HELSINKI UNIVERSITY OF TECHNOLOGY
Faculty of Information and Natural Sciences
Department of Computer Science and Engineering

Jan Lönnberg

Understanding students' errors in concurrent programming

Licentiate's Thesis submitted in partial fulfillment of the requirements for the degree of Licentiate of Science in Technology.

Espoo, 11th May 2009

Supervisor: Professor Lauri Malmi
Instructor: Anders Berglund, PhD

HELSINKI UNIVERSITY OF TECHNOLOGY
Faculty of Information and Natural Sciences
Department of Computer Science and Engineering

ABSTRACT OF
LICENTIATE'S THESIS

Author: Jan Lönnberg

Date: 11th May 2009

Pages: 1 + v + 56 + 27

Title of the thesis:
Understanding students' errors in concurrent programming

Language:
English

Professorship: Software Systems

Code: T-106

Supervisor: Professor Lauri Malmi

Instructor: Anders Berglund, PhD

This thesis describes two complementary studies centred around the programming assignments in the Concurrent Programming course at TKK. One is a qualitative, explorative study of how students understand concurrent programs and approach developing and testing concurrent programs. This study is based on interviews with students working on the final programming assignment in a concurrent programming course.

The other study examines the defects in the programs students have written as solutions for the programming assignments in this course and the underlying causes of these errors and the applications in developing teaching, grading and debugging of this information.

I present the effects of the students' approaches to constructing and testing programs on their work, how teaching can be improved to support the students in performing these tasks more effectively and how software tools, especially software visualisation for debugging, can be designed to support the development, testing and debugging of concurrent programs.

Keywords: concurrent programming, phenomenography, errors, defects, program visualisation

TEKNISKA HÖGSKOLAN
Fakulteten för informations- och naturvetenskaper
Institutionen för datavetenskap

SAMMANDRAG AV LICENTIATAVHANDLING

Utfört av: Jan Lönnberg

Datum: 11 maj 2009

Sidantal: $1 + v + 56 + 27$

Arbetets namn:
Understanding students' errors in concurrent programming (Hur studenter gör fel i parallellprogrammering)

Språk:
Engelska

Professur: Programteknik

Kod: T-106

Övervakare: Professor Lauri Malmi

Handledare: Fil. dr Anders Berglund

I denna avhandling beskrivs två kompletterande undersökningar kring övningsarbetena på TKKs parallellprogrammeringskurs (Concurrent Programming). Den ena är en kvalitativ och explorativ studie av studenters sätt att uppfatta parallellprogram och hur de handskas med att utveckla och testa parallellprogram. Denna undersökning bygger på intervjuer med studenter som jobbar med parallellprogrammeringskursens sista övningsarbete.

Den andra undersökningen handlar om felen i programkoden i studenternas inlämnade övningsarbeten, varifrån de här felen kommer och hur den här informationen kan användas för att förbättra undervisning, bedömning och avlusing.

Jag förklarar hur studenternas programmeringsarbete påverkas av deras sätt att arbeta och hur man kan förbättra undervisningen för att stöda studenterna i detta. Vidare beskriver jag hur program kan utvecklas, särskilt för programvisualisering i avlusningssyfte, för att stöda utveckling, testning och avlusning av parallellprogram.

Nyckelord: parallellprogrammering, fenomenografi, fel, defekter, programvisualisering

Preface

When I started my postgraduate studies a few years ago, this was not the licentiate's thesis I was expecting to write. Nonetheless, I found myself guided expertly by my supervisor, Lauri Malmi, and the instructor of my doctoral thesis, Mordechai Ben-Ari, into the field of computing education research. The reason for this seemingly strange detour was simple: to understand the human aspects of the context for which I intended to develop new tools, not just the technical ones. This brought me into contact with a tightly knit and friendly community of researchers, many of whom have contributed to expanding my horizons. First and foremost among these is Anders Berglund, my instructor, who guided me through the phenomenographic research process. I am particularly indebted to these three for all their guidance and many fruitful discussions.

Each of the aforementioned people brought me into contact with a number of people that I've had many interesting conversations with, in particular the members of UpCERG, the Uppsala Computing Education Research Group, and COMPSER, the Computer Science Education Research Group at TKK. I have also found myself a part of collaboration in CER between the Nordic and Australasian countries by participating in the conferences frequented by this clique on both sides of the globe and collaborating with Judy Sheard, Simon and Margaret Hamilton on classifying programming education research at CER conferences. I would like to thank all of these people for welcoming me into their research community.

TES, the Finnish Foundation for Technology Promotion, the TKK Graduate School of Computer Science and Engineering and TKK itself, through a doctoral studies scholarship, provided the funding for this research. Hecse, the Helsinki Graduate School in Computer Science and Engineering, provided funds for travel and also arranged for the additional guidance of Ilkka Niemelä and Keijo Heljanko.

Finally, I would like to thank my parents, without whom I, in more ways than I can enumerate, would not have got to where I am now.

Otaniemi, 11th May 2009,

Jan Lönnberg

Contents

Chapter 1

Introduction

It is almost inevitable that any major project is done to address many different people's needs; not only do the needs of the many outweigh the needs of the one, but it's also desirable to kill two birds with one stone. While this licentiate's thesis can be read as an empirical work on computer science education, my own reasons for doing it are not entirely within that domain. To help the reader understand my decisions and my vision for the future, I will explain my long-range goals and plans and how they have shaped the work presented here.

The work presented in this thesis has its roots in both the fields of software engineering and computer science education. The connection between students' understandings and education research should be clear; the software engineering perspective is less obvious. The long-range goal of my postgraduate research is to help programmers produce better concurrent programs. My approach to this is to develop methods and tools to help programmers understand what a concurrent program does. Better understanding of one's programs' behaviour is essential for both debugging and learning purposes.

1.1 The need for correct concurrent programs

It is estimated that software errors lead to costs of tens of thousands of millions of euros every year [58]. Finding and correcting defects (*bugs*) in a program (an activity known as *debugging*) accounts for a large part of the development time of software and is, in practice, an essential step in producing reliable software.

Many modern software systems are *concurrent*; they involve simultaneously executing or unpredictably interleaved processes. There are many reasons why writing concurrent software is necessary or beneficial. One is performance: computing in parallel allows many tasks to be completed much faster. Another is that the nature of the system one is constructing, for example, a network server or an interactive application, inherently involves concurrency. However, programmers writing concurrent programs face a great challenge in getting concurrent processes to reliably interact properly in the face of this nondeterminism [2, 4]. For these reasons, I am interested in exploring methods to make it easier to find, and then eliminate, these defects and developing software tools that apply these methods.

There are several reasons to focus on students as a target audience. One is that their inexperience means they need more help. Another is that it is easier to introduce new ways to work to students than to experienced professionals with ingrained habits. Finally, helping

students understand their mistakes not only helps them get their programs to work; it also helps them learn.

1.2 Learning concurrent programming

Hughes et al. [31] have conducted a review of two important forums for computer science education research: the journal Computer Science Education (CSE) and the proceedings of the Special Interest Group in Computer Science Education (SIGCSE). They state that while several articles have been published in these forums on the subject of teaching concurrent programming, most of them are about teaching methods and technology. Some contain evaluation, but it is mostly in the form of comments from students or lecturers. They uncovered no articles whatsoever on the subject of evaluating students' knowledge of concurrent programming. They argue that there is a need for quantitative empirical evidence. I argue, based on the mixed methods paradigm suggested by Johnson and Onwuegbuzie [36], that there is a need for theory formation before meaningful quantitative results can be achieved. Also, I found it hard to produce meaningful quantitative results about students' concurrent programming skills without a meaningful underlying model. I explain these difficulties in more detail in Subsection 4.4. The solution to the limitations of the qualitative and quantitative research paradigms is to apply both, allowing the researcher to achieve both theory and confirmation, to find both meaning and numbers and to learn and apply both of these two contrasting paradigms. Thus, in this thesis, I present a qualitative model of several issues related to students' understanding of concurrent programs and then use this as a basis for structuring my quantitative data. This information should be helpful in developing teaching of concurrent programming.

1.3 Research questions

It is a well-established practice of software design to identify the needs of the intended users and then design solutions to address them. The general questions I want to answer are:

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?
3. What kind of tools can assist a programmer in finding and understanding these most problematic defects, and how well do they work?

My primary approach to helping programmers understand their programs is *software visualisation*; using a variety of presentation techniques to facilitate understanding of programs and algorithms and their behaviour. Price et al. [56] state that while software visualisation (SV) "has tremendous potential to aid in the understanding of concurrent programs", few SV systems have seen production use, especially in the domain of tools for professional programmers. They also note that when an 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. Thus, in order to answer the third question properly, I must answer the first two.

This thesis describes my empirical research into how students write concurrent programs. This work provides answers to the first two research questions presented above by shedding light on how students think about concurrent programming and develop concurrent programs, allowing the interpretation of their defects in terms of the underlying errors. The errors and defects can then be seen in the light of the students' approaches and understandings to determine how the students' approaches and understandings differ from those of a programmer skilled in concurrent programming. This allows preliminary answers to the final research question, in the form of suggestions for how to aid students, to be presented. The full design, implementation and evaluation of these tools will be the subject of a later thesis.

1.4 Papers

This thesis consists of three papers and some work of mine that has only been partially published previously [43]. The first two of these papers describe a phenomenographic inquiry I have performed, with the other authors assisting by providing feedback and suggestions when constructing outcome spaces and in writing the paper. I have performed the other phases of data collection and analysis myself and written most of the papers.

The final paper mentioned here describes an application for the information collected in the phenomenographic study. I have written the paper based on ideas I have discussed with the other two authors.

P1 Jan Lönnberg and Anders Berglund. Students' understandings of concurrent programming. In Raymond Lister and Simon, editors, *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007)*, volume 88 of *Conferences in Research and Practice in Information Technology*, pages 77–86. Australian Computer Society, 2008.

P2 Jan Lönnberg, Anders Berglund and Lauri Malmi. How students develop concurrent programs. In Margaret Hamilton and Tony Clear, editors, *Proceedings of the Eleventh Australasian Computing Education Conference (ACE2009)*, volume 95 of *Conferences in Research and Practice in Information Technology*, pages 129–138. Australian Computer Society, 2009.

P3 Jan Lönnberg, Lauri Malmi and Anders Berglund. Helping students debug concurrent programs. In Arnold Pears and Lauri Malmi, editors, *Proceedings of the Eighth Koli Calling International Conference on Computing Education Research (Koli Calling 2008)*. Uppsala University, 2009. To appear.

Chapter 2

Related work

As this thesis is closely related in its goals and methods both to research in software engineering and in computing education, it is necessary to consider related work in both fields. In the following, I first briefly explain the aspects of concurrent programming most relevant to this thesis (Section 2.1) and then describe software defects and the process of finding and eliminating them with emphasis on the types of defects that can occur in concurrent programs (Section 2.2). Having established the programming context, I then present relevant parts of the fields of computing education (Section 2.3) and program visualisation (Section 2.4).

2.1 Concurrent programming

A *concurrent* program is a program that contains two or more *processes* that co-operate to achieve a goal, where each process is a set of sequentially executed instructions like a sequential program. If the underlying system has separate memory spaces for different programs, a lightweight variant of a process called a *thread* may also exist that shares memory and many other system resources with other threads. The term "process" is then used to refer to an address space with at least one thread running in it. In this thesis, I will follow common usage in using the term "process" in situations where the distinction is irrelevant (e.g. when discussing many concurrent algorithms and concepts on a general level) but distinguishing where necessary between processes and threads (e.g. when discussing a program implemented using Java [26] or POSIX [34]).

Several important types of concurrent program can be discerned [2, 4]:

Multithreaded programs are programs running in a single memory space on a single computer that perform many simultaneous activities and are therefore easy to structure in terms of separate threads. Many interactive computer programs, especially in graphical environments, belong to this category.

Parallel programs are programs that are designed to use multiple processors at the same time to perform intensive calculations quicker.

Distributed programs are programs that are divided among computers in different places, for example to provide access to remote data, co-ordinate activities over a wide area or provide redundancy.

