Last operation in Thread 1: 59: new Second();
```

Figure 5.2. Atropos showing the ends of the thread executions in a trace of `Second`

5.2.2 Navigating the DDG

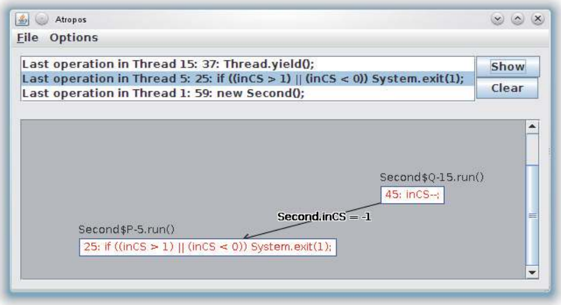
The visualisation uses the termination of the threads in the program as *starting points*. It is assumed that at least one of these corresponds to a symptom. This could be:

- A thread terminating abnormally due to an uncaught exception;
- A thread detecting incorrect behaviour and aborting itself or the whole program; unit tests and other assertions usually behave like this;
- A thread being stuck in a deadlock.

Once the user has opened a trace, the list at the top of the window shows the starting points for the DDG. By choosing the right thread to examine, the user can work backwards from the thread termination to the failure that caused it.

Let us look at one possible execution of the example in Appendix B, `Second`. Opening the trace of this execution in Atropos results in the view in Figure 5.2. For debugging purposes, the interesting operation is the one in thread 5 (highlighted) that terminated the program by calling `System.exit(1)`.

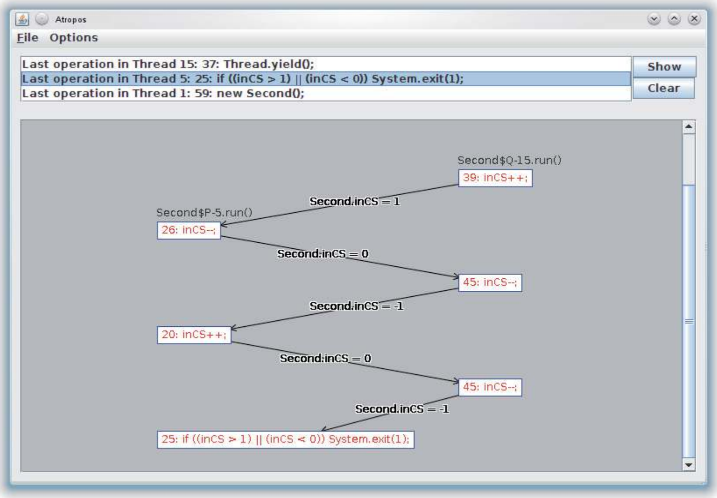
To keep the visible graph manageable, all dependencies of vertices are hidden unless the user requests that they be shown, which may cause more vertices to be shown. To show where a value read by an operation came from, one can choose the relevant value from the list of data sources of the vertex. Similarly, to show where a value written by the operation was used, one can select the value from the list of data uses of the vertex. There are also commands to show all data sources and uses. The last branching operation, such as `if` or `while` can be shown. This is easier to understand



```
Last operation in Thread 15: 37: Thread.yield();
Last operation in Thread 5: 25: if ((inCS > 1) || (inCS < 0)) System.exit(1);
Last operation in Thread 1: 59: new Second();
```

```
Second$Q-15.run()
45: inCS--;
Second.inCS = -1
Second$P-5.run()
25: if ((inCS > 1) || (inCS < 0)) System.exit(1);
```

Figure 5.3. Atropos showing an incorrect value causing the program to abort



```
Last operation in Thread 15: 37: Thread.yield();
Last operation in Thread 5: 25: if ((inCS > 1) || (inCS < 0)) System.exit(1);
Last operation in Thread 1: 59: new Second();
```

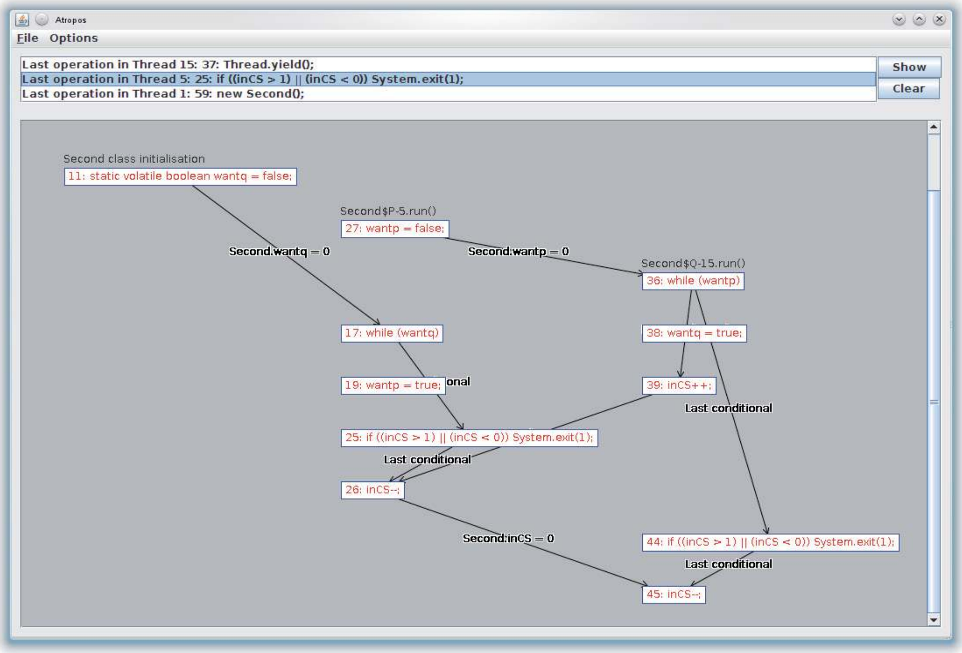
```
Second$Q-15.run()
39: inCS++;
Second.inCS = 1
Second$P-5.run()
26: inCS--;
Second.inCS = 0
45: inCS--;
Second.inCS = -1
20: inCS++;
Second.inCS = 0
45: inCS--;
Second.inCS = -1
25: if ((inCS > 1) || (inCS < 0)) System.exit(1);
```

Figure 5.4. Atropos showing unexpected changes in a counter variable

and calculate than a control dependency.

In the example, it is obvious that the value of inCS is of interest, since the program specifically aborted its execution because the value was out of range. By checking the value of inCS that was read by this condition, we see that it was found to be -1, which is an incorrect value for a count of threads in a critical section. This is shown in Figure 5.3.

The next step is to determine why the value of inCS is incorrect. If inCS was decreased from 0 to -1 when exiting a critical section, the value of inCS must be been incorrect earlier. Using this reasoning, the changes to inCS are examined, revealing a sequence in which an increment of inCS is followed by two decrements; in other words, two threads exiting the critical section after each other without anyone entering it. This is shown in Figure 5.4.



```
Last operation in Thread 15: 37: Thread.yield();
Last operation in Thread 5: 25: if ((inCS > 1) || (inCS < 0)) System.exit(1);
Last operation in Thread 1: 59: new Second();
```

Figure 5.5. Atropos showing failure to uphold mutual exclusion

This raises the question of how the example program got into a state where two threads are in the critical section at once. By checking how the program branched to reach these two successive decrements of `inCS`, one can find the operations where the threads exited their wait loops. By checking which values were read by these operations, one can see that the value of `wantq` that was read was the initial value of `false` and the value of `wantp` was written on exiting the critical section. These values are both correct in the situation where both threads are at the start of their cycles. Indeed, the problem seems to be that the threads do not see the other thread trying to enter the critical section. By examining what happens after the threads decide to enter the critical section, one can see that the threads do not indicate their intent to enter the critical section until the other has already decided to do so; by almost simultaneously checking whether they can enter the critical section, they can both do so at once. The result of these steps is shown in Figure 5.5.

Although this explains the problem with the mutual exclusion in the example, it does not explain why `inCS` became negative. By checking where the different values of `inCS` were used, it can be found that both threads read, incremented and wrote `inCS` at the same time, causing one of the increments to be lost. This is shown in Figure 5.6.

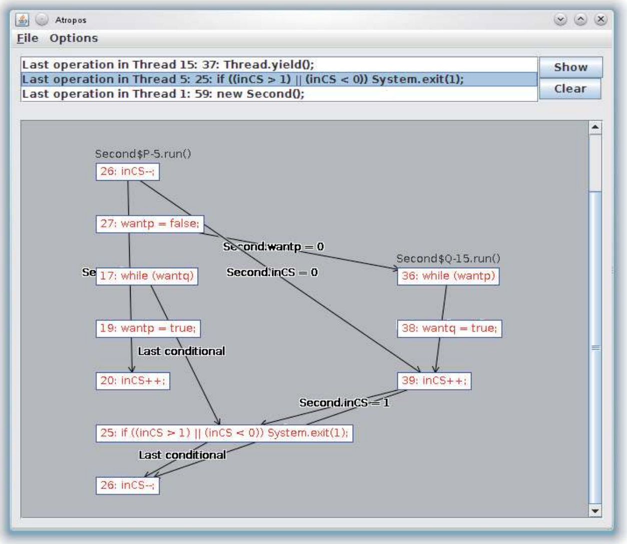
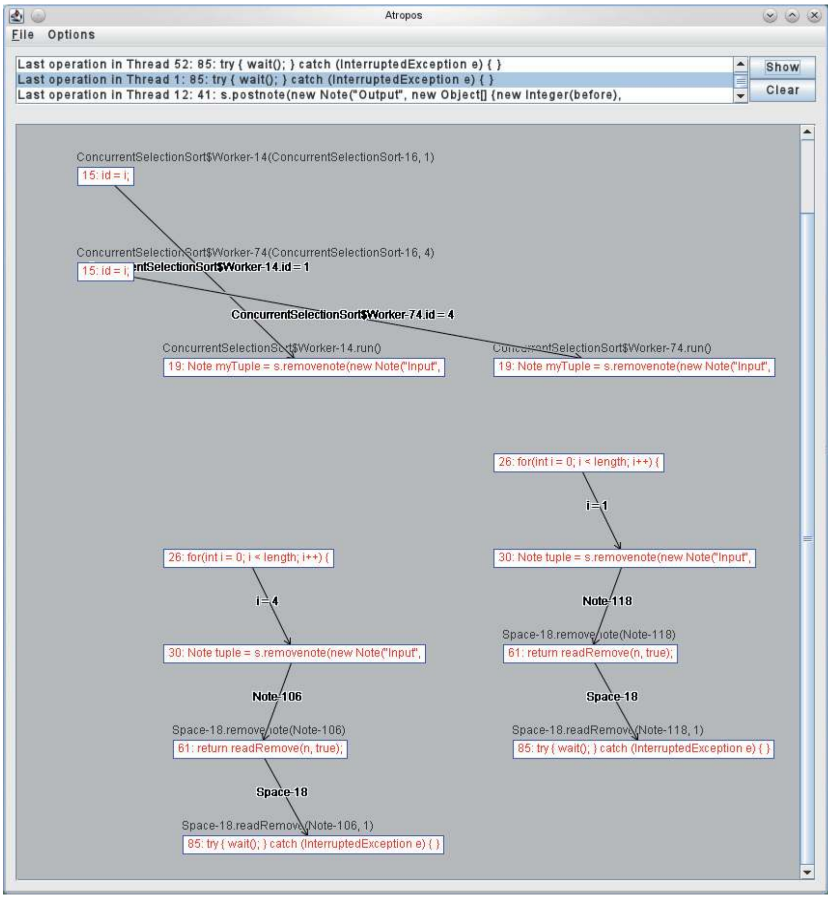

Last operation in Thread 15: 37: Thread.yield();
Last operation in Thread 5: 25: if ((inCS > 1) || (inCS < 0)) System.exit(1);
Last operation in Thread 1: 59: new Second();

Figure 5.6. Atropos showing a race between two increment operations

Since students work with concurrent data structures such as tuple spaces at multiple levels of abstraction, as noted in Subsection 4.1.4, Atropos provides a way for users to raise the level of abstraction and remove implementation details from the visualisation by grouping together lines executed as part of a method call to a single vertex.

Figure 5.7 shows a visualisation of another example trace. The graph shows an execution of the incorrect concurrent sorting program used in the evaluation in Chapter 6 and reproduced in Section A.2. The program contains a circular locking dependency: the input tuple removed from the tuple space on line 19 and returned to it on line 40 may be needed by another thread at line 30.

Here, the place where two threads deadlock has been used as a starting point for exploration. By following the data dependencies backwards (the arrows shown, labelled with the data that was transferred), it has been determined which tuples are being waited for (the index `i` in the `for` loops). The tuples removed at the start of the execution of these two threads are also shown, together with the sources of the indices of the tuples that were removed. To keep the screenshot simple, many of the dependencies explored to find this information have been hidden. The process we expected students to use with Atropos is described further in Subsection 6.3.4.



```
Last operation in Thread 52: 85: try { wait(); } catch (InterruptedException e) { }
Last operation in Thread 1: 85: try { wait(); } catch (InterruptedException e) { }
Last operation in Thread 12: 41: s.postnote(new Note("Output", new Object[] {new Integer(before),
```