Most workstation and server operating systems (e.g. Unix) provide support for dividing running processes between processors. Alternatively, a simulator can be used to simulate a multiprocessor machine for research or teaching purposes [55].

Although a primer on concurrent programming is beyond the scope of this thesis, I will briefly describe the aspects of concurrent programming that are relevant to this study.

The most important distinction between sequential and concurrent programs is the inherently nondeterministic behaviour of concurrent execution; it is unknown how much of one process is executed during the time another one executes an instruction. The greatest challenge in writing concurrent programs is getting concurrent processes to reliably interact properly in the face of this nondeterminism [2, 4].

In order to co-operate, processes must be capable of communicating with each other. Many different mechanisms for interprocess communication (IPC) are available for this; only those relevant to this study will be described here.

2.1.1 Basic IPC mechanisms

One of the most common IPC mechanisms is *shared memory*: memory that can be read and written by many different threads. This is typical of multithreaded systems written for e.g. Java [26] or POSIX [34].

Other IPC mechanisms are typically provided to complement shared memory, many of which focus on preventing processes from proceeding with potentially harmful execution. The simplest is the *lock* or *mutex*, which allows the programmer to designate *critical sections* in a program associated with a lock. Only one of the critical sections associated with a lock can run at any given time. While the lock solves the problem of mutual exclusion, it does not provide a way for a process to wait for another to complete an operation without *busy waiting*, repeatedly checking whether a condition is true. [4]

The *semaphore* can be considered an extension of the lock; it is essentially a shared non-negative integer value accessible through two operations: `V` (also known as `post` [34] or `release` [69]), which increases the value of the semaphore by one, and `P` (also known as `wait` [34] or `acquire` [69]), which waits until the value of the semaphore is positive and then decreases it by one. Semaphores commonly include an additional non-blocking equivalent to `P` called `tryAcquire` [69] or `trywait` [34] that succeeds if the semaphore has a positive value (which is decremented by one) or reports an error if the semaphore's value is zero. An important property of these operations is that they are *atomic*; except when an operation is explicitly waiting for something to happen, none of the intermediate states are visible to other processes nor can a process change anything affecting an ongoing operation in another process.

Another common IPC construct is the *monitor*, which consists of a lock and *conditional variables* associated with an object. Condition variables provide a way to wait for an event by containing a set of waiting processes. They have three operations: `wait`, which adds a process to the wait set and starts waiting, `signal` or `notify`, which removes one process from the wait set, waking it up and, in some variants such as in Java, `notifyAll`, which empties the wait set, waking all waiting processes. [4, 26]

A variety of *message-passing* mechanisms are often used in distributed systems; as the name implies, these involve processes sending messages to each other and receiving them either by waiting for a message or by buffering messages for later reading.

2.1.2 Tuple spaces

Gelernter [24] describes, as a central part of the distributed programming language *Linda*, an interprocess communication mechanism called a *tuple space*. As its name implies, a *tuple space* consists of a space containing *tuples*, data records consisting of a *tag* (an identifier for the type of tuple) and (an ordered list of) zero or more data values. A tuple space can be accessed through 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. A pattern matches a tuple in the space if the pattern and tuple are the same length and all the data values in the pattern are the same as the corresponding values in the tuple. 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. All these operations are atomic.

Many variations on tuple spaces are possible. Later versions of Linda add versions of `in()` and `read()` that do not wait for a matching tuple to be put in the tuple space if none exists in the tuple space at the time they are called; these non-blocking versions either report that no matching tuple exists or return a matching tuple the same way the normal, blocking, operations do [17].

Tuple spaces can be used in many different ways. Therefore, they can be considered to be generalisations of several different IPC mechanisms. On the one hand, they are a form of shared memory; on the other, they can be seen as semaphores with associated data or a message-passing mechanism; all of these can easily be implemented using a tuple space [24].

2.2 Software defects

Software often behaves differently than intended; in other words, it *fails*. This is because its developers have written code that does something other than what it should; the program has *defects* (commonly known as *bugs*); discrepancies between the actual program and a corresponding correct one (i.e. a program that would always do what it should). Defects are introduced into programs by *developers* (the people involved in developing a program, in particular, programmers and designers) making an *error*; a mistake, misconception or misunderstanding. Descriptions of a what a program (or parts of it) should do are called *specifications*. Many different specifications may be written for the same code with different focuses (ranging from user requirements to detailed descriptions of the intended internal behaviour); in some cases (especially when a specification is incomplete), whether some behaviour of a program constitutes failure or not depends on which specification is used. [14]

Note that this definition implies that all defects result, in at least one situation that a program is expected to cope with, in failure. It also implies that a defect is always caused by an error made by a developer. The latter assumption is not, strictly speaking, true, since defects can in some cases be introduced by e.g. hardware-related problems resulting in data corruption. It is usually taken for granted that human error is overwhelmingly the most common cause of defects [14], so in this thesis I will also ignore all non-human sources of defects.

The process of trying to show that the result of a phase of software development (in the case of the implementation phase, the resulting program) meets the requirements for the

phase (the specification for the program) is called *verification*. The process of removing a bug, once it is known to exist, is called *debugging*.

2.2.1 Verification

Verification is typically done by *testing* the program. Testing a program involves executing it with the intent of getting defects to manifest themselves as failures or evaluating the program's *quality*, which includes all desirable and required aspects of the software such as correctness, reliability, usability and security. Especially when the goal is to find defects, it is important to choose test cases to find as many different possible defects as possible. In order to find incorrect behaviour (either the program failing to do what it should or doing something it should not), the expected result or output of each test must be known and checked against the actual result or output. Testing should be a planned part of the software development process and carried out by a group independent of the development group. Both valid and invalid input conditions must be tested. [14, 51]

The nondeterminism inherent in concurrency complicates verification a lot. Not only may some defects only manifest if many specific features of the concurrent program are used at the same time, these failures may not even occur consistently.

One way to address the problem of finding concurrency bugs is to increase the chance of interleavings that lead to failure. *Stress testing*, in which a system is made to use as much resources as possible by subjecting it to a heavy load (e.g. large amounts of input), is a testing approach that often uncovers problems related to concurrency and timing, such as race conditions and deadlocks. [14]

The usefulness of stress testing can be further improved by making sure process interleavings occur often and in many different places. One straightforward and realistic approach is to distribute the program's threads over multiple processors. Another way to do this is to automatically and randomly change the thread scheduling to make concurrency failures more likely to occur (e.g. [68]).

There are several approaches to ensuring correctness despite nondeterminism, including *deductive proofs* (usually manually constructed) and *model checking*, which can be based on a simplified computational model, such as in Spin [30], or an actual programming language, e.g. Java PathFinder [70].

Model checkers are often used to find concurrency-related defects by specifying requirements that are exhaustively checked against all the possible states or executions of the program. If the requirement does not hold, a counterexample is generated that consists of an execution of the program that violates the requirement. Model checking can in many cases be used to prove correctness properties. However, as model checkers require that the program has quite a small state space and does not interact with entities outside the program model, adapting programs to a model that can be verified is often hard and error-prone work [70].

2.2.2 Debugging

Debugging is usually done by executing the program with different inputs and examining its intermediate and/or final results. This is typically done by manually adding test code that calls different parts of the program that is being examined and examining the results in

textual form or through a program intended to aid debugging (a *debugger*). Some debuggers, such as DDD [71] employ *visualisation* to make the data shown easier to understand; this is explored in more depth in Section 2.4.

Most of the previous work on debugging has concentrated on studying the current state of the program and examining the execution of individual processes. Concurrent programs, however, are likely to have defects that involve unexpected interactions between concurrent processes that are hard to debug using these methods. A promising solution to this that complements existing verification and debugging techniques involves visualisations designed to aid the programmer in understanding the interactions between the operations performed in a program.

Debuggers are programs intended to be used to find bugs in programs by tracing through the execution of a program and examining and editing variable values. Most debuggers have some execution control facilities (e.g. single stepping, breakpoints, watchpoints and expression evaluation) and some way to view variable values.

Most current debuggers are based on ideas from *FLIT* (*Flexowriter Interrogation Tape*), which introduced symbolic debugging (allowing the user to work with variable names, labels and instruction mnemonics instead of memory addresses and numerical instruction codes) and breakpoints [67]. Almost 50 years later, most current debuggers are based on the same paradigm: run the program to a specified point, stop it and examine the values of the variables. A few improvements have been made, such as *single stepping*, the automatic placement of temporary breakpoints after the next instruction to be executed to execute one instruction at a time. These debuggers can be referred to as *command line debuggers* (referring to their user interface) or *traditional debuggers* (referring to their heritage).

The most common style of debugger today is a traditional debugger with a graphical user interface. Most *integrated development environments* (development software packages containing an editor, a compiler and linker and a debugger within a common user interface), such as Eclipse and Microsoft Visual Studio, contain a debugger of this type. Debuggers of this type are referred to by their authors as "visual debuggers" or "graphical debuggers". To avoid confusion when discussing debuggers that use software visualisation techniques in Section 2.4, I will refer to traditional debuggers with graphical user interfaces as *graphical debuggers* and reserve the term *visual debugger* for debuggers with software visualisation features.

2.2.3 Classification of defects

In order to meaningfully analyse a variety of different related phenomena, one needs a way to group them together into categories that bring out the aspects that are relevant to one's analysis. Likewise, trying to communicate results without a suitable classification quickly becomes bogged down in details.

Defects in programs can be classified in a variety of ways, depending on the relevant aspects, which in turn depends on the purpose of the analysis. For example, Eisenstadt [22] and Grandell et al. [27] form categories from the observed defects. Yet, as their contexts and goals differ, their classifications have major differences. Eisenstadt is interested in understanding the process of debugging in order to develop better tools, and therefore studies only bugs that have been hard to find. For the same reason, he classifies these defects based on why they were difficult to find, how they were tracked down and the underlying error or violated requirement. Grandell et al., on the other hand, are computing education research-

ers trying to understand and improve learning to program and studied programs created by novice programmers. They divide defects into syntax and logic defects and construct from their data a division of the latter into five specific categories: accepting erroneous input, incorrect/missing algorithm, mixing up constructs, incorrect or redundant declarations and other.

Others (such as Spohrer and Soloway [65]) construct a classification based on distinctions they wish to study, such as whether the defects are caused by misconceptions about programming language constructs. Defects can be classified, for example, based on their symptoms (how and when it manifests itself), or on the difference on the syntactic level in the code between the incorrect code and the intended correct code. In both cases, a variety of different category sets have been formed by different authors.

If sufficiently detailed information is available on the underlying errors, defects can be classified based on the underlying error. For example, to construct a cognitive model of programming errors, the errors can be classified by the type of cognitive breakdown involved in the error (lack of knowledge, mistake) and the information involved (as done by Ko and Myers [39]). Alternatively, the part of the program design that is incorrect can be used, as in the goal-plan analysis done by Spohrer and Soloway [65]. Knuth [37] makes use of introspection to classify his own errors according to the aspect of his programming process he failed in.

In a software development context, many different types of information related to bugs are useful, resulting in a multifacted classification scheme such as the IEEE standard classification for software anomalies (deviations from expectations observed in program operation or documentation, including bugs) [33]. In the IEEE classification, bugs are classified according to a wide range of properties, such as how and when the defect was detected, the type of the defect and the error underlying it and the impact of the defect. Beizer's classification [3] focuses on the aspect of the program or development task that was incorrectly developed (e.g. requirements, data structures, data processing).

2.3 Computing education research

The field of computing education research can be seen as education research limited to the context of computing. This is reflected in how Pears et al. [54] divide the field of computing education into four areas:

Studies in teaching, learning and assessment including case studies, empirical studies, reports on ways to carry out instruction and discussions of practice, teaching methodology and assessment

Institutions and educational settings including issues of student retention, international collaboration, curricula and innovation on the institutional level

Problems and solutions in the form of advances in teaching technology and methodology

CER as a discipline including methdology and organisation of computing education research

While this thesis details empirical work and thus belongs firmly in the first area, it is strongly oriented toward forming an empirical background for improvements to teaching,

particularly in technological form, which belongs in the third area. It is hardly surprising that much of the relevant related work is in these areas as well. Appropriate research methodology is important to any research, so work in the first three areas will often, and indeed should, involve work in the area of CER as a discipline.