```
ConcurrentSelectionSort$Worker-14(ConcurrentSelectionSort-16, 1)
15: id = i;
ConcurrentSelectionSort$Worker-74(ConcurrentSelectionSort-16, 4)
15: id = i;
ConcurrentSelectionSort$Worker-14.id = 1
ConcurrentSelectionSort$Worker-74.id = 4
ConcurrentSelectionSort$Worker-14.run()
19: Note myTuple = s.removenote(new Note("Input",
ConcurrentSelectionSort$Worker-74.run()
19: Note myTuple = s.removenote(new Note("Input",
26: for(int i = 0; i < length; i++) {
i = 1
30: Note tuple = s.removenote(new Note("Input",
Note-118
Space-18.removenote(Note-118)
61: return readRemove(n, true);
Space-18
Space-18.readRemove(Note-118, 1)
85: try { wait(); } catch (InterruptedException e) { }
26: for(int i = 0; i < length; i++) {
i = 4
30: Note tuple = s.removenote(new Note("Input",
Note-106
Space-18.removenote(Note-106)
61: return readRemove(n, true);
Space-18
Space-18.readRemove(Note-106, 1)
85: try { wait(); } catch (InterruptedException e) { }
```

Figure 5.7. Atropos showing a deadlocked sorting algorithm

5.3 Discussion

Simplicity was a key criterion in the design of Atropos. One aspect of this was that it should contain only the DDG visualisation and the features necessary to support a backward debugging strategy. This would allow the DDG visualisation to be evaluated on its own, rather than as one potentially ignored feature among many. Also, keeping the feature set small decreased both development time and the time to learn the system. However, as shown in Chapter 6, using only a DDG is not practical, and it would have been more useful to have complementary visualisations. This is discussed further in Subsection 6.3.4.

One problem with the layered graph layout is that vertices often overlap each other or edge labels. This was addressed by adding the ability to drag vertices to other positions.

The traces collected by Atropos typically contain a very large amount of data values that have been read under data-race free conditions. These values can be reconstructed based on the happens-before order, which means that they can be removed from the trace files. This ought to make the trace files roughly as small as those in data-race-unaware replay systems.

The size of the in-memory representation of the DDG is a much larger scalability problem and one for which no easy solution exists. This can be avoided, in part, by keeping executions short.

6. How Students Make Use of the Visualisation

Once Atropos was implemented and working, evaluating it was clearly the next step. The research questions addressed in this chapter are:

5. How does the visualisation of program executions help students understand and debug defective concurrent programs?

6. How could the visualisation be improved to further help students in these tasks?

Using the answers to these questions, we can answer the question: “In what ways did the visualisation in the previous part assist the students?”

To answer these research questions, a study of how students use Atropos was made. The additional research questions were:

7. What information and understandings do students get when using Atropos?

8. What information are they looking for? In particular, what information do they try to get, but fail?

9. How are they using Atropos? What operations are they using? What is their strategy for finding the information they want?

The evaluation was done during the sessions for the last round of exercises in the 2010 instance of the course, as described in Section 3.2. The tasks in this round are described in Appendix A.

Originally, the study was intended to include a quantitative comparison of student performance using Atropos and not using Atropos. Hence, the students were divided into two groups: A and B. The first and third tasks were intended to evaluate what the students know, while the second and fourth tasks were intended to compare how the students work with and without visualisation. Group A used Atropos in task 2 but not in task 4. Group B used Atropos in task 4 but not in task 2. However, since only 21 students participated in the sessions (in part, due to students dropping out of the course; 79 students registered for the course, but only 37 students took the exam at the end of the course), the quantitative comparison was left out of the study.

Also, even though the sessions were extended from the original two hours to allow students to complete their work (in one case almost up to four hours), none of the pairs completed task 4. Hence, the focus in this study is on a qualitative analysis of task 2, which is the task most relevant to the research questions.

A mixed-method research design [37] that is primarily qualitative was used. Some quantitative analysis was added to support the conclusions of the qualitative analysis.

For more details, especially quotes that illustrate the categories, see Publication VI.

6.1 Data Collection

The conversations of the participating pairs of students and the contents of the screen(s) of the computer(s) they used were recorded using a microphone and video screen capture software. The students were required to work in pairs; the students themselves chose whom to work with. This was done in order to encourage them to verbalise their thoughts and to discuss their plans and approaches to the tasks.

Ten pairs of students participated. Two of the pairs have been left out due to lack of usable recordings of conversations to analyse, caused by technical problems in recording. One of the groups that was left out conversed in French, which would have been problematic to analyse. Four of the remaining pairs of students were in group A and four in group B. The following analysis focuses on the qualitative aspects and on evaluating the visualisation, specifically the second task as performed by the three pairs in group A who worked on task 2 during the sessions: Charles and Ada (46 minutes), Peter and John (45 minutes) and Brian and Dennis (51–53 minutes; these students worked independently on different tasks at some points)[1]. They were all in group A and spoke Finnish.

6.2 Analysis

The analysis was based on determining the operation foci of the students; what they are doing in terms of concrete activities that are part of constructing a solution to their problem. Operation foci are determined by the subgoal students are trying to achieve at a given time.

The first version of the operation foci was created before the data were collected, based on the operation foci described by Yehezkel et al. [81] and the activities of Kiesmüller [39]. During the analysis, the categories were refined to form the categories shown in Table 6.1. The foci are identified by numbers; in the case of subfoci, a letter is appended.

The descriptions of students' operation foci form a large part of the answer to RQs 7, 8 and 9. Also, the operation foci related to Atropos highlight issues that answer RQs 5 and 6. In preparation for analysis, the usable parts of the recordings were transcribed and translated into English. Actions performed on the computers by the students and their results were added to the transcript to aid in the analysis.

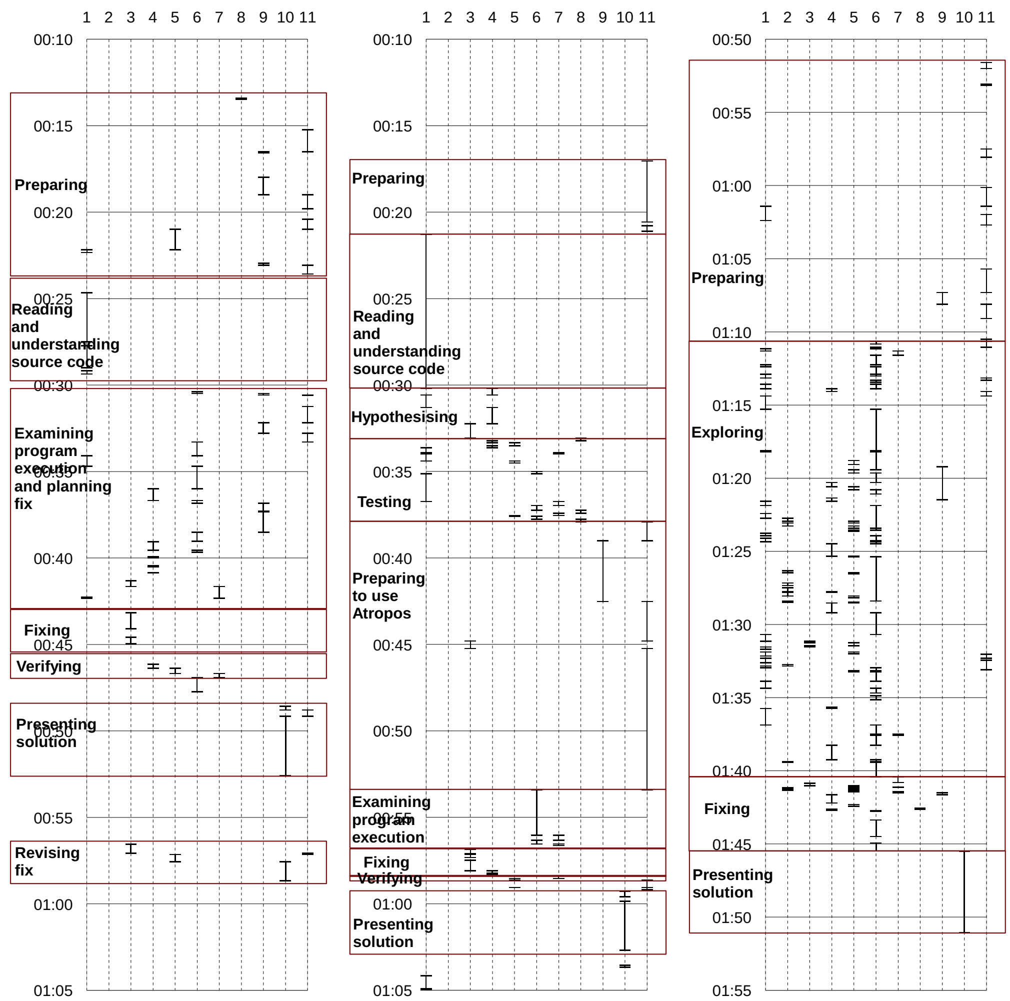
Initially, the analysis was done by attempting to place each statement in one category for each of the above classifications using a spreadsheet. This emphasised individual statements outside their context and gave little support for effectively expressing structure within categories. Hence, we switched to expressing the transcript like a script and forming quotations from contiguous stretches of text. The quotations were linked to codes that were organised into the categories presented below. I did this analysis using ATLAS.ti.

The categories changed somewhat during the analysis based on the data. In particular, the foci that did not fit into the original categories were used to refine the categories. The operation foci and the coding of one group of students were checked by Lauri Malmi and adjusted until a consensus was reached.

[1]The students' names have been changed.

1 Understand Program Code E.g. reading the source code and looking at how parts interrelate **1A Understand the Program to Debug** Understanding the program that the students should debug (in static terms) **1B Understand Other Available Code** Understanding other code involved in the task (e.g. library code, concurrency constructs)
2 Add Debug Code
3 Modify Code **3A Fix Bugs**
4 Formulate Hypotheses Formulate and discuss hypotheses about what a program does and how it differs from the intended behaviour
5 Run the Program
6 Observe Program Behaviour Observe the program behaviour e.g. by reading console output or log files or by looking at the visualisation **6A Explore a Trace** Decide what to look at next in an execution trace, including trying out different commands in Atropos to find something useful **6B Interpret Observations** Discussion of representations of program behaviour and what they mean in terms of the program's execution **6C Unexpected Atropos Behaviour** Discussion of how Atropos does not behave as expected by the students and working to get it to act as expected
7 Determine Correctness Check that a program works correctly **7A Compare Desired and Observed behaviour** **7B Create Test Cases** Plan and set up test cases and executions **7C Analyse a Class of Scenarios Statically** Discuss and examine how the code reacts to a type of situation
8 Determine Goals Any activity related to understanding the desired behaviour of the program
9 Understand Atropos All activities with the goal of understanding Atropos, not directly trying to achieve anything related to the task (e.g. reading Atropos's manual)
10 Present Solution to Assistant
11 Prepare Prepare for other operation focus **11A Create a Trace** Work on and discuss how to create a trace, assuming the test case has already been decided **11B Prepare Code** Prepare code or software for another operation focus **11C Set Up and Start Atropos** Get Atropos running, up to and including loading a trace

Table 6.1. Operation foci



(a) Charles and Ada **(b)** Peter and John **(c)** Brian and Dennis

Figure 6.1. Operation foci over time for each group

6.3 Results

In this section, the results of the analysis are summarised in the form of diagrams of operation foci over time for groups of students (see Figure 6.1). In the diagrams, the Y axis shows time from the start of the recording (in hours and minutes). The X axis is labelled with the number of the operation focus (as shown in Table 6.1). Sections of student activity for which an operation focus has been found are marked as vertical lines with start and end markers. To aid in understanding the diagrams, the students' activities have been divided into parts marked with red boxes labelled with a description of the activities.

6.3.1 What Students Try to Achieve with Atropos

One approach to answering RQ 8 is to look at why students use Atropos; how it fits into their process of solving a debugging task.

Like Isohanni and Knobelsdorf [34] found, some students do not use the provided visualisation tool even when instructed to do so or only use it in a limited fashion. Students seem to use Atropos because they are told to do so, not because they see a need for it. Some of the students did not even start Atropos until after they felt they had identified the bug. Considering that they are clearly trying to **Understand Atropos (9)** for much of the beginning of the task, this suggests that most of the students had never used Atropos before. This is strange, since Atropos was used in one of the tasks in the previous round of exercises. This suggests that Atropos would need to be better integrated with the rest of the course to convince students to use it. Indeed, one of the students explicitly suggested that the course should include at least a mini-lecture on how to apply Atropos effectively. The experiences of Isohanni and Knobelsdorf [34] suggest that this may not be enough, but their experiences may be due to the fact that VIP was not directly connected to debugging strategies. Giving students a visualisation tool that supports a new way of working (for example, a debugging strategy) is probably not sufficient to teach them this way of working unless the visualisation tool explicitly provides guidance on how it can be used. I will return to the question of how the visualisation can guide students in Subsection 6.3.4.

6.3.2 Identifying Incorrect Behaviour

Once the students believe they have fixed a bug, they will hopefully **Determine Correctness (7)** of the program. As noted by Ben-David Kolikant [7], Ben-David Kolikant and Ben-Ari [8], students may not agree with the professional definition of correctness, which is that the program is efficient, legible, documented, modular and always produces the right output irrespective of input and thread interleavings. Some students' behaviour seems to be more consistent with seeing a correct program as one a teaching assistant will accept as correct or simply not, as suggested by Ben-David Kolikant and Ben-Ari [8], considering the possibility that they may have made an error. For example, at the end of the task, Charles and Ada make a change to the program, run it and then decide to present their change to the assistant without discussing whether it is correct at all. In terms of the purposes of programming assignments described in Subsection 4.1.5, this is seeing the programming task in the framework of the university's requirements ("Assignment"). This is reflected in their testing style and in terms of the testing approaches described in Subsection 4.1.6, this suggests an additional category below "Unplanned": "Testing not required". Lack of experience with concurrent programming probably accounts for this behaviour.

Even the students who tested their solution did not verify that their corrected program correctly handled the type of interleaving that caused the incorrect program to fail. However, Brian did run the fixed program several times to confirm it did not deadlock where the incorrect program did.

Students are used to relying on external support such as automated assessment systems for their testing. If there is no consequence to not testing, students do not bother to test. Students can be encouraged to test their programs by making the thoroughness of their testing an evaluation criterion [20].

In a debugging task, some students will attempt to work around or fix a bug without confirming their hypothesis of what the bug is through examination of program execution. Atropos is thus largely irrelevant to their needs. Working around a bug can be prevented by specifically requiring that the corrected program has similar time and space requirements, which would preclude making copies of data or making the concurrent execution sequential.

It is hardly surprising that students have simplistic ways to **Determine Correctness (7)** in these tasks compared to, for example, those described in Subsection 4.1.6, which involved programming assignments in which students were told to explain how they had ensured the correctness of their solution, e.g. by testing. Also, each weekly exercise round was intended to be about two hours of work, while each programming assignment was supposed to be 20 hours. The availability of almost instant feedback from teaching assistants and lack of consequences of submitting an incorrect solution may also have contributed to the general lack of interest in testing. One response to RQ 6 is thus: in order to encourage students to make use of testing and debugging tools, the effort needed to get started must (seem to) be less than other options, such as asking an assistant to look at the program.

6.3.3 Debugging Process

The students' debugging processes, which are relevant to answering RQ 9, are shown in Figure 6.1. The large-scale structure of their processes appears to be sound. One would expect, that after initially having to **Understand Atropos (9)**, **Determine Goals (8)** and **Prepare (11)**, that the students would make sure they **Understand Program Code (1)** and then **Run the Program (5)** in order to **Determine Correctness (7)**. Once the program has been found to fail, the resulting execution would then be used to track down the bug; a process in which the students would **Formulate Hypotheses (4)** based on what they learn when they **Observe Program Behaviour (6)** and **Determine Correctness (7)**. Once they are satisfied they have confirmed their hypothesis of what the defect is, they would **Fix Bugs (3A)** and **Run the Program (5)** and **Observe Program Behaviour (6)** in order to **Determine Correctness (7)** of their corrected program. What the students have done mostly fits this pattern, which would suggest that Atropos would fit well into their large-scale approach to debugging. In line with the results of Ahmadzadeh et al. [2], Brian and Dennis did not take the time to **Understand Program Code (1)** first (see Figure 6.1) and hence had difficulties debugging.

6.3.4 Successful and Unsuccessful Use of Atropos

As a basis for comparison, I will first explain how I expected students to apply Atropos in the debugging task (task 2 in Section A.1):

1. Determine that the program has deadlocked by having all threads waiting for another thread to insert a tuple in the space. This can easily be deduced by checking where the threads' execution blocked.

2. Identify which tuples are being waited for and in which place in the main program. This can be done by determining from where the blocked operations were called. It should be noted that examining the call stack at the time the program deadlocked is also sufficient to collect the information required so far; this can be done using e.g. Eclipse's debugger like Brian did.

3. Identify that the reason why these tuples cannot be found is that they were removed from the space earlier and where this happens.

4. Finally, the problem can be fixed by returning the missing tuples to the space before getting others. This fix can then be verified by running the program and confirming that it still runs correctly even when threads are interleaved in ways that caused the bug to manifest in the original version of the program.

A view of Atropos illustrating the failure is shown in Figure 5.7.

In order to answer RQs 7, 8 and 9, and hence RQ 5 and RQ 6, properly, one must look at how students used and failed to use Atropos. Students were able to use Atropos to extract information to help clear up some misunderstandings of, for example, what threads exist in a running program. However, this information is derived from the list of how threads ended, not from the dependence graph view itself. This happened even though the students used a debugging style that relied heavily on examining source code, trying to reason about it statically and rewriting suspicious parts of the program (cf. [23]). Peter and John spent 11 minutes (Charles and Ada 5 minutes) trying to **Understand Program Code (1)**, but both groups only spent 3 minutes trying to **Observe Program Behaviour (6)**.

Several students were effectively prevented from finding the relevant data in step 2 using Atropos by difficulties caused by the visualisation displaying implementation details of the tuple space as a consequence of following data and control dependencies of the `wait` operations in which the program deadlocked. Dennis managed to group the operations in the tuple space method executions together, but the sheer amount of data dependencies of a method execution put a stop to his progress. Grouping operations by method execution is specifically intended to address the former case, but is apparently not something students can easily discover, especially since the examples of using Atropos did not require this. Ada suggested that a more intuitive operation would be to show the operation that invoked the method execution to which a selected operation belongs. In the latter case, grouping together multiple reads of the same values in the context menu could have made it far easier to navigate. In both cases, one can argue that the problem is that the students are being shown what is happening in the tuple space even though they are interested in how it is used.

To assist students unfamiliar with backward debugging, it would be helpful if Atropos itself provided more explicit guidance on how it can be effectively applied. To help users get started, Atropos could explicitly identify symptoms, such as deadlocks or incorrect behaviour, and recommend these as starting points for backward debugging. The backward debugging process could be supported by providing the option to mark operations as correct or incorrect behaviour, making it easier to see what possible causes of an incorrect operation's behaviour have been explored.

Similarly, to help users examine executions at an appropriate level of abstraction, it would be useful if users could select, before starting to explore a trace, for which classes the execution of methods is shown by default; in other words, which classes' execution the user wants to examine. A third option would be to start exploring a thread from not just the last operation, but all the running method calls in that thread; in other words, all the operations that would be shown in a stack trace.

It would be worthwhile to investigate how to support forward debugging strategies more effectively, perhaps by supporting forward navigation along dependencies to complement the backwards navigation already available.

Another possible approach is to complement the DDG with an overview of the program's execution trace, for example in the form of the tree view of RetroVue [12] or as a list of operations performed by a specific thread, at a specific line of code or on a specific variable. This would enable users to skip to interesting parts of the program execution rather than find a route back along the DDG from a symptom. The lists of operations would also provide an overview of program execution that some students tried to achieve by repeatedly requesting the previous line until they had a list of what a thread had done.

7. Validity

In this chapter, I reflect on validity issues involved in this research, based on the presentation of validity by Johnson and Christensen [35]. Lincoln and Guba [47] use a similar framework as a basis for their own quality criteria.

7.1 Descriptive Validity

The problem of ensuring *descriptive validity*, describing what the researchers saw and heard accurately, is to a great extent avoided in this research by using artefacts and recordings of the students as data rather than notes made by the researcher.

7.1.1 Recordings

The recordings include noise from equipment and the environment, especially in the exercise sessions, where many students were working at the same time. Due to speakers' movements, the microphone was, at worst, several meters from the speaker. Hence, some recordings (especially the exercise session of Charles and Ada) could not be completely transcribed. Technical issues also prevented two recordings from being analysed.

Miscommunication can be caused by requiring people to express themselves in an unfamiliar language. The students were allowed to choose between Finnish, Swedish and English to mitigate this. In most cases, the students used their native Finnish or were otherwise fluent in Finnish. One of the interviewees (labelled Elena) mentioned having difficulty communicating in English.

7.1.2 Code Analysis

It is possible to make use of established practices for software quality assurance to find defects in students' programs, as described in e.g. [11]. Defects may pass unnoticed through testing, code review and other measures to find them. However, combining different techniques is known to increase the chances of finding errors noticeably. Both automated testing and code review were used. Combining both approaches as well as having the classification of each defect checked by two people also helps ensure that defects are correctly identified.

7.1.3 Affecting the Students

On the path from a student's mind to a publication describing his understandings, several steps occur that can introduce misinterpretations. The first is the student's own expression of his understandings; he may be miscommunicating on purpose or communicating incorrectly or unclearly. Johnson and Christensen [35] note that the former can become a problem if the student does not trust the researcher, or if he perceives that the researcher or someone else is pressuring him to answer in a specific way. He may also believe that he can gain something by misrepresenting his own perceptions.