Fincher and Petre [23] use a division with ten subfields (which Pears et al. combine to form their first three areas). These include (in roughly descending order of relevance to this thesis):

Student understanding including issues such as "What design behaviours do students exhibit?", is clearly the subfield to which this thesis primarily belongs.

Animation, visualisation and simulation in the form of visualisations to help students understand concurrent program behaviour is the primary motivation for this work.

Assessment can be done more accurately and easily if one understands how students think and work.

Educational technology in the form of tools to assist students in creating and testing concurrent programs is another motivation.

Teaching methods can be developed that address problems in student understanding.

Transferring professional practice into the classroom is both a goal and a tool in teaching CS; an awareness of students' practices is helpful in achieving this.

The remaining subfields are not directly relevant to this thesis. This division provides a framework for the following discussion of related work.

The above classifications essentially ignore the context. Therefore, dividing by context constitutes an orthogonal division of the field. This is one of the classifications used by Simon [61]. One of his categories of context, *programming*, obviously contains the context of this work, but is very broad, with surveyed conferences containing at least 35 % programming papers and, in one case, 74 % programming papers [61–63]. Many of these papers are also clearly about an introductory programming context. For this reason I will only describe work on students' understandings of and development practices for concurrent programming in the following, except where the context is similar enough for results to be comparable (e.g. an advanced programming course).

2.3.1 Students' understanding of programming

Students may approach the task of developing a concurrent program with different goals in mind than their teachers. Ben-David Kolikant [7] describes how students understand correctness in a concurrent programming context and how this affects the development process. She notes that students define a "correct program" as a program that exhibits "reasonable I/O for many legal inputs". Experts, on the other hand, define a program of good quality to be one that behaves correctly for all legal input and is efficient, legible, documented and modular. Roughly a third of the students were sometimes satisfied with only compiling their program to ensure it is correct and half of them did not check that their program's output is correct.

Considering what students are trying to achieve, it is hardly surprising that they use unsuitable approaches when they develop concurrent programs. Ben-Ari and Ben-David Kolikant [5] describe how high-school students' concurrent programming conceptions and working methods change during a course on the subject. They found that students have difficulties limiting themselves to operations permitted by the concurrency model, make assumptions based on informal concepts rather than use formal rules and avoid using concurrency and apply development approaches that do not work well in concurrent programming, such as testing a program with a few representative inputs.

One reason for these problems is that students act as users of programs rather than developers. Ben-David Kolikant [6] describes learning concurrent programming in terms of entering a community of computer science practitioners. She finds that the students initially approach the concurrent programming assignment from a user's perspective, in which only the program behaviour seen through the user interface is taken into account, and not all of them are able to switch to a programmer's perspective.

Carbone et al. [16] have performed a similar study of students' approaches to programming assignments with students on the first year of their Bachelor of Computer Science or Bachelor of Information Management and Systems studies. They have also used semi-structured interviews and students' self-reports as their data sources, and have found different motivations for programming and technical and personal skills that affect programming. Boustedt [12] describes the understandings and experiences of students developing part of a large software system in a object-oriented programming course. He describes phenomenographically how the students understand different parts of the system (interfaces, plugins, systems) in terms of perspectives ranging from a basic definition of the term to an understanding of its usefulness in creating software. He also classifies students' descriptions of the development process (including the outcome of the assignment and the various stage of it). Clancy [19] describes a variety of different misconceptions and problematic attitudes found among novice programmers.

Eckerdal and Berglund [20] present the way in which learning to program is understood by different students. The understandings range from the limited perspective of learning programming being to understand a programming language and use it to write programs to seeing learning to program as learning a new method of thinking that can be used for problem solving, even outside the programming course.

Booth [11] describes the different ways students on an introductory programming course understand programming, programming languages, learning to program, functions, recursion and correctness. A common theme to the outcome spaces she finds is a progression. This progression starts from surface knowledge (such as understanding recursion on a syntactic level as a feature of the ML programming language). It proceeds through making use of this knowledge to implement programming concepts (e.g. using recursion to achieve iteration) and finally reaches solving a problem (e.g. using recursion to repeatedly reduce a problem to a simpler one until a base case is reached). She also describes the approaches used by the students to write programs. Eckerdal and Thuné [21] as well as Sorva [64] provide a recent example from the computer science domain: novice Java programmers perceive objects and classes in various ways. The results of Eckerdal and Thuné are similar to Booth's: objects and classes are seen purely syntactically, as active entities or descriptions of such and as models of real world phenomena. Sorva, on the other hand, finds several understandings that are incorrect, not just incomplete.

2.3.2 Assessment

The results of an assignment can, collectively, be used to determine whether students are effectively learning what they should. In particular, if a large number of students has problems understanding and/or applying some relevant knowledge, the teaching of this knowledge needs to be improved. As correctness and efficiency are typically the focus of programming courses, especially in concurrent programming, I will concentrate here on assessment of these properties and skills related to achieving them in programs to the exclusion of other issues such as style.

One problem is that students may produce many defects unrelated to the subject matter they are being taught; in other words, the assignment may be testing the wrong knowledge and skills. If the defects can be traced to misconceptions about the assignment or any assignment- or course-specific additions or changes to the environment in which it is done (if any), the students may be distracted from learning relevant matters by difficulties specific to the assignment. Penalising students for defects that are arguably caused by a badly-designed assignment rather than any problem the student may have is hardly just, so it is important to recognise or eliminate these defects.

Automatic assessment of programming assignments is typically done in systems such as BOSS [44] or CourseMarker (formerly CourseMaster) [29] by executing test runs on the code to be assessed and assigning a grade based on the number of tests that passed [1]. With larger programming assignments, this technique is usually used as a supplement to manual assessment instead of a substitute [1].

Testing only exposes the symptoms of a defect. Therefore, it is hard for an automatic assessment system to tell which errors are relevant to the assignment. Again, this work is intended to help mitigate this by identifying errors in students' concurrent programs that are irrelevant from the point of view of the learning goals of our Concurrent Programming course. These can then be taken into account in the testing system by correcting the defect in the student's program or working around them (by avoiding problematic cases in order to test other functionality).

Another of the problems with automated assessment (as with testing in general) is that it is hard to design tests that detect all common errors and distinguish between different types of error without empirical data on the errors to look for from real students. My research into error types and frequencies in concurrent programming assignments is also intended to mitigate this problem.

As noted in Section 2.2, testing does not work as well with concurrent programs, as the relative timing of the execution of different operations can have a critical effect on both the desired and the actual behaviour of the program. In the case of concurrent programming assignments, the nondeterminism inherent in concurrent programs can in many cases be mitigated by repeating tests many times with different interleavings. Although model checking does this exhaustively, it easily becomes prohibitively expensive in terms of CPU time and memory when applied to large programs. This means that alternative methods to make a program's concurrent operations interleave in different ways, such as adding random delays to the source code or using a model checker to generate random interleavings, may be more effective in grading concurrent programming assignments. For this reason, manual static assessment (reading and analysing the source code without executing it) seems to be favoured for concurrent program assessment.

The exceptions typically involve a very limited form of concurrent programming and

an alternative approach to finding defects. One example is *SYPROS* [28, 57], an intelligent tutoring system intended to allow students with basic theoretical knowledge of concurrent programming to practise the use of semaphores in simple but important problems such as the reader-writer problem. The student solves the problem by adding semaphores and operations on them to the statements given in the specification in order to meet the requirements. SYPROS adapts to the behaviour of the student by using an expert system that models the problem-solving process using rules to transform synchronisation conditions into synchronisation statements that are added to the program and combined using a set of heuristics. This expert system tracks the student's activity in the graphical task editor and attempts to find an explanation for the student's activities that matches a variant of the solution process. If the student has problems forming a valid solution (e.g. repeatedly makes the same mistake in many exercises, fails to take a goal into account, does not proceed from an incomplete solution), SYPROS will start to provide information on what's wrong with the student's solution in the form of hints of increasing specificity including displaying animations of counterexamples that demonstrate where his or her code fails. In extreme cases, the correct solution will be provided and a simpler problem given. SYPROS is limited to simple novice-level synchronisation problems with semaphores. However, within that narrow field it provides feedback roughly equivalent to that of a personal tutor to students with little added investment on the part of the teachers; in particular, it helps students see their mistakes and get them unstuck when they are stuck. Naturally, developing such an expert system requires detailed knowledge of how students understand concurrent programming concepts and the process of developing a concurrent program.

2.4 Program visualisation

Software visualisation is "the use of the crafts of typography, graphic design, animation, and cinematography with modern human-computer interaction and computer graphics technology to facilitate both the human understanding and effective use of computer software". It has two major subfields: *program visualisation* ("visualization of actual program code or data structures in either static or dynamic form") and *algorithm visualisation* ("visualization of the higher-level abstractions which describe software", such as algorithms and the data structures they manipulate). [66]

Software visualisation is a very large field, so it is necessary in this thesis to limit the discussion to a few relevant examples of the field of *program visualisation*, with an emphasis on visualisations that can be used to visualise concurrent programs effectively for the purpose of understanding unexpected behaviour in a learning or debugging context.

A lot of visualisation research has been done that 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 [32].

2.4.1 Visual debugging

Program visualisation has been applied to debuggers to create *visual debuggers* such as DDD [71]; debuggers with graphical representations of data. They do not appear to fully make use of potentially useful visualisation and interaction techniques; most have very limited visualisations and many provide only a graphical replacement for the traditional

textual user interface. DDD's visualisation mostly consists of displaying data structures or objects as boxes interconnected by arrows representing pointers. Lens [49, 50] allows animations to be constructed based on data in programs and is arguably therefore more a tool for constructing program visualisations than the visual debugger it is claimed by its authors to be.

Most debuggers concentrate on individual threads and can only show the current state of the program while the cause of a program malfunction usually lies in the past (which is especially problematic in concurrent programs, as duplicating a failure may be difficult); RetroVue [15], with its tree view of all executed operations, ability to examine all previous states of the program and thread display showing lock interactions between and execution times of threads, is a clear exception to this. However, it does not aid the programmer much in finding interrelated operations except in the case of direct lock interactions.

The Whyline [38], which uses dynamic dependence graphs to explain to novices the reason why a program did something (wrong), addresses the problem of explaining relationships, but is limited to low-level explanations of simple causal chains in a limited beginners' environment.

2.4.2 Visualising concurrent programs

A few debuggers and program visualisation systems have been designed with concurrency in mind. Most of them (e.g. JAVAVIS [53] and JAN [42]) use sequence diagrams to display method calls; JaVis [48] adds collaboration diagrams to show interactions between objects. These diagrams have a level of detail suitable for debugging, but become cumbersome for complex executions. Kraemer [40] 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.

Chapter 3

Empirical work

In this chapter, I explain the empirical work I have done, concentrating on the setting and the data collection and analysis methods I have applied. First, in Section 3.1, I explain the concurrent programming course that forms the setting for this study. I then explain the methodology applied (Section 3.2) and then how data collection (Section 3.3) and analysis (Section 3.4) were done in practice.

3.1 Setting

This study is centred around the T-106.420/T-106.5600[1] Concurrent Programming course[2] at Helsinki University of Technology (TKK) in Autumns 2005 to 2008. 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 Concurrent Programming course is held once every autumn, and is originally based on a coursebook by Andrews [2], which was replaced in 2006 by a new book by Ben-Ari [4] with books on concurrent programming in Java by Goetz et al. [25] and Lea [41] as supplementary information for the programming assignments.

Up to the academic year 2004-2005, the course was mandatory for Software Technique students and one of two alternatives for Telecommunications Software and Applications students. The course remains mandatory for students of the NordSecMob Master's Programme in Security and Mobile Computing and can be part of the Master's Programme in Mobile Computing - Services and Security. This means that most students have completed a Bachelor's degree or a roughly equivalent part of a Master's degree.

[1] The course code was changed for the Autumn 2006 instance, but the content remained essentially the same.