This is particularly problematic if an interviewee does not see the researcher as a neutral party because, for example, the researcher determines the student's grades. However, when studying learning in the context of a university course, it is advantageous for the researcher to be familiar with the subject matter and how it is taught; it is hard to accurately interpret something one does not understand. While I designed the assignment used in this research, my influence on the assessment of the course is small, since I only help the actual teaching assistants as required. In particular, the year the interviews were done, I did not participate in teaching the course.

However, at least one interviewee appeared to perceive the interviewer as one of many assistants on the course (despite being told, like all the other interviewees, several times, both in person and in writing, that the interviewer is not a member of the course staff and the interview will not affect the interviewee's grade on the course) and was reluctant to offer information.

For practical reasons, the code analysis has been done solely on the final version of the code submitted by the students before the deadline, leaving any defects that the students themselves found and corrected out of consideration. Although it would be possible to require students to submit, for example, every version of their program that compiled successfully (like in [71]), this is impractical as students often work on their own computers instead of those provided by the university. Also, the students would have to collect all the different versions of their code; in both cases, the students would be well aware that their code is being collected for analysis. This awareness could then affect their way of working and skew the results.

7.2 Interpretive Validity

Interpretive validity is the accuracy of the interpretation and presentation done by the researcher. Johnson and Christensen [35, 36] note that the obvious way for a researcher to avoid incorrect interpretation is to use descriptions that are as close as possible to the student's own words. Interpretive validity is often ensured by using quotes from the people being studied. The quotes must be presented with sufficient context to allow correct interpretation. This has been applied both in the analysis and reporting in this work. Another way a researcher can improve interpretive validity is to ask the subject whether the researcher's interpretation is (in the subject's opinion) correct. In my interviews, I confirmed my interpretation of some unclear statements made by the interviewees by asking them questions like "Do you mean. . . ?".

Interpretive validity is much more difficult to ensure when the data consists of code and descriptions of its underlying design and the desired result of the interpretive process is, essentially, what the programmer was thinking at the time he wrote the code. In cases where the programmer has explicitly described his reasoning (in comments or in a design report), little interpretation is needed. However, in the data I collected in Autumn 2005, I only found statements from the students for roughly half the observed defects, and even then, they were not always sufficient to pinpoint the exact error the student made other than in general terms (e.g. failure to correctly design a concurrent data structure) [50]. The other half of the errors was classified by generalising from similar errors or forming hypotheses about the underlying errors. The interpretive validity of classification of the latter half of the errors may therefore have been very bad. This led to the additional phenomenographic research described in Chapter 4.

7.3 Theoretical Validity

Theoretical validity is the degree to which the theories the researcher has formed from the data fit the data. In the case of my research, the qualitative theories are mostly explanations for different defects in terms of a chain leading from cause(s) of an error to the error and from there to the defect. The corresponding quantitative theories are quantifications of the qualitative theories: how common are the defects, errors and causes, and what are the probabilities of one leading to another?

The strategies suggested by Johnson and Christensen [35] to ensure theoretical validity are *extended fieldwork*, which in my case means studying the same assignments over several different years, *theory triangulation* (using multiple theories and perspectives), *pattern matching* (checking whether complex or unique predictions hold in the data) and *peer review*. A variant of pattern matching that is natural when possible problems with the teaching or assignments in a university course are involved is to change the teaching or assignments to avoid the problem the theory states and see whether the students' performance changes as the theory predicts. Extraneous variables, such as other changes to the course or participating students, are likely to complicate this assessment.

7.4 Researcher Bias

Researcher bias is a researcher finding what he wants to or expects to find and is a threat to the *confirmability* of Lincoln and Guba [47] (their equivalent of objectivity). Bias can be divided into finding things that are untrue and not finding things that are true (and relevant). The former is particularly a risk when interpreting results (and is addressed in that section) and can be addressed through mechanisms to ensure that results are correct. The latter is more problematic, as it involves ignoring parts of the data or possible theories. The solutions suggested by Johnson and Christensen [35] are *reflexivity* (the researcher tries to determine what his (potential) biases are and minimise their effects) and *negative-case sampling* (trying to find the unexpected). The fact that my results include the unexpected result that half the students' errors on the Autumn 2005 course were due to misunderstanding the requirements of the assignment has led to negative-case sampling in the sense that the initially unexpected case has become the expected case.

Bias in the analysis of students' programs was mitigated by making use of the students' own comments, explanations and solutions as well as the results of the phenomenographic study to form subgoals and plans.

7.5 Internal Validity and Credibility

Internal validity is the degree to which the researcher is justified in saying that an observed relationship is causal. This is not usually considered relevant to phenomenographic research, where it is replaced with *credibility*: whether the findings are believable to the people they describe [47]. However, similar approaches can be used to ensure both.

Internal validity and credibility can be improved through the use of multiple methods or data sources. In my case, the multiple methods are collecting data on a large (roughly 50–100 for each assignment) number of students in the form of both code and reports describing the way they reasoned as well as using interviews and video recording to collect supplementary information directly from a small number of students. The multiple data sources are students from different instances of the same course. It would probably be beneficial to study concurrent programming outside the course studied here; this is discussed further in the following section.

Internal validity may be threatened by a wide range of issues [35]. In my research, the most severe threats appear to be related to the lack of a proper control group. For example, improvements in student's solutions for the assignments may be due to e.g. changes to the lectures or course literature, differences in scheduling, and a number of changes to the assignments. Each change to the course is an extraneous variable when attempting to determine the effect of another change. Arranging many different versions of the same course that differ only in one way is highly impractical. However, throughout the time the students' programs and reports were collected, the lecturer has been the same, the contents of the course mostly the same and roughly the same assignments have been used. Also, it is somewhat implausible that other factors would result in precisely the changes in students' work that were observed as consequences of the changes to the teaching and assignments described in Section 4.2.

7.6 External Validity, Transferability and Dependability

External validity is important if results are to be useful outside the research setting. One approach is to describe the setting in detail and allow the reader to compare his own situation to yours and determine whether your research is applicable (*naturalistic generalisation*). This is the approach I have used here. This corresponds to the *transferability* of Lincoln and Guba [47].

Replication is also used to ensure external validity [35] and *dependability* (the equivalent to reliability) [47]; in my case, I have studied the same assignments in different annual instances of the same course, allowing me to assess how well the results generalise between different years. Replicating the experiment in another university would give a better idea of whether the results generalise outside our concurrent programming course. This applies both to quantitative and qualitative results.

Transferability is ensured by *purposive sampling*: choosing people to examine or interview in such a way that the widest range of information is collected [47]. How I did this is explained in Section 4.1.

7.7 Construct validity

Construct validity is the degree to which the characteristic or *construct* under examination is accurately represented in the study; in other words, how well the characteristic was operationalised. This is not relevant to the phenomenographic study in Section 4.1. However, the interpretation of the results on students' defects in Section 4.2 is affected by questions of construct validity, unless one takes the position that the study is specifically intended to measure the defects in students' solutions to these particular assignments, not concurrent programs in general as stated in the research questions of Chapter 4.

The research questions for the defect analysis refer to all the concurrent programs that will ever be written by each student. Since answering this directly would almost certainly require data collection over an unacceptably long time, the question must be operationalised. A way is needed to estimate a student's future performance in as yet unknown programming tasks; this is the same problem that teachers face when assessing the skills of their students. The programming assignments described here are designed for this purpose and include the concurrency-related aspects of different real-world programming tasks. The assignments, due to limits on the time students can spend on them, are quite simple compared to many real-life problems. Hence, there is less potential for incorrect interaction with other parts of a program. Also, the specifications of the assignments force students to use a specific design; in real life, they may have the option to avoid many concurrency-related problems by making design choices such as avoiding concurrency entirely or using different concurrency mechanisms.

Using multiple operationalisations of a construct is a commonly used approach to improving construct validity. The three different assignments that have been used can be considered different operationalisations of the desired construct. An individual assignment does not cover, for example, all concurrency mechanisms. Together, the assignments cover many different aspects of concurrent programming.

By comparing the results of the defect analyses of the different assignments and different versions of the same assignments, one can draw some useful conclusions about the effect of the choice of assignments on the results. It is clear that requirement-related defects are strongly affected by how the assignment is explained to students and that providing tests helps students detect certain types of defect. The Trains assignment also shows how other issues specific to an assignment can account for many defects.

There are, however, several types of defect that occur in many different assignments, such as incorrect memory management and incorrectly interacting parts of programs. This would suggest that these defects are likely to show up in other concurrent programming assignments as well.

8. Conclusions

In this chapter I summarise the results of the thesis in terms of the questions posed in Section 1.2 and the contributions made. I also present possible directions for future work.

8.1 Understanding and Debugging Needs of the Students

The first question was: "What needs do students have with regard to understanding and debugging concurrent programs?". In Chapter 4, I presented a mostly phenomenographic analysis of how students understand and approach concurrent programming and an analysis of what students do wrong in their concurrent programs.

Students were found to think about the concurrency constructs they implemented at several levels of abstraction; any tools provided to the students should allow students to work at these different levels of abstraction. The students' programs initially contained large amounts of requirement-related defects that got in the way of learning from the assignment, many of which were eliminated by communicating requirements more clearly, especially in the form of packages of unit tests. The remaining defects largely manifested themselves nondeterministically and due to incorrect algorithms or implementation; these were indicative of difficulties the students had with concurrent programming. Many of these were cases of solutions to subproblems conflicting with each other.

8.2 Addressing the Students' Needs through Visualisation

To answer the question "How can the needs from the previous part best be addressed through visualisation?", I designed a visualisation tool based on interactive exploration of dynamic dependence graphs, presented in

Chapter 5, to help students understand the execution of concurrent Java programs and debug them. This tentatively answered the question.

Atropos introduces several improvements on the previous state of the art. As described in Section 5.1, Atropos can, unlike previous systems, produce execution traces and hence reconstruct executions of concurrent Java programs that include data races, making it an improvement on other Java replay systems described in Subsection 2.4.1. Atropos can also produce and visualise dynamic dependence graphs of these programs, distinguishing it from the previous DDG-based debugger, the Whyline [43].

The development of Atropos can also be seen as an example of the development of a software visualisation based on empirical studies of its users and their needs, which Hundhausen et al. [32] argue would be a useful approach to take in SV research. In particular, the design of Atropos is based on empirical data on the defects in the target audience's programs, which is an uncommon approach but similar to the work of Ko and Myers [42].

8.3 Evaluating the Visualisation

Determining whether the students' needs actually were addressed was done in Chapter 6, in which I asked "In what ways did the visualisation tool in the previous part assist the students?" and how the tool could be improved.

This was done through a mixed-method-based evaluation, inspired by similar studies by Yehezkel et al. [81] and Kiesmüller [39], that sheds light on how students make use of the visualisation tool and pinpoints the strengths and weaknesses of the current design. Evaluation studies of this type seem to be common in the CS education research community, as noted in Section 2.5, but uncommon otherwise (no evaluation studies of the debuggers mentioned in Subsection 2.4.1 appear to have been done).

While it seems that the visual representation used by Atropos can be interpreted by students without undue effort, navigating through a program is still difficult. While the dependence graph visualisation would seem to be useful for understanding short causal chains at the level of proficiency with the visualisation our students showed, it is necessary to develop ways to support students in navigating the graph before they can make full use of Atropos.

I suggest several approaches to helping students navigate execution traces. One is to complement the DDG with an overview of the program's execution trace such as the tree view of RetroVue [12]. Another is to provide the ability to examine all operations done on a particular variable. Both approaches would provide an overview of program execution that some students tried to achieve by repeatedly requesting the previous line or branch. A third approach would be the ability to examine the data structures in the program at a certain time as in most debuggers. This would enable the user to check, for example, whether several values form a consistent state, which is hard to do in Atropos. Currently Atropos assumes that all symptoms are at the end of execution: the program terminates or hangs; allowing output operations to be used as starting points would remove the need to add explicit assertions on all output to get a useful starting point. Finally, providing explicit guidance on how to apply a slicing debugging strategy would help students learn this strategy and how to apply it in Atropos.

All of these approaches are straight-forward to implement since Atropos already has all the relevant information available to its current visualisation, suggesting that Atropos could easily be made much more easy to use for students, which would allow them to make full use of its potential.

8.4 Implications for Teaching

The results presented in Subsection 6.3.1 support the result in Subsection 4.1.5 that students primarily see programming assignments as a task to be completed to get a grade and that they will avoid any tasks they see as extraneous such as testing their program or examining its behaviour in detail. Hence, if a teacher wants students to test their programs or examine the behaviour of a program in detail, this must be made an explicit part of the task with clearly defined requirements.

Like Isohanni and Knobelsdorf [34] noted, students seem to avoid using visualisation even when instructed to do so. Instead, they prefer reasoning about the program statically. Simply giving students exercises in which to use a visualisation tool is therefore not enough to get them to use it. To avoid this, I recommend integrating the use of a visualisation tool into lectures so that it is familiar to the students when they are expected to use it.

As noted in Section 4.2, communicating the requirements of assignments to students is difficult. Students may not realise that they have misunderstood or missed a requirement unless they are given a representation of the requirements in an unambiguous and easily applied form such as a set of unit tests.

8.5 Future Work

The improvements to the visualisation of Atropos suggested in Section 8.3 should improve the usability of the tool, allowing students to find what they are looking for more easily. It would then be appropriate to perform a similar evaluation on the improved Atropos, but with a larger amount of students and tasks to allow quantitative comparisons of e.g. students' debugging time and effectiveness.

There are several other ways in which Atropos could be improved. Atropos uses a lot of memory for its dependence graph representation. This could be improved by e.g. performing the dependence analysis or storing its results using more coarse-grained operations than bytecode operations. Also, storing the entire dependence graph in RAM may not be necessary; reconstructing parts of it as needed or storing some of it on disk could significantly decrease the memory requirements of Atropos. The execution trace files could be made smaller by removing variable values in cases where no data races occurred.

Atropos could be adapted for other contexts such as distributed systems by piggybacking vector clock information onto messages in a similar fashion to what Atropos currently does for shared variables and locks. Model checker counterexamples, especially from Java PathFinder, would be an obvious alternative source of execution traces to visualise. Model checker counterexamples are likely to be much shorter than stress test execution traces, which would make them easier to process, navigate and understand.

The effects on learning of using visualisation tools such as Atropos are also a possible subject for future work. In particular, it would be interesting to examine how students' understandings of concurrency change as they use the tool and whether use of visualisation has any effects on post-course skills. This would help establish how the visualisation tools aid in learning in practice.

Bibliography

[1] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. Dynamic slicing in the presence of unconstrained pointers. In *TAV4: Proceedings of the symposium on Testing, analysis, and verification*, pages 60–73, New York, NY, USA, 1991. ACM. ISBN 0-89791-449-X. doi: http://doi.acm.org/10.1145/120807.120813.

[2] Marzieh Ahmadzadeh, Dave Elliman, and Colin Higgins. An analysis of patterns of debugging among novice computer science students. In *ITiCSE '05: Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, pages 84–88, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-024-8. doi: http://doi.acm.org/10.1145/1067445.1067472.

[3] Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, 2000.

[4] Cyrille Artho and Klaus Havelund. Applying Jlint to space exploration software. In Bernhard Steffen and Giorgio Levi, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *Lecture Notes in Computer Science*, pages 297–308. Springer Berlin / Heidelberg, 2004. ISBN 978-3-540-20803-7. URL `http://dx.doi.org/10.1007/978-3-540-24622-0_24`.

[5] Mordechai Ben-Ari. *Principles of Concurrent and Distributed Programming*. Pearson Education, second edition, 2006.

[6] Yifat Ben-David Kolikant. Learning concurrency as an entry point to the community of computer science practitioners. *Journal of Computers in Mathematics and Science Teaching*, 23(1):21–46, 2004.

[7] Yifat Ben-David Kolikant. Students' alternative standards for correctness. In *The Proceedings of the First International Computing Education Research Workshop*, pages 37–46, 2005.

[8] Yifat Ben-David Kolikant and Mordechai Ben-Ari. Fertile zones of cultural encounter in computer science education. *Journal of the Learning Sciences*, 17(1):1–32, January 2008.

[9] Anders Berglund. Phenomenography as a way to research learning in computing. *Bulletin of Applied Computing and Information Technology*, 4(1), July 2006.

[10] Anders Berglund, Ilona Box, Anna Eckerdal, Raymond Lister, and Arnold Pears. Learning educational research methods through collaborative research: the PhICER initiative. In Simon and Margaret Hamilton, editors, *Proc. Tenth Australasian Computing Education Conference (ACE 2008)*, volume 78 of *Conferences in Research and Practice in Information Technology*, pages 35–42, Wollongong, NSW, Australia, 2008. Australian Computer Society.

[11] Ilene Burnstein. *Practical Software Testing*. Springer, 2003.

[12] John Callaway. Visualization of threads in a running Java program. Master's thesis, University of California, June 2002.

[13] Bernard Carré, Jon Garnsworthy, and William Marsh. SPARK — a safety-related Ada subset. In *Ada in transition: Proceedings of the 1992 Ada UK International Conference*, pages 31–45, London, UK, October 1992. IOS Press.

[14] Jong-Deok Choi, Bowen Alpern, Ton Ngo, Manu Sridharan, and John Vlissides. A perturbation-free replay platform for cross-optimized multithreaded applications. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium*, San Fransisco, USA, April 2001. IEEE Computer Society.

[15] Markus Dahm. Byte code engineering with the BCEL API. Technical Report B-17-98, Institut für Informatik, Freie Universität Berlin, April 2001.

[16] G. Di Battista, P. Eades, R. Tamassia, and I.G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, Upper Saddle River, NJ, 1999.

[17] Edsger W. Dijkstra. Cooperating sequential processes. circulated privately, 1965. URL http://www.cs.utexas.edu/~EWD/ewd01xx/EWD123.PDF.

[18] Edsger W. Dijkstra. Over seinpalen. circulated privately, 1965. URL http://www.cs.utexas.edu/users/EWD/ewd00xx/EWD74.PDF.

[19] Thomas Dy and Ma. Mercedes Rodrigo. A detector for non-literal Java errors. In Carsten Schulte and Jarkko Suhonen, editors, *Proceedings of the 10th Koli Calling International Conference on Computing Education Research (Koli Calling 2010)*, pages 118–122, Koli, Finland, October 2010. ACM.