[2] Course web sites at:
2005: `http://www.cs.hut.fi/Studies/T-106.420/main.html`
2006: `http://www.cs.hut.fi/Studies/T-106.5600/2006/english.shtml`
2007: `http://www.cs.hut.fi/Studies/T-106.5600/english.shtml.old`
2008: `https://noppa.tkk.fi/noppa/kurssi/t-106.5600/etusivu`

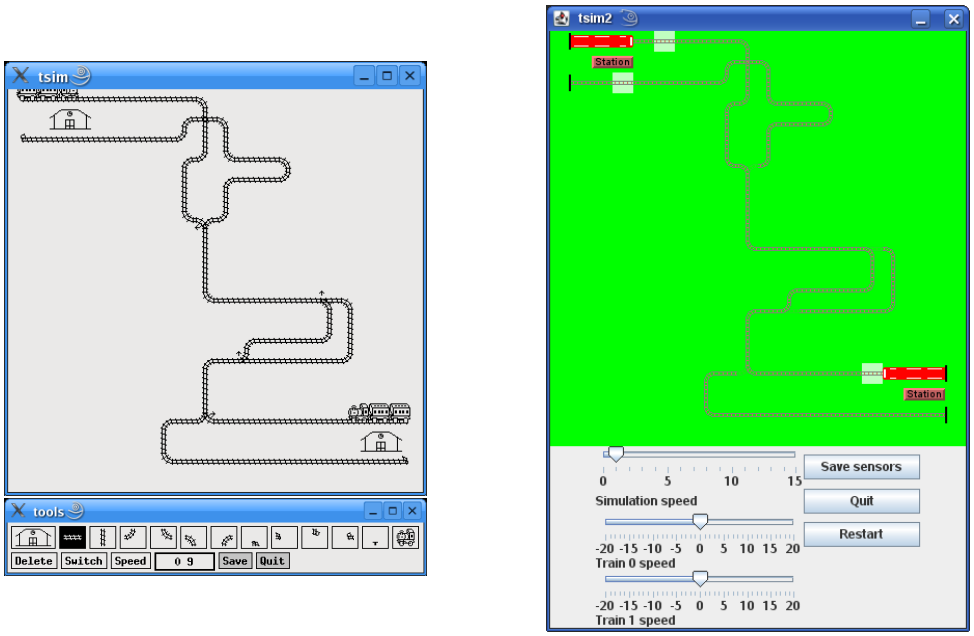

Figure 3.1: tsim (left) and tsim2 (right) showing track used in Assignment 1

3.1.1 Assignments

To allow students to apply this knowledge in practice and assess their skill, I was asked by the professor responsible for the course to design a set of programming assignments for the course. The resulting three assignments have been used with small changes in all yearly instances of the Concurrent Programming course since 2005. Students can either work alone or work on one solution as a pair.

In 2005, 2007 and 2008, students were required to submit a program that solves the specified problem as well as a textual report explaining how their solution works. In 2006, the students were initially required to submit only their Java source code. In the event that their solution was rejected, they were required to submit corrected program code and a report explaining the reasoning behind the erroneous code and the steps they took to correct it. In 2008, to compensate for only being allowed one attempt at each assignment, students were given access to the test package previously used only by the course staff for assessment.

Each assignment was explained to the students in the form of a specification on the course's web site and a presentation by me of roughly an hour in standard lecture format.

Trains

In the first assignment (*Trains*), 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 together with a test package that runs the simulation quickly without the user interface. The students' task is to write code that drives these 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 simulator allows students to practise basic concurrent programming using semaphores in an environment where failures are clearly visible (for example, a mutual exclusion defect can manifest itself as a collision between trains). The supplied track 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).

Reactor

The second assignment (*Reactor*) concentrates on the Reactor design pattern [59] 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 (`synchronized`, `Object.wait()`, `Object.notify()` and `Object.notifyAll()`), 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 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.

Tuple space

In the third assignment (*Tuple space*), the student implements a simple tuple space (see Subsection 2.1.2) containing only blocking `get` and `put` operations on tuples implemented as `String` arrays. They are 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 assignment lets students familiarise themselves with writing distributed systems using message passing.

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

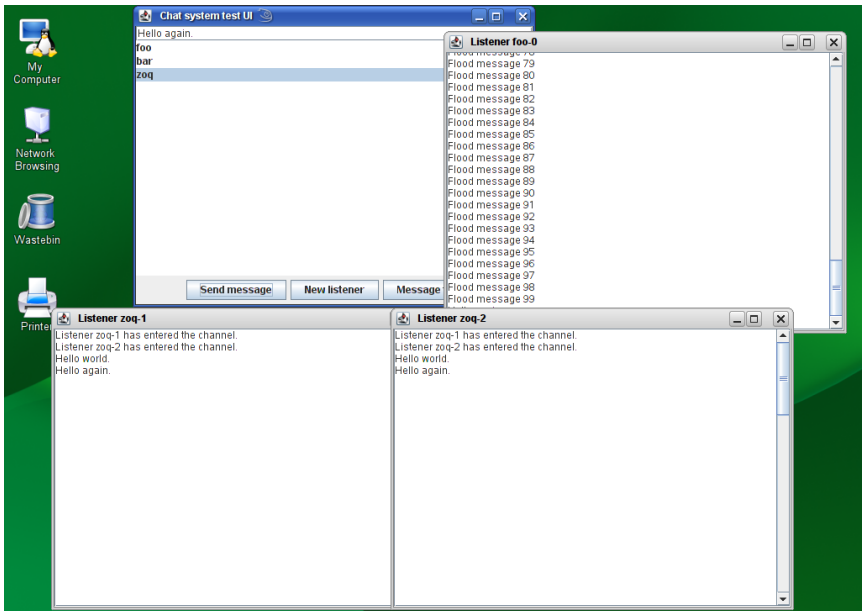

Figure 3.2: Chat UI for testing of Assignment 3

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 Methodology

Originally, this study was designed as a primarily quantitative classification of defects found in our students' concurrent programming assignments. However, due to lack of understanding of the students' understandings and approaches to developing, less than half of the defects could be classified by the area in which the student made an error, as was my intent [43]. In order to create a model of how students understood the relevant concurrent programming concepts and approached the process of developing their program, a qualitative part was added to find the different ways in which students understand concurrent programming and approach the programming task. This model was then used as a basis for the classification.

3.2.1 Phenomenography

The key research approach in this study was *phenomenography*. This approach aims to reveal the different ways in which something is understood in a cohort [46]. In recent years, the interest in phenomenographic research has increased in the computing education community, since the results that are offered, focusing both on the learners and what they learn about, have been shown to be useful within computing education [8, 9]. This interest can be seen in the amount of phenomenographic studies of programming published at conferences such as ICER, ITiCSE and Koli Calling in 2005–2008; these papers outnumber programming studies that use similar methods such as grounded theory and content analysis even when one takes into account conferences where phenomenography was not used to research

programming in this time period, such as SIGCSE, ACE[3] and NACCQ [60]. The use in this study is mostly consistent with this, as the aim is to reveal how certain phenomena are understood by a cohort.

Marton [45] notes that the world can be studied in terms of two different perspectives that he terms *first-order* and *second-order*. The first-order perspective involves examining and making statements about selected aspects of the world (*phenomena*), while the second-order perspective involves examining and making statements about how people experience these phenomena.

Experiences can be seen as a form of relationship between a phenomenon and the person who experiences it. The experience involves distinguishing the phenomenon from its context. It is clear that different people experience the world in different ways, none of which is a complete understanding. [46]

Marton [45] argues that although educational research often focuses on the first-order perspective, the second-order perspective can also be fruitful in educational research. He notes further that second-order knowledge cannot normally be derived from first-order knowledge; we have no effective way to deduce how different people think about the world from what we know about it.

Marton [45] continues by describing the second-order research approach, phenomenography, as "research which aims at description, analysis, and understanding of experiences" and states that its focus is on understanding the variation in these experiences. The outcome of a phenomenographic research project is thus a set of *categories of description* grouped into *outcome spaces*, where each category describes a qualitatively different way in which a phenomenon is understood. Each outcome space contains the different understandings found for a phenomenon or an aspect of it, typically organised in a hierarchy of complexity [46]. In other words, phenomenography is used to understand the qualitative and collective variation of understandings or experiences of a phenomenon.

Berglund [8] describes the process of phenomenographic research in computer science education as consisting of a data collection phase and an analysis phase. In the data collection phase, the researcher interviews students about the phenomenon under investigation. The students are chosen with the intent of getting a diverse sample, in order to get a rich variation in their experiences of the phenomenon. Similarly, the interview must allow the student to express his understanding of the phenomenon of interest in many different ways. The interviews are then transcribed for analysis, during which the researcher looks for quotes that illuminate the students' various understandings and classifies quotes into categories of description. The analysis phase is typically iterative, with the tentative categories changing repeatedly as the researcher refines his analysis.

3.3 Data collection

The data collection was done during the 2005 to 2008 instances of the Concurrent Programming course. All data were collected from students on the Concurrent Programming course and either consisted of their work for the course or interviews with this work as their subject.

[3] Note, however, that this thesis includes a phenomenographic paper about programming from ACE *2009*!

3.3.1 Students' programs and reports

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 assistants working according to specifications I provided and whose work I checked and, as needed, assisted with. For the 2005 course, I did all the assessment myself. In 2006, Teemu Kiviniemi did most of the assessment, in part using his own classification; the results for this year are therefore omitted. In 2007, Kari Kähkönen, Sampo Niskanen, Pranav Sharma and Yang Lu each did roughly a quarter of the assessment using, to ensure consistency, a defect classification I made based on the 2005 results. In 2008, the assessment was done using the same defect classification by Ari Sundholm, Pranav Sharma and Pasi Lahti. This classification is explained further in Subsection 3.4.2.

3.3.2 Interviews with students

I conducted interviews with eight selected students regarding the Tuple space assignment (see Subsection 3.1.1) of the Concurrent Programming course of Autumn 2006. The interviews were conducted between the announcement of initial submission results and the resubmission of failed assignments. The focus of the interviews was on the development process, especially the students' reasoning behind their design, with a focus on problems found in their programs.

Twelve groups of students (nine students who did the assignment alone and three pairs who collaborated on the assignment) were selected for interview based on the assessments of their initial submissions for the third assignment. In order to maximise the variation of experiences based on the information available to us about the students, 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 passed the assignment (on their first try) were chosen and 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.

While only students who failed the assignment participated in the interviews, this can be seen as purposive sampling (emphasising problems learning concurrent programming). Successful students could have more advanced understandings. However, the understandings described in this paper already cover a wide range from novice to expert understandings, which suggests that also interviewing more successful students may not have affected the results.

The interviewees were allowed to choose the language of the interview from the languages I speak fluently: Finnish, English or Swedish. Four students who worked alone were interviewed in their native Finnish. The pair of students was interviewed in Finnish, as this is the native language of one of them and the other, whose native language is Swedish, is also fluent in Finnish. Two East European students were interviewed in English, their primary language of instruction at our university. The pair of students was interviewed together.

The interviews were semi-structured, i.e. they were in the form of a conversation using a set of prepared questions as conversation starters, and lasted from 30 minutes up to almost

an hour. This allowed students to, in addition to the topics I raised, talk about related issues such as their experiences with the other assignments in the course or with programming in a professional context. 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 all of the interviews using a single table-top microphone and transcribed them. I also wrote down the main points of the interview directly after the interview. These recordings and notes, as well as the code and documents submitted by the students and the teaching assistant's assessments of the students' submissions, form the source material. 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. When interviewing, I had identical printouts of the code and a copy of the feedback given by the teaching assistant to the students, as well as a copy of the guidelines for the interview (in Appendix B). The printouts included line numbers to make it easier to reference sections of code verbally and unambiguously.

3.4 Analysis

Initially, the data analysis was intended to only involve finding and classifying the defects in students' concurrent programs under the assumption that I could easily identify these. When this failed due to my lack of understanding of students' understandings of concurrent programming, I started a phenomenographic study with the help of Anders Berglund, a researcher from Uppsala University with experience in using phenomenography to study students' understandings in computer science.

3.4.1 Understandings

I did the analysis in discussion with Anders Berglund and, in a later stage, Lauri Malmi. Specifically, we first discussed the contents of two interview transcripts and then 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. Initially, quotes were grouped together if they appeared to express similar standpoints (for example, tuples are, or are represented as, arrays). Then they were grouped together into tentative categories representing similar understandings of a phenomenon, which were grouped into tentative outcome spaces based on the phenomenon being discussed.