[20] Stephen H. Edwards. Improving student performance by evaluating how well students test their own programs. *Journal on Educational Resources in Computing*, 3(3):1–24, 2003.

[21] Marc Eisenstadt. My hairiest bug war stories. *Communications of the ACM*, 40(4):30–37, 1997. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/248448.248456.

[22] Eitan Farchi, Yarden Nir, and Shmuel Ur. Concurrent bug patterns and how to test them. In *Proceedings of 2003 IEEE International Symposium on Parallel & Distributed Processing (IPDPS 2003)*. IEEE, April 2003.

[23] Sue Fitzgerald, Gary Lewandowski, Renée McCauley, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. Debugging: Finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education*, 18(2):93–116, June 2008.

[24] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[25] Andy Georges, Mark Christiaens, Michiel Ronsse, and Koen De Bosschere. JaRec: a portable record/replay environment for multi-threaded Java applications. *Software — Practice and Experience*, 34(6):523–547, May 2004.

[26] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006.

[27] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Sun Microsystems, third edition, 2005.

[28] Linda Grandell, Mia Peltomäki, and Tapio Salakoski. High school programming — a beyond-syntax analysis of novice programmers' difficulties. In *Proceedings of the Koli Calling 2005 Conference on Computer Science Education*, pages 17–24, 2005.

[29] GroboUtils Project. GroboUtils home page. `http://groboutils.sourceforge.net/`, 2003.

[30] Gerard Holzmann. The model checker Spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.

[31] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Notices*, 39(12):92–106, December 2004. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1052883.1052895.

[32] Christopher D. Hundhausen, Sarah A. Douglas, and John T. Stasko. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages and Computing*, 13(3):259–290, June 2002.

[33] IEEE. IEEE standard glossary of software engineering terminology. Std 610.12-1990, IEEE, 1990.

[34] Essi Isohanni and Maria Knobelsdorf. Behind the curtain: Students' use of VIP after class. In *ICER '10: Proceedings of the Sixth International Workshop on Computing Education Research*, pages 87–95, Aarhus, Denmark, August 2010. ACM.

[35] Burke Johnson and Larry Christensen. *Educational Research: Quantitative, Qualitative and Mixed Approaches*. Pearson Education Inc., second edition, 2004.

[36] R. Burke Johnson. Examining the validity structure of qualitative research. *Education*, 118(2):282–292, 1997.

[37] R. Burke Johnson and Anthony J. Onwuegbuzie. Mixed methods research: A research paradigm whose time has come. *Educational Researcher*, 33(7): 14–26, 2004.

[38] JUnit. JUnit. `http://junit.sourceforge.net/`.

[39] Ulrich Kiesmüller. Diagnosing learners' problem-solving strategies using learning environments with algorithmic problems in secondary education. *Trans. Comput. Educ.*, 9(3):1–26, 2009. doi: http://doi.acm.org/10.1145/1594399.1594402.

[40] Donald Ervin Knuth. The errors of TeX. *Software — Practice and Experience*, 19(7):607–685, 1989.

[41] Andrew J. Ko and Brad A. Myers. Designing the Whyline: a debugging interface for asking questions about program behavior. In *CHI '04: Proceedings of the 2004 conference on Human factors in computing systems*, pages 151–158. ACM Press, 2004. ISBN 1-58113-702-8. doi: http://doi.acm.org/10.1145/985692.985712.

[42] Andrew J. Ko and Brad A. Myers. A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages & Computing*, 16(1-2):41–84, 2005.

[43] Andrew J. Ko and Brad A. Myers. Debugging reinvented: Asking and answering why and why not questions about program behavior. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*, pages 301–310, Leipzig, Germany, May 2008. ACM.

[44] Eileen Kraemer. Visualizing concurrent programs. In *Software Visualization: Programming as a Multimedia Experience*, chapter 17, pages 237–256. MIT Press, Cambridge, MA, 1998. ISBN 0-262-19395-7.

[45] Raimondas Lencevicius, Urs Hölzle, and Ambuj K. Singh. Dynamic query-based debugging. Technical Report TRCS 98-34, University of California, December 1998.

[46] Bil Lewis. Debugging backwards in time. In Michiel Ronsse, editor, *Proceedings of the Fifth International Workshop on Automated Debugging*, Ghent, Belgium, September 2003.

[47] Yvonna S. Lincoln and Egon G. Guba. *Naturalistic Inquiry*. Sage Publications, 1985.

[48] Klaus-Peter Löhr and André Vratislavsky. JAN - Java animation for program understanding. In *2003 IEEE Symposium on Human Centric Computing Languages and Environments (HCC 2003)*, pages 67–75, October 2003.

[49] Jan Lönnberg. Visual testing of software. Master's thesis, Helsinki University of Technology, October 2003.

[50] Jan Lönnberg. Student errors in concurrent programming assignments. In Anders Berglund and Mattias Wiggberg, editors, *Proceedings of the 6th Baltic Sea Conference on Computing Education Research, Koli Calling 2006*, pages 145–146, Uppsala, Sweden, 2007. Uppsala University.

[51] Jan Lönnberg. *Understanding students' errors in concurrent programming*. Licentiate's thesis, Helsinki University of Technology, 2009.

[52] Jan Lönnberg, Ari Korhonen, and Lauri Malmi. MVT — a system for visual testing of software. In *Proceedings of the Working Conference on Advanced Visual Interfaces (AVI'04)*, pages 385–388, May 2004.

[53] Jeff Magee and Jeff Kramer. *Concurrency: State Models and Java Programs*. Wiley, 2006.

[54] Ference Marton. Phenomenography — describing conceptions of the world around us. *Instructional science*, 10(2):177–200, 1981.

[55] Ference Marton and Shirley Booth. *Learning and Awareness*. Lawrence Erlbaum Associates, 1997.

[56] Friedemann Mattern. Virtual time and global states of distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226, Chateau de Bonas, France, October 1988. Elsevier.

[57] Renée McCauley, Sue Fitzgerald, Gary Lewandowski, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. Debugging: A review of the literature from an educational perspective. *Computer Science Education*, 18 (2):67–92, 2008.

[58] Katarina Mehner. JaVis: A UML-based visualization and debugging environment for concurrent Java programs. In Stephan Diehl, editor, *Software Visualization*, pages 163–175, Dagstuhl Castle, Germany, 2002. Springer-Verlag.

[59] Robert Charles Metzger. *Debugging by Thinking*. Elsevier, 2004.

[60] Paul Mulholland. A principled approach to the evaluation of SV: A case study in Prolog. In M. Brown, J. Domingue, B. Price, and J. Stasko, editors, *Software Visualization: Programming as a Multimedia Experience*, chapter 29, pages 439–451. The MIT Press, Cambridge, MA, 1998.

[61] Laurie Murphy, Gary Lewandowski, Renée McCauley, Beth Simon, Lynda Thomas, and Carol Zander. Debugging: the good, the bad, and the quirky – a qualitative analysis of novices' strategies. In *Proceedings of the 39th SIGCSE technical symposium on Computer science education*, SIGCSE '08, pages 163–167, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-799-5. doi: http://doi.acm.org/10.1145/1352135.1352191.

[62] Rainer Oechsle and Thomas Schmitt. JAVAVIS: Automatic program visualization with object and sequence diagrams using the Java Debug Interface (JDI). In Stephan Diehl, editor, *Software Visualization*, pages 176–190, Dagstuhl Castle, Germany, 2002. Springer-Verlag.

[63] Blaine A. Price, Ronald M. Baecker, and Ian S. Small. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4(3): 211–266, 1993.

[64] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: A flexible framework for creating software model checkers. In *Proceedings of Testing: Academic & Industrial Conference — Practice And Research Techniques*, June 2006.

[65] Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster. A comparison of bug finding tools for Java. In *ISSRE '04: Proceedings of the 15th International Symposium on Software Reliability Engineering*, pages 245–256, Washington, DC, USA, 2004. IEEE Computer Society. doi: http://dx.doi.org/10.1109/ISSRE.2004.1.

[66] Douglas C. Schmidt. Reactor: An object behavioral pattern for concurrent event demultiplexing and dispatching. In James O. Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Program Design*. Addison-Wesley, 1995.

[67] Viktor Schuppan, Marcel Baur, and Armin Biere. JVM independent replay in Java. In *Proceedings of the Fourth Workshop on Runtime Verification (RV 2004)*, volume 113 of *Electronic Notes in Theoretical Computer Science*, pages 85–104. Elsevier, January 2005.

[68] Koushik Sen. Concolic testing. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated Software Engineering*, ASE '07, pages 571–572, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-882-4. doi: http://doi.acm.org/10.1145/1321631.1321746.

[69] Judy Sheard, Simon, Margaret Hamilton, and Jan Lönnberg. Analysis of research into the teaching and learning of programming. In *ICER '09: Proceedings of the fifth International Computing Education Research Workshop*, pages 93–104, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-615-1. doi: http://doi.acm.org/10.1145/1584322.1584334.

[70] Juha Sorva. The Same But Different — Students' Understandings of Primitive and Object Variables. In Arnold Pears and Lauri Malmi, editors, *The 8th Koli Calling International Conference on Computing Education Research*, Koli Calling '08, pages 5–15. Uppsala University, 2008.

[71] James C. Spohrer and Elliot Soloway. Novice mistakes: are the folk wisdoms correct? *Communications of the ACM*, 29(7):624–632, 1986. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/6138.6145.

[72] James C. Spohrer, Elliot Soloway, and Edgar Pope. A goal/plan analysis of buggy Pascal programs. *Human-Computer Interaction*, 1:163–207, 1985.

[73] Thomas G. Stockham and Jack B. Dennis. FLIT — Flexowriter Interrogation Tape: A symbolic utility program for the TX-0. Memo 5001-23, MIT, July 1960.

[74] Scott D. Stoller. Testing concurrent Java programs using randomized scheduling. In *Proceedings of Second Workshop on Runtime Verification (RV)*, volume 70(4) of *Electronic Notes in Theoretical Computer Science*. Elsevier, July 2002.

[75] Sun Microsystems Inc. Java Platform Debugger Architecture. `http://java.sun.com/javase/technologies/core/toolsapis/jpda/`, 2010.

[76] Keith Trigwell. A phenomenographic interview on phenomenography. In J. Bowden and E. Walsh, editors, *Phenomenography*, pages 62–82. RMIT University Press, 2000.

[77] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232, April 2003.

[78] Anneliese von Mayrhauser and A. Marie Vans. Program understanding behavior during debugging of large scale software. In *ESP '97: Papers presented at the seventh workshop on Empirical studies of programmers*, pages 157–179, New York, NY, USA, 1997. ACM Press. doi: http://doi.acm.org/10.1145/266399.266414.

[79] Mark Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[80] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.

[81] Cecile Yehezkel, Mordechai Ben-Ari, and Tommy Dreyfus. The contribution of visualization to learning computer architecture. *Computer Science Education*, 17(2):117 – 127, June 2007.

[82] Andreas Zeller. Animating data structures in DDD. In *The proceedings of the First Program Visualization Workshop – PVW 2000*, pages 69–78, Porvoo, Finland, 2001. University of Joensuu.

A. Assignment Used in Visualisation Evaluation

The task descriptions from weekly exercise 5 of the 2010 instance of T-106.5600 Concurrent Programming are reproduced in Section A.1. File and directory names are references to files accessible on Aalto University Unix systems. References to "Ben-Ari" are exercise numbers from [5]. Section A.2 contains the program used in task 2.

A.1 Tasks

Task 1 (Ben-Ari, 9.1) Implement a general semaphore using just one tuple (P and V operations are sufficient).

Task 2 The concurrent sorting program in `/u/35/jlonnber/shared/ConcurrentSelectionSort.java` does not work. Explain a scenario in which it fails (in terms of where the defect is and how this causes the program to fail to achieve its goal) and fix the problem.

Task 3 (Ben-Ari, 9.4) Implement a bounded buffer using a tuple space.

Task 4 The matrix multiplication program in `/u/35/jlonnber/shared/MM.java` does not work. Explain a scenario in which it fails (in terms of where the defect is and how this causes the program to fail to achieve its goal) and fix the problem.

A.2 `ConcurrentSelectionSort.java`