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. Some quotes were left out of consideration as not fitting the research approach and/or area of interest of this particular study. In some cases, only a few quotes regarding a phenomenon

were found, in which case the data were deemed insufficient for further analysis.

For most of the questions we investigate, we have found that the quotes we found fit into a phenomenographic framework. In one case, that of sources of failures taken into account by students, the standpoints expressed by the students do not refer to a single phenomenon but several related phenomena. Here, we have chosen a different method to list the sources of failure mentioned by the students, although the data collection and analysis has essentially followed phenomenographic procedure.

3.4.2 Defects and errors

In order to serve the requirements of both teaching and tool development, 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 is lacking in the student who introduced the defect. In the other classification, defects are divided based on whether the program failures they cause occur deterministically.

Note that apparently non-functional requirements (such as using a mechanism that is not available) can be considered within a framework of errors resulting in defects 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 (e.g. different threads actually see separate locks instead of one). Since such requirements are typically based on a notional execution environment, it is natural to use the failure induced by this type of error in such an environment for classification purposes. This also makes this classification by failure consistent when limitations of a notional environment are introduced in the real environment, 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 or her understand his or her 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 [65]. With some minor refinements and additional defect classes, this classification was used as a basis for assessment in 2007 and 2008.

I initially combined the defect classes described above 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 Section 4.4.

Classifying defects by failure

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

Deterministic failures occur consistently for a given input (or sequence of stimuli in the case of a reactive program) 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. History-based debugging methods can also be used.

Nondeterministic failures are hard to duplicate; a logging-based debugging approach is therefore more useful than traditional debugging, since the failure only needs to occur once while logging is being done.

Since the debugging technique must be chosen based on the symptoms and nondeterministic failures may appear to be deterministic in many tests, it makes sense to always use techniques appropriate for nondeterministic failure when debugging concurrent programs.

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

Chapter 4

Results

The results of the phenomenographic part of the study take the form of sets of phenomenographic outcome spaces describing the different ways in which the interviewed students understood the purpose of the programming task, developing and testing a program, the sources of failure involved in the programming task and the tuple space involved in the assignment. It is important to note that a student may, and indeed should have, many different understandings of each of the aforementioned phenomena.

In this chapter, I will only present the phenomenographic results briefly; for detailed explanations of each category including quotes of the students' descriptions, see the original articles [P1,P2]. The phenomenographic results are divided into three parts: how students understand tuple spaces (Subsection 4.1), how they understand the goal of the programming task (Subsection 4.2) and how they understand developing a program (Subsection 4.3). After that, I describe the defects found in the students' programs in Subsection 4.4.

4.1 Students' understandings of tuple spaces

Tuple spaces were described by the interviewees in a number of different ways [P1] 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 4.1.

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.

The tuple space categories span a large portion of the revised Bloom taxonomy for CS by Johnson and Fuller [35]. The lowest level of this taxonomy is *knowledge*; remembering facts. In our programming assignment, the specification for the tuple space can be considered an example of this as it is presented as a fact by the textbook and the teaching staff. Above knowledge in Bloom's taxonomy is *comprehension*, followed by *application*. Usage of a tuple space is a form of application. Above application comes *analysis* (which arguably is part of implementing a tuple space), *evaluation* (which corresponds to our category of the same name) and *synthesis* (which programming itself arguably is). The revised taxonomy adds *higher application* at the top of the taxonomy; implementation and usage informed by

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 4.1: Categories of tuple spaces (from [P1])

evaluation could be considered to belong here.

The fact that the implementation and use of tuple spaces in the solution was mandatory in the assignment and that the assignment therefore essentially consists of several small implementation tasks and is not conducive to synthesis nor evaluation makes the appearance of the latter somewhat surprising. The question of whether the assignments should be more open-ended to encourage analysis, synthesis and evaluation is raised; this is a question that should be considered when determining the learning goals of the course.

4.2 Students' understandings of the goal of program development

The interviewees express the purpose of a programming assignment in different ways shown in Table 4.2, 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. [P2]

In order for a program to be useful, it must interact with other entities. Distributed programs such as the chat system in this assignment rely on multiple nodes joined by a communication network and are therefore susceptible to communication breakdowns. All but the most isolated programs must also interact with other software, especially libraries and operating systems. Finally, many programs must interact with users. As real-world entities tend to be imperfect, risks stemming from these entities can be found that may cause the program to fail. The students mentioned three types of failure sources: other systems (as expected, network failure was mentioned as a possible source), the user and the programmer. They are summarised in Table 4.3.

This list of failure sources is closely related to the outcome space of purposes of the

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 4.2: Purposes of the programming task (from [P2])

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

Table 4.3: Sources of failure (from [P2])

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 and avoid making it too time-consuming for the students, the course staff explicitly told the students that they could ignore the possibility of network failure in the assignment. Nonetheless, it is one of the sources of failure taken into account by the students. Similarly, while taking the possibility of user error into account is important in many real-life programs, 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 these cases, students have gone beyond the assignment's requirements into "real-world" scenarios. The course staff did, however, expect that students would try to minimise the consequences of programmer error by sticking to simple solutions, since this both decreases development time and improves the chances of meeting the assignment deadline imposed by the course staff.

This suggests that teachers should not discourage students from taking system and user failure into account. It is easy for students to test how their program handles user error; indeed, they may do so accidentally. However, generating failures in underlying systems such as networks is harder and is an area in which it may be useful to provide additional tools to students that, for example, allow network connections to be interrupted at will to test how such a situation is handled.

The students' aims in a project course in computer systems have been explored by Berglund and Eckerdal [10]. These findings show similarities with the purpose of the programming task discussed here, as they encompass both requirements set by the university and an

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 4.4: Categories of developing and debugging (from [P1])

environment extending the formal requirements, looking toward a professional life. There is also a similarity to the categories of what it means to learn to program of Eckerdal and Berglund [20] in that programming is seen as a problem solving technique and a skill usable outside the programming course.

The different ways in which the students understand the purposes of the programming task are also related to the concept of *relative correctness* of Ben-David Kolikant [7] in that students have different understandings of what the program is supposed to be; with the important distinction that relative correctness may involve accepting failure ("The program is correct but it is not finished."), while the purposes of the programming task are alternative interpretations of the goal in the form of in which context the program works as intended.

These understandings of the assignment purpose are also reminiscent of the motivations for working on programming tasks found by Carbone et al. [16] among first-year students learning programming. Their achieving motivation corresponds to the students here viewing the programming task as an assignment, while intrinsic motivation is a very plausible explanation for going beyond the requirements of the assignment.

4.3 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 (see [P1]). Each category differs from the previous in that it moves further from the actual program code toward a human-oriented "big picture". This is summarised in Table 4.4.

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, in which formulating an implementation in a programming language and testing and debugging this implementation are essential. The solving technical problems category can be seen as design. The specification the students

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 4.5: Software development process models (from [P2])

are provided with is more or less a finished architecture design. Hence, students need not do any design other than module design (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 Section 4.2. 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. It can become a hindrance in programming assignments such as the one in this study when it distracts students from the intended goal of producing a reliable implementation of a specification or if the student in a later professional software development context chooses to ignore specifications in favour of his own interpretation of the requirements for the program. In this case, the expectation that the the student or programmer does not deviate from the specification should be made clear. However, it may be useful for programming teachers to consider whether 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 (see [P2]). The process understandings are summarised in Table 4.5.

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

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 4.6: Testing approaches (from [P2])

accepted ways to find a solution to a programming problem.

In a study of novice programmers, Booth [11] showed that students approach the task of programming in different ways, varying from cut-and-paste solutions (labelled "Expedient" by Booth) to developing solutions in a structured way, focusing on the problem domain (labelled "Structural"). The results concerning the development process in this project are similar. The differences can be sought in factors such as the differing subject areas, the experience of the students and the development of computer science in the past 16 years.

Carbone et al. [16] describe several skill deficiencies exhibited by the students they interviewed. The inability to break down a programming task they find is similar to how our students do not find a design necessary. However, in our case, the students used this approach successfully to produce an inefficient and naïve but functional implementation of a tuple space, which is neither unexpected nor problematic. The ineffective tinkering and lack of problem solving skills found by Carbone et al. correspond to our trial and error category (and possibly coding to understand), showing that increased experience unfortunately does not always lead to improvements in necessary skills. While it is expected that novice programmers have inadequate problem-solving skills, it is expected of students close to getting a Master's degree that they be able to apply commonly-used problem-solving strategies.

The students I interviewed similarly had approaches to testing that reflected different understandings of the intent of testing (see [P2]). These are shown in Table 4.6. The testing

approaches range from the superficial and, by the students' own admissions, inadequate, to testing with increasing degrees of purposefulness, 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.

4.4 Defects in students' concurrent programs

As explained in Chapter 1 and Section 3.2, this thesis consists of an analysis of students' understandings of concurrent programming and an analysis of their errors in light of their understandings. In this section, I describe the results of the latter. The defect classifications are formed as described in Subsection 3.4.2.

Initially, I performed the analysis of defects using only the students' programs and reports as data and constructed a classification schema based on the assessment criteria of the Concurrent Programming course at the time and on defect classifications found in the literature, especially the classification of Eisenstadt [22]. The results of this analysis can be found in [43]. The top level of classification in that analysis was a division into:

Concurrency errors Misconceptions or design errors related to concurrency

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

One problem with this classification was that only a small amount of the students' errors could be unambiguously placed in one of the above categories; only 23 %, 45 % and 34 % for the respective assignments. This was because asking students to explain the reasoning behind their entire solution in a written report did not give enough information to reconstruct their errors. Another reason was that some errors can fit into many classes.

As described in Subsection 3.4.2, 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 phenomenographic outcomes spaces of the previous sections of this chapter led to some changes to the classification. While it would be possible to distinguish between errors made in designing the solution and implementing it, students did not consistently make this distinction, as seen in Table 4.5. For this reason, it is hard in some cases and not very useful to make this distinction. The distinction between concurrency and general programming errors is similarly ignored. One reason is that, 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. Another reason is that the phenomenographic study did not show that students make this distinction. Some did, however, show an awareness of the difference between deterministic and nondeterministic errors, as seen in Table 4.6. Table 4.5 also shows that understanding the

requirements of the assignment can be seen as a source of difficulties that is great enough to structure one's work around. In Tables 4.2 and 4.3, 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.

Requirement-related error A programmer can fail to understand part of a specification correctly or fail to take it into account properly when designing or implementing his solution. Some understandings of the goals of a programming task (e.g. seeing a passing grade as the goal of a programming assignment) can lead to this. Pointing out 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 Some 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, such as concurrent programming.

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 of the assignment being confusing, not lack of any understanding or skill relevant to concurrent programming in general. 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. Since some students do not have a clear distinction between creating an algorithm and an implementation, these are grouped together. 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

	2005	2007	2008
Submissions	128	60	52
Requirement	53	10	11
Programming	3	0	0
Assignment	70	20	10
Incorrect	28	16	5
Deterministic	39	2	0
Nondeterministic	115	44	26
Total	154	46	26

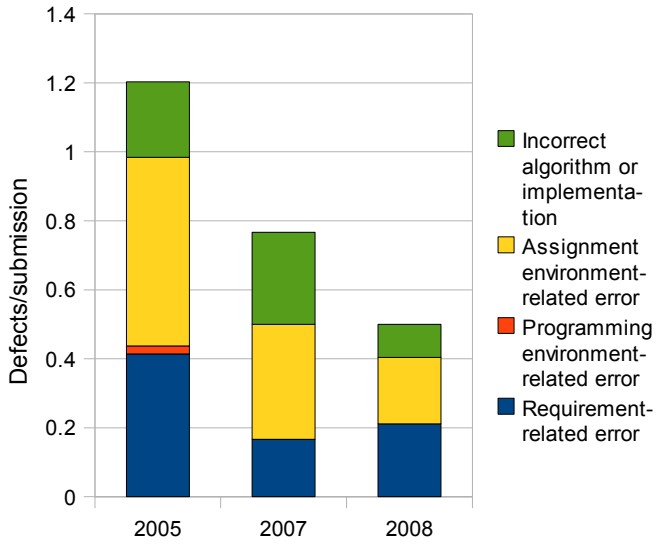

Figure 4.1: Defects found in Trains assignment

common, surprising or illustrative of students' understandings of concurrent programming. A more detailed list of defects can be found in Appendix A in the form of a summary of the fine-grained classification used for assessment in 2007 and 2008.

4.4.1 Trains

An interesting aspect of the Trains assignment (described in Subsection 3.1.1) is that, since the train simulation combined with the student's train control code takes no input from outside, almost all failures are nondeterministic; a deterministic failure would occur in every possible execution, making it easy to detect by simply letting the trains run once around the track. It is therefore not surprising that all the deterministic failures are due to misunderstandings of the requirements. Since the concurrent programming aspect of the assignment is easy in comparison to the other assignments, it is hardly surprising that most of the students' errors are related to the simulator and what they are supposed to do with it. Figure 4.1 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.

Requirement-related errors

The train simulator used in the Trains assignment proved to have some confusing aspects in its original form used in 2005. Particularly problematic was that the 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, such as shared variables. This allowed students to avoid much of the expected semaphore usage. The assignment also required students to implement the required random delay at stations themselves, which in many cases was replaced by a fixed delay. These problems were eliminated in the 2006 version of the assignment by redesigning the simu-

lator and its API so that the options available to the student in the simulation environment matched the requirements.

After this, the most common form of requirement-related error (accounting for almost all of the requirement-related errors in 2008) is that at least one train uses the secondary choice for a track or station platform even when the primary choice was free, ignoring the requirement to use the upper platform or shorter track where possible. This requirement exists to prevent statically allocating one alternative to each train, removing the need for choosing between alternative tracks. However, it is vague, hard to test for (our test package does not detect it) and overlooked by a few students every year.

Programming environment-related errors

In three cases in 2005, students had clear misunderstandings of the Java language or standard API, such as accidentally generating negative random numbers or failing to understand the purpose of the `break` statement at the end of a `case` of a `switch`. Before 2004, introductory programming was taught at TKK using Scheme instead of Java, so some students may have been unfamiliar with Java.

Assignment environment-related errors

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. Some of the rules of the simulation were also not obvious to the students.

By far the most common type of error here was placing the sensors used to release a track segment too near a switch, allowing the other train to enter or change the switch before the first has left. This type of error has decreased since 2005, probably because the simulator and its documentation have been revised for clarity several times. Other sensor-related issues, ignoring the crossroads at the top of the track and setting the trains' speed too low account for the rest of the errors in this category.

Incorrect algorithm or implementation

Almost all the solutions were close enough to being correct for specific problems to be identifiable. Most of the errors were found in the train segment reservation code. Some solutions consisted of subsolutions that did not combine properly or relied on train events happening in a specific order. Others had more localised problems, such as changing switches at the wrong time or not at all, using the wrong semaphore or the right semaphore at the wrong time or initialising a semaphore to the wrong value. 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.

Only a few errors were obvious implementation slips, such as forgetting a `break` or `else`, matching sensors incorrectly, parenthesising a logical expression wrong, making an array one element too small or accidentally duplicating or commenting out code.

	2005	2007	2008
Submissions	107	51	40
Requirement	93	112	38
Programming	15	11	1
Incorrect	51	56	17
Deterministic	94	102	8
Nondeterministic	65	77	49
Total	159	179	57

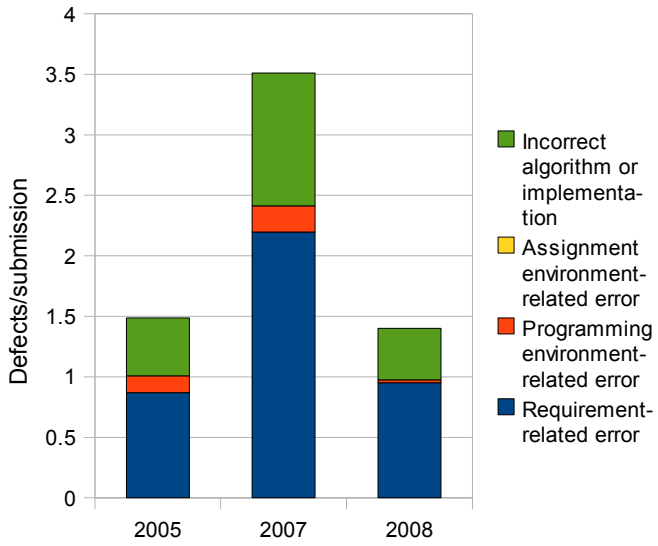

Figure 4.2: Defects found in Reactor assignment

4.4.2 Reactor

The Reactor design pattern in the second assignment (see Subsection 3.1.1) turned out to be hard to understand for some students; in many cases, the students' programs are correct solutions to what they consider to be the problem. Clarifying the intent and structure of the Reactor pattern was clearly necessary, so I wrote a simplified explanation of the Reactor pattern for the next year's course.

The defects found are summarised in Figure 4.2. The increase in defects between 2005 and 2007 can be mostly ascribed to more aspects of the programs being taken into account in assessment, such as memory use.

Requirement-related errors

Until students were provided with a test package 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 package was introduced.

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. This error by itself accounts for more than three quarters of the requirements issues found in 2008. The fact that it remained common in 2008 is probably due to the fact that the test package does not include a test case for this scenario.

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 [59] 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()`. Using busy waiting or polling in the Reactor or Hangman and failure to terminate properly accounts for the remaining cases.

Programming environment-related errors

In 2005, the console I/O required by the Hangman client was by far the most problematic aspect of the programming environment. The client was deemed unnecessary and removed the next year. Several cases of using a fixed TCP port number when required to use a free one as shown in the example code have been found.

Four cases in 2005 were due to misconceptions about Java.

Incorrect algorithm or implementation

Many 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, like the previous error, due to improved assessment guidelines. Again, the testing package makes this type of error easier to detect.

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.

A few obvious implementation slips were found, such as having the Hangman server and client connected to different ports, starting a thread twice, declaring an array that was one element too small and using a stack instead of a queue.

4.4.3 Tuple space

In the tuple space assignment (described in Subsection 3.1.1), the requirements of the assignment were once again problematic in the 2005 original. However, many of the defects found were clear indications of careless or unskilled concurrent programming. By this time, the Java programming environment was apparently familiar to the students, as no clear misunderstandings of the programming environment were found. The defects found are summarised in Figure 4.3.

Requirement-related errors

As in the first 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; making the corresponding error in later years and causing failures in the distributed context was much less common. There were fewer problems with the division between tuple space and chat system than between Reactor and Hangman. Polling occurred in a few cases in either the chat system or tuple space.

The most commonly ignored requirement of the chat system's functionality was that messages stay in order. As an example of the variety of other errors of this type, a few students in 2005 and 2007 allowed their chat system to combine messages stored in the

	2005	2007	2008
Submissions	84	49	39
Requirement	93	49	21
Assignment	3	0	0
Incorrect	70	51	36
Deterministic	98	58	28
Nondeterministic	68	42	29
Total	166	100	57

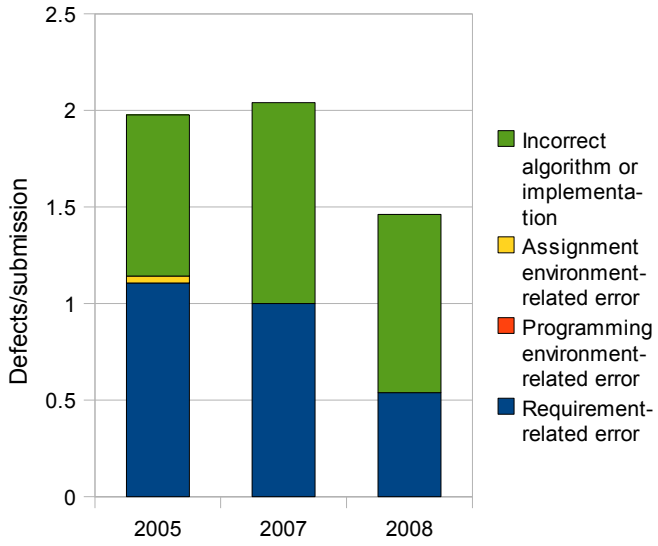

Figure 4.3: Defects found in Tuple space assignment

log for delivery to new listeners into one message that looked the same to the user of the provided GUI (a clear example of ignoring the specification as long as the user experience is the same). Yet again, the test package seems to help students understand they have a problem.

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.

Assignment environment-related errors

The GUI provided to the students to make the requirements easier to understand sends messages when listeners leave (and, in 2005, when they join) a channel; this caused some students to require this behaviour for their implementation to work.

Incorrect algorithm or implementation

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.

Cleaning up after a handle is removed for use 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.

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 grind to a halt 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.

4.4.4 Summary

In this section, I've described the defects in the programs students have written as solutions for the programming assignments in the Concurrent Programming course. The two largest classes of defect are related to understanding requirements and incorrect algorithms and implementation. The assignment environment in the Trains assignment is misunderstood by many students. Improvements to the assignment descriptions, the train simulator and providing students with a packages of tests for their programs are probably the reasons for a noticeable decrease in defects over the years, particularly misunderstandings of requirements and the assignment environment.

Chapter 5

Conclusions

In this chapter, I explain how the results described in this thesis can be used to encourage more useful understandings in students through changes to teaching and assignments as well as through software tools. In other words, this chapter describes how teachers and computer science education researchers in the context of concurrent programming can make use of the results of this thesis to achieve their goal: helping students learn.

In general, the results of this study suggest that students have a large variety of understandings of many concepts in the field of concurrent programming. These understandings range from simple and incomplete understandings that can easily lead to problems for the student when developing software to the advanced and multifaceted understandings expected of software development professionals. Many understandings of the course itself and what is expected from students in e.g. programming assignments are also problematic.

5.1 Understanding program execution

Based on what has been uncovered in this study, it seems that students have problems understanding what happens when a program is executed. Some students saw the programming assignment as an ideal problem, in which many limitations of real-life programming, such as finite memory or network delays, do not apply. Many students introduce defects in their programs that appear to be caused by misunderstanding or reasoning incorrectly about concurrent program execution.

Part of the problem is that the runtime behaviour of a concurrent program, a necessary part of the programmer's perspective, is hard to examine or interpret, preventing students from effectively understanding what their program does and reasoning in terms of the relevant concurrency model. Another possible problem is that the models of concurrency used in textbooks such as those by Andrews [2] and Ben-Ari [4] do not match the concurrency model of e.g. Java [26] in all the relevant aspects. For example, Java allows compilers and multiprocessor architectures to reorder operations within a thread as long as all the operations *within this thread* produce the same result. This means that other threads may read combinations of values of variables that are impossible in textbook concurrency models. To address this, I suggest a greater emphasis in teaching concurrent programming on real-world concurrency models than the aforementioned textbook models. In order to understand how their programs fail, they should be shown how their programs really behave so that they can realise that their understanding of concurrency is incomplete and correct

them.

In the case of students ignoring memory use, giving them tools to study memory allocation would help them understand how their programs use (or misuse) memory. In its most basic form, this could involve using a profiler to get information on the maximum memory use of their program. More detailed visualisations, such as charts that show memory use over time categorised by where the memory is allocated, can be used to help students understand memory use in more detail. Other resource usage issues, such as use of CPU time or network or disk capacity, can be addressed using similar visualisations.

5.2 Communicating goals

Students may also have a different understanding of what they are trying to achieve than their teachers. Many of the students in this study wrote programs that were missing required functionality or implemented this functionality in ways that conflicted with requirements or required additional limitations on the runtime environment. One reason we found for this was that students had different aims in their assignment, seeing it primarily as something they have to do to get a grade or as an ideal problem in an ideal context in which simplifying assumptions apply. The students also considered different potential sources of problems: the hypothetical user of the program (even when the assignment was specified in terms of the input and output of methods, not user requirements), underlying systems that could fail, especially network connections in a distributed system, and the programmer (the student) as a error-prone human.