```java
public class ConcurrentSelectionSort {
    final Space s = new Space();
    final int length;

    public ConcurrentSelectionSort(String[] l) {
        for(int i = 0; i < l.length; i++)
            s.postnote(new Note("Input", new Object[] {new Integer(i), l[i]}));
        length = l.length;
    }

    public class Worker extends Thread {
        final int id;

        public Worker(int i) {
            id = i;
        }

        public void run() {
            Note myTuple = s.removenote(new Note("Input",
                                                  new Object[] {new Integer(id),
                                                                null}));
            String myValue = (String)myTuple.p[1];

            /* Count amount of preceding values. */
            int before = 0;
            for(int i = 0; i < length; i++) {
                if (i == id)
                    continue;

                Note tuple = s.removenote(new Note("Input",
                                                    new Object[] {new Integer(i),
                                                                  null}));
                String value = (String)tuple.p[1];

                if (value.compareTo(myValue) < 0 ||
                    (value.compareTo(myValue) == 0 && i < id))
                    before++;
                s.postnote(tuple);
            }
            s.postnote(myTuple);
            s.postnote(new Note("Output", new Object[] {new Integer(before),
                                                         myValue}));
        }
    }

    public void run() {
        for(int i = 0; i < length; i++)
            new Worker(i).start();

        for(int i = 0; i < length; i++) {
            Note out = s.removenote(new Note("Output",
                                              new Object[] {new Integer(i), null}));
            System.out.println(out.p[1]);
        }
    }

    public static final String[] TEST_INPUT = {
        "foo", "bar", "zoq", "fot", "pik"
    };

    public static void main(String[] str) {
        new ConcurrentSelectionSort(TEST_INPUT).run();
    }
}
```

B. `Second.java`

This program, an incorrect implementation of mutual exclusion, is adapted from the second attempt described by Dijkstra [17] leading up to Dekker's algorithm for mutual exclusion. Classes `P` and `Q` are identical except that references to `p` and `q` have been reversed. The shared variable `inCS` is used to keep track of how many threads are in the critical section. If the critical section works correctly, this number should always be `0` (when neither thread is in the critical section) or `1` (when either thread is in the critical section).

```java
/* http://www.pearsoned.co.uk/HigherEducation/
     Booksby/Ben-Ari/ */
/* Second attempt; Modified to exit if critical section
   counter shows something other than 0 or 1. */
class Second {
 /* Number of processes currently in critical section */
 static volatile int inCS = 0;
 /* Process p wants to enter critical section */
 static volatile boolean wantp = false;
 /* Process q wants to enter critical section */
 static volatile boolean wantq = false;

 class P extends Thread {
  public void run() {
   while (true) {
    /* Non-critical section */
    while (wantq)
     Thread.yield();
    wantp = true;
    inCS++;
    Thread.yield();
    /* Critical section */
    System.out.println("Processes in critical section: "
            + inCS);
        if ((inCS > 1) || (inCS < 0)) System.exit(1);
        inCS--;
        wantp = false;
      }
    }
  }

  class Q extends Thread {
    public void run() {
      while (true) {
        /* Non-critical section */
        while (wantp)
          Thread.yield();
        wantq = true;
        inCS++;
        Thread.yield();
        /* Critical section */
        System.out.println("Processes␣in␣critical␣section:␣"
            + inCS);
        if ((inCS > 1) || (inCS < 0)) System.exit(1);
        inCS--;
        wantq = false;
      }
    }
  }

  Second() {
    Thread p = new P();
    Thread q = new Q();
    p.start();
    q.start();
  }

  public static void main(String[] args) {
    new Second();
  }
}
```

Errata

Publication I

In Subsection 5.2, four of the twelve groups selected for interview were pairs, not three.

Publication IV

In Subsection 3.3, ‘failure’ should be ‘symptom’.

Publication VI

In Subsection 5.4, ‘failure’ should be ‘symptom’.

Aalto-DD 24/2012

ISBN 978-952-60-4529-0
ISBN 978-952-60-4530-6 (pdf)
ISSN-L 1799-4934
ISSN 1799-4934
ISSN 1799-4942 (pdf)

Aalto University
School of Science
Department of Computer Science and Engineering
www.aalto.fi