The purposes of the programming task and sources of failure described in this thesis suggest that many of the errors made by students are misunderstandings of what their program is supposed to do and what situations it is expected to cope with rather than actual misunderstandings of concurrent programming itself. It is hard for a student to discover such problems by himself if all he has to go on is a specification in natural language that is open for several different forms of misinterpretation (such as ambiguity in the text or simply misreading it). The large amount of requirement-related errors in all three assignments and the way they decreased in the second and third assignment when the test package was provided to the students supports this theory. Assuming that it is desirable to have clearly-defined and specific goals (which is useful in guiding students' learning and simplifies assessment), this suggests that teachers should make goals more explicit and concrete. One important aspect is that the goals should specify *what* the student should achieve rather than *how*, allowing students to find their own solutions to problems; rather than seemingly arbitrarily forbidding the use of monitors, students should be working in an environment where monitors do not work. Students should also be provided with ways to explore problems related to these goals. The student should see his or her program clearly fail to work correctly rather than be told afterwards that he or she did something the wrong way or failed to take a usage scenario into account.

Two different types of measures have been taken on our Concurrent Programming course to address these issues. One was to change the environments in which several of the assignments were done to make limitations more concrete, such as actually making the Trains and Tuple space assignments function as distributed systems (in the form of separate processes) rather than as threads within one virtual machine. The other major change was made after several students each year requested that they be given access to the package of

tests for the assignments used by the course staff to support assessment. Giving the students a test package that clearly states whether their solution fulfils the specification's demands appears to have decreased the amount of errors even in assignments where students had easy acccess to tests, such as Trains. This supports the theory that students sometimes fail to find defects in their programs because they do not recognise incorrect behaviour when they see it, either because they do not check the results of their program in sufficient detail or do not know what the expected behaviour is.

5.3 Verification

The students in this study have a wide range of approaches to testing. Some students used completely unplanned, cursory, testing. Some tried to 'break' the system (e.g. through stress testing), while others covered a variety of different cases. Moreover, some students found they cannot test their program adequately by themselves and need help from another person or tool, that testing in itself is not sufficient or that you have to prove your program correct by hand. In particular, the large amount of defects found in students' programs that cause failures nondeterministically suggest that testing software that helps make these defects manifest will help students find their defects by themselves. The results of the Reactor assignment seem to bear this out. However, no such dramatic improvement can be seen in the Tuple space assignment, probably because the students could not debug their programs despite knowing that they contain bugs.

The students' verification approaches could be improved by providing testing tools to generate scenarios that are hard to discover using normal testing procedures and more explicit and detailed guidance on how to apply different verification techniques in practice. The assignment itself could be changed to encourage students to learn and apply different verification techniques by explicitly requiring models, as done by Brabrand [13], or by requiring students to create suitable tests, e.g. using test-driven development.

Adapting programs to a model that can be checked using a model checker is often hard and error-prone work. This makes this approach especially impractical for students to use in an assignment unless the modelling of their solution is in itself a goal of the assignment as in Brabrand's course above or the assignment is carefully designed to facilitate efficient model checking.

An alternative approach to finding concurrency bugs is to increase the chance of interleavings that lead to failure. Stress testing is a well-known approach, and its usefulness can be further improved by making sure interleavings occur often and in many places. One straightforward and realistic approach is to distribute the program's threads over multiple processors. Another way to do this is to automatically and randomly change the thread scheduling to make concurrency failures more likely to occur (e.g. [68]). This is the approach used by the automated testing system of our concurrent programming course to increase test effectiveness.

5.4 Understanding program failure

Based on the empirical results described herein, I suggest that what students need to effectively understand what their concurrent programs do is a tool to generate execution history

visualisations automatically from a running program that are easy to understand and navigate and provide the information needed by the student in an easily understandable form.

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.

There are several indications that students understand their program in terms of constructs of a higher level of abstraction than those in the Java code. One is that students see developing a program as solving algorithmic problems rather than straightforward implementation. Another is that they understand tuple spaces at higher levels of abstraction than their actual implementation. This suggests that program visualisation tools should allow users to choose a level of abstraction by grouping together parts of the code or execution to correspond to their understandings, similarly to the ability to change between program- and algorithm-level behaviour suggested by Price et al. [56]. The tool could then visualise the behaviour of the program in a fashion closer to the student's view. For example, if students understand their programs as sets of communicating entities, the tool should be able to show them the communication between these entities and the relevant aspects of their state even though this state may be spread out over several objects and part of the communication is implicit in locking mechanisms.

Based on the empirical work described in this thesis, I constructed a systematic way of presenting feedback to students that explains defects found in their programs:

1. An execution of the program which fails. This can be automatically generated by testing or model checking and shown using the visualisation described above. The requirements of the assignment should be such that not adhering to them causes the program to fail in some situation the student can reconstruct. If the failure is incorrect behaviour (e.g. output), show it and the sequence of events leading up to it, focusing on the information relevant to the student understanding the failure. If the failure is resource overuse (e.g. memory), show how the resource is used (when and where).

2. A description of the defect that causes the failure. This can be expressed as a change to the code that eliminates the defect. Apart from failing executions, possible defects can in many cases be automatically listed based on empirical information on defects and the failures they result in. However, determining the exact defect will in most cases involve manual debugging work.

3. The underlying error. Using empirical information on the reasoning behind similar defects, and available information on the student's reasoning (e.g. documentation, comments, structure of code), a teacher can describe what he or she thinks the error is.

4. A teacher can try to determine what the student has not understood well enough and explain it.

5. Suggestions for how to detect similar problems: verification strategies effective against this type of defect and design strategies that avoid introducing them.

An important open educational question is raised by this: How much help should students get in finding their programming errors? On the one hand, helping students correct

their misunderstandings is an important facet of teaching. On the other, students should be given a chance to develop debugging skills. On the third and somewhat unrealistic hand, students should be taught how to systematically produce correct code without producing and debugging incorrect code.

5.5 Future work

In this section I explain my plans for future research into tools for concurrent programming based on the results of this study. While the context of this study is a concurrent programming course, many of these tools may be useful in or easily adapted to other programming contexts.

5.5.1 Generating and recording failing executions

Ensuring that the process interleavings that lead to failure are generated is difficult when concurrency is involved. The amount of defects found in our students' programs that may or may not manifest as failure depending on how processes interleave indicates the need for verification techniques that do not rely on program behaviour being deterministic. One approach, which is used in the test package for our Concurrent Programming course and by e.g. Stoller [68], is to introduce frequent and random changes in the scheduling of a Java program's threads to increase the amount of different thread interleavings and hence improve the chances of thread interleaving. Evaluating the effectiveness of this technique in the context of concurrent programming assignments will help determine whether it is an appropriate solution or whether an alternative should be sought. Some additional work is also necessary to extract the operations executed in a program, especially if the program is concurrent. Replay algorithms such as that of Choi et al. [18] can be used to store the execution sequence of a program for later analysis. Adapting such an algorithm to produce the information necessary for debugging is necessary for the planned work on visualisation described in Subsection 5.5.2.

While applying model checking directly to existing programming assignments is not practical, a combination of different changes to programming assignments may make it practical to apply model checking to them. One is to change the assignment to reduce the resulting programs to, for example, self-contained Java programs that can easily be checked by Java PathFinder. Another is to require the students to model their solution themselves. Using a model checker would make it easy to ensure that all possible interleavings of operations are covered and also makes the extraction of detailed information on execution order very easy.

5.5.2 Visualising execution history

Understanding what happens in a program, especially for debugging purposes, can be simplified by different visualisation and interaction techniques. Showing the user the executed instructions helps the user understand what the program is doing. In particular, showing the user the sequence of instructions that led to an unexpected event can be very useful; studies show that programmers often require information on the causes of an event and how different events are interconnected when looking for hard-to-find bugs [22, 47]. Concurrency can also make it very hard to trace the cause of an unexpected event.

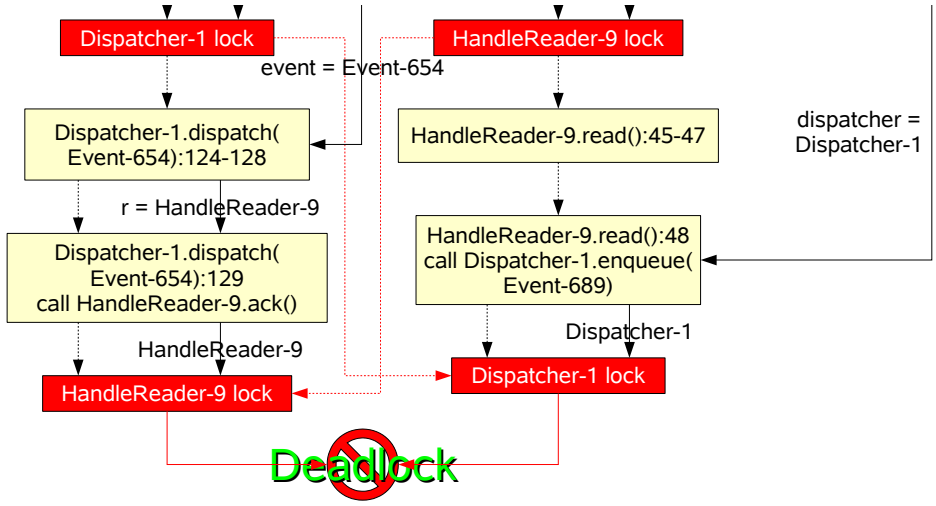

Figure 5.1: Example of dynamic dependence graph view of a concurrent program failure

My research in this area will entail generating execution history visualisations automatically from a running program, designing user interfaces around these visualisations that are easy to understand and navigate and ways to produce visualisations that provide the information needed in a specific debugging scenario in an easily understandable form. In particular, answering queries about the reasons for events and states in a program is a promising new idea of this type that as yet has not been developed to a level at which it can be used in software development. Dynamic dependence graphs (DDGs) can be used for visualisations that aid in finding and understanding cause-effect chains, answering this type of query, and have been found useful in some types of debugging situations in educational visual programming environments [38]. DDGs are of particular interest for concurrent programs, as interactions between threads (e.g. locking operations, reading variables, message passing) are clearly shown as edges. For this reason, I will focus on developing DDGs, although I may add other execution history visualisations (call trees, sequence diagrams, collaboration diagrams) as supplements for navigation and understanding. Traditional visualisations such as sequence or collaboration diagrams are obvious approaches to doing this, and dynamic dependence graphs are a promising new addition. An example of such a graph is shown in Figure 5.1.

5.5.3 Visualising the current state

In current visual debuggers, the variables and data structures in the running program are shown graphically. This view can be extended (similarly to UML collaboration diagrams [52]) to show, in addition to executed operations, locking interactions between threads and objects in e.g. Java, making it easier to see which operations have been executed by different threads on an object.

5.5.4 Summary

This work provides the empirical background on which I intend to base the development of visualisations for concurrent programs. Students introduce many defects in their programs that appear to be caused by misunderstanding or reasoning incorrectly about concurrent program execution; clearing up such misunderstandings is a goal of the visualisations. Understanding how students understand concurrent programming is also a great help in constructing visualisations to be easily understood by students.

References

[1] Kirsti Ala-Mutka. A survey of automated assessment approaches for programming assignments. *Computer Science Education*, volume 15, number 2, pages 83–102, 2005.

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

[3] Boris Beizer. *Software Testing Techniques*. International Thomson Computer Press, 2 edition, 1990. ISBN 1850328803.

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

[5] Mordechai Ben-Ari and Yifat Ben-David Kolikant. Thinking parallel: The process of learning concurrency. In *Fourth SIGCSE Conference on Innovation and Technology in Computer Science Education*, pages 13–16, Cracow, Poland. 1999.

[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*, volume 23, number 1, pages 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] Anders Berglund. Phenomenography as a way to research learning in computing. *Bulletin of Applied Computing and Information Technology*, volume 4, number 1, July 2006.

[9] 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. Australian Computer Society, 2008.

[10] Anders Berglund and Anna Eckerdal. What do our students strive for? Insights from a distributed, project-based course in computer systems. In *Proceedings of 5th Annual Finnish/Baltic Sea Conference on Computer Science Education*, pages 65–72, 2005.

[11] Shirley Booth. *Learning to program: A phenomenographic perspective*. Acta Universitatis Gothoburgensis, doctoral dissertation, University of Gothenburg, Sweden, 1992.

[12] Jonas Boustedt. *Students working with a Large Software System: Experiences and Understandings*. Licentiate thesis, Uppsala University, May 2007.

[13] Claus Brabrand. Constructive alignment for teaching model-based design for concurrency. In *Proc. 2nd Workshop on Teaching Concurrency (TeaConc '07)*, Siedlce, Poland. June 2007.

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

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

[16] Angela Carbone, John Hurst, Ian Mitchell and Dick Gunstone. An exploration of internal factors influencing student learning of programming. In Margaret Hamilton and Tony Clear, editors, *Proceedings of the Eleventh Australasian Computing Education Conference (ACE2009)*, volume 95 of *Conferences in Research and Practice in Information Technology*, pages 25–34, Wellington, New Zealand. Australian Computer Society, 2009.

[17] Nicholas Carriero and David Gelernter. How to write parallel programs: a guide to the perplexed. *ACM Computing Surveys*, volume 21, number 3, pages 323–357, 1989. ISSN 0360-0300.

[18] 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*, page 10 pp, San Fransisco, USA. IEEE Computer Society, April 2001.

[19] Michael Clancy. Misconceptions and attitudes that interfere with learning to program. In *Computer Science Education Research* Fincher and Petre [23].

[20] Anna Eckerdal and Anders Berglund. What does it take to learn 'programming thinking'? In *Proceedings of The First International Computing Education Research Workshop*, pages 135–142, 2005.

[21] Anna Eckerdal and Michael Thuné. Novice Java programmers' conceptions of "object" and "class", and variation theory. *SIGCSE Bulletin*, volume 37, number 3, pages 89–93, 2005. ISSN 0097-8418.

[22] Marc Eisenstadt. My hairiest bug war stories. *Communications of the ACM*, volume 40, number 4, pages 30–37, 1997. ISSN 0001-0782.

[23] Sally Fincher and Marian Petre. *Computer Science Education Research*. Routledge, 2004.

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

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

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

[27] 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.

[28] Christian Herzog. From elementary knowledge schemes towards heuristic expertise — designing an ITS in the field of parallel programming. In Claude Frasson, Gilles Gauthier and Gordon I. McCalla, editors, *Proceedings of 2nd International Conference on Intelligent Tutoring Systems*, volume 608 of *LNCS*, pages 183–190. Springer, June 1992.

[29] Colin Higgins, Pavlos Symeonidis and Athanasios Tsintsifas. The marking system for CourseMaster. In *Proceedings of the 7th annual conference on Innovation and Technology in Computer Science Education*, pages 46–50. ACM Press, 2002. ISBN 1-58113-499-1.

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

[31] Connor Hughes, Jim Buckley, Chris Exton and Des O'Carroll. Towards a framework for characterising concurrent comprehension. *Computer Science Education*, volume 15, number 1, pages 7–24, March 2005.

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

[33] IEEE. IEEE standard classification for software anomalies. Technical Report Std 1044-1993, IEEE, 1994.

[34] IEEE. The single UNIX specification, version 3. Technical Report Std 1003.1-2004, IEEE, 2004.

[35] Colin G. Johnson and Ursula Fuller. Is Bloom's taxonomy appropriate for computer science? In Anders Berglund and Mattias Wiggberg, editors, *Proceedings of 6th Baltic Sea Conference on Computing Education Research, Koli Calling*, pages 120–123. Uppsala University, 2006.

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

[37] Donald Ervin Knuth. The errors of TeX. *Software — Practice and Experience*, volume 19, number 7, pages 607–685, 1989.

[38] 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.

[39] 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*, volume 16, number 1-2, pages 41–84, 2005.

[40] Eileen Kraemer. Visualizing concurrent programs. In *Software Visualization: Programming as a Multimedia Experience* Stasko et al. [66], chapter 17, pages 237–256. ISBN 0-262-19395-7.

[41] Doug Lea, editor. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, second edition, 1999.

[42] 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.

[43] 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. Uppsala University, 2007.

[44] Michael Luck and Mike Joy. A secure on-line submission system. *Software - Practice and Experience*, volume 29, number 8, pages 721–740, 1999.

[45] Ference Marton. Phenomenography — describing conceptions of the world around us. *Instructional science*, volume 10, number 2, pages 177–200, 1981.

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

[47] 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. ACM Press, 1997.

[48] 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. Springer-Verlag, 2002.

[49] Sougata Mukherjea and John T. Stasko. Applying algorithm animation techniques for program tracing, debugging, and understanding. In *Proceedings of the 15th international conference on Software Engineering*, pages 456–465. IEEE Computer Society Press, 1993.

[50] Sougata Mukherjea and John T. Stasko. Toward visual debugging: integrating algorithm animation capabilities within a source-level debugger. *ACM Transactions on Computer-Human Interaction (TOCHI)*, volume 1, number 3, pages 215–244, 1994.

[51] Glenford J. Myers, Tom Badgett, Todd M. Thomas and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, second edition, 2004.

[52] *Unified Modeling Language (UML), version 1.5*. Object Management Group, 2003.

[53] 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. Springer-Verlag, 2002.

[54] Arnold Pears, Stephen Seidman, Crystal Eney, Päivi Kinnunen and Lauri Malmi. Constructing a core literature for computing education research. *SIGCSE Bulletin*, volume 37, number 4, pages 152–161, 2005. ISSN 0097-8418.

[55] Arnold Neville Pears. Using the DiST simulator to teach parallel computing concepts. In *International Forum on Parallel Computing Curricula*, Wellesley, Massachusetts. 1995.

[56] Blaine A. Price, Ronald M. Baecker and Ian S. Small. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, volume 4, number 3, pages 211–266, 1993.

[57] Daniela Rothenhöfer and Christian Herzog. SYPROS — an intelligent tutoring system for parallel programming. In Chan Tak-Wai, editor, *Proceedings of 1993 International Conference on Computers in Education*, pages 300–305, Taiwan. December 1993.

[58] RTI. The economic impacts of inadequate infrastructure for software testing. Planning report 02-3, NIST, May 2002.

[59] 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.

[60] Judy Sheard, Simon, Margaret Hamilton and Jan Lönnberg. Analysis of research into the teaching and learning of programming. Submitted for review.

[61] Simon. A classification of recent Australasian computing education publications. *Computer Science Education*, volume 17, number 3, pages 155–169, September 2007.

[62] Simon. Classifying computing education papers: Process and results. In *ICER '08: Proceedings of the ACM Workshop on International Computing Education Research*, Sydney, Australia. ACM, September 2008.

[63] Simon. Koli Calling comes of age: an analysis. In Raymond Lister and Simon, editors, *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007)*, volume 88 of *Conferences in Research and Practice in Information Technology*, pages 119–126, Koli, Finland. Australian Computer Society, 2008.

[64] Juha Sorva. Students' understandings of storing objects. In Raymond Lister and Simon, editors, *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007)*, volume 88 of *Conferences in Research and Practice in Information Technology*, pages 127–135, Koli, Finland. Australian Computer Society, 2008.

[65] James C. Spohrer and Elliot Soloway. Novice mistakes: are the folk wisdoms correct? *Communications of the ACM*, volume 29, number 7, pages 624–632, 1986. ISSN 0001-0782.

[66] John T. Stasko, John B. Domingue, Marc H. Brown and Blaine A. Price. *Software Visualization: Programming as a Multimedia Experience*. MIT Press, Cambridge, MA, 1998. ISBN 0-262-19395-7.

[67] 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.

[68] 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.

[69] *Java Platform, Standard Edition 6 API Specification*. Sun Microsystems, 2008.

[70] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park and Flavio Lerda. Model checking programs. *Automated Software Engineering Journal*, volume 10, number 2, pages 203–232, April 2003.

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

Appendix A

Lists of defects found in assignments

In the following, the different classes of defects found in the assignments of the Concurrent Programming course at TKK in 2005, 2007 and 2008 are listed together with the amount of instances of each class. Where a type of defect is not considered a defect in a particular year or the defect occurs only in code written to meet a requirement that did not exist that year, "N/A" is used instead of "0" to distinguish this from the case where students could have had defects belonging to this type but did not.

A.1 Defects in Trains assignment

Name	2005	2007	2008
Requirement	53	10	11
Delay at station not random	16	N/A	N/A
Trains not restarted at station	0	2	0
Less preferred choice used when both available	11	7	10
Forbidden simulation information used	16	N/A	N/A
Forbidden interprocess communication used	7	N/A	N/A
Polling or busy-waiting used	3	1	1
Programming environment	3	0	0
Fall-through `switch` not understood	1	0	0
Random numbers assumed to be non-negative	2	N/A	N/A
Assignment environment	70	20	10
Crossing unprotected	8	2	1
Release sensor too close	47	12	4
Reservation sensor too close	9	5	2
Identification of sensor event wrong	4	1	2
Trains move too slowly	2	0	1
Incorrect algorithm/implementation	28	16	5
Algorithm incomplete or incomprehensible	1	2	0
Conflicting subsolutions	4	0	0
Does not handle delays in initial start	1	1	3
Switches used incorrectly	3	2	1
Try to acquire then release and reacquire	1	0	0
Try to acquire to see what to release	1	1	0
Try to acquire both alternatives	1	2	0
Misplaced acquire operation	1	3	0
Misplaced, missing or incorrect release operation	6	5	0
Semaphore initial value wrong	2	0	1
Code editing errors	3	0	0
Array too small	1	0	0
Missing `break`	1	0	0
Incorrect parenthesising	1	0	0
Wrong flag variable	1	0	0

A.2 Defects in Reactor assignment

Name	2005	2007	2008
Requirement	93	112	38
Buffer can grow without limit	N/A	26	30
Busy-waiting or polling	8	6	1
Reactor API changed	31	37	2
All events dispatched to all handlers	4	0	0
`handleEvents` dispatches multiple events	14	N/A	N/A
`handleEvents` dispatches only one event	N/A	1	0
Multithreaded Hangman server	8	1	0
Event handlers in wrong thread	5	5	1
Hangman client does not quit	4	N/A	N/A
Hangman server does not quit	1	3	1
Output format wrong	8	19	2
Input format wrong	1	2	0
Hangman game rules wrong	9	12	1
Programming environment	15	11	1
Console I/O problems	10	N/A	N/A
Fixed TCP port	N/A	10	0
Reserved TCP port	1	N/A	N/A
`notify` or `wait` without holding lock	1	1	1
Checking for available input wrong	1	0	0
`String.concat` expected to modify `String`	1	0	0
Using `Thread.stop`	1	0	0
Incorrect algorithm/implementation	51	56	17
Wait for event even when one is available	2	1	0
Read each handle in turn	7	0	0
Unprotected shared variable or structure	5	3	0
Server reported ready before it is	1	0	0
Server never says it is ready	0	1	0
Buffer can be overwritten when full	1	3	0
Wait condition wrong	2	3	0
Wrong lock	0	1	0
Wrong thread notified	4	2	0
`interrupt` check wrong	0	1	0
Events before main loop lost	1	0	1
Buffer has arbitary size limit	0	0	1
Circular lock dependency	2	0	0
New event signalled too early	1	0	0
Client connects to wrong port	1	N/A	N/A
Array too small	1	0	0
Stack instead of queue	1	0	1
Class in wrong package	1	0	0
Flag variable not reset	0	0	1
Thread started twice	0	1	0
Fails to ignore events from removed handle	21	40	13

A.3 Defects in Tuple space assignment

Name	2005	2007	2008
Requirement	93	49	21
Shared variables in chat system	38	3	2
`synchronized` in chat system	14	5	2
Tuplespace assumed not distributed	N/A	3	0
Tuple space API changed	3	7	2
Polling	4	1	1
Messages combined into one	3	4	0
Wrong amount of old messages	2	0	1
Channel name assumed alphabetic	1	N/A	N/A
Message order ignored	8	9	3
One specific message always ignored	0	10	0
Tuple size limited	3	0	0
Key required in tuple	9	0	0
Only tuples used by chat	1	0	0
Set, not bag, of tuples	2	1	0
Tuples not copied into space	N/A	2	10
`get` does not wait	1	1	0
`get` returns multiple matches	1	3	0
No pattern matching	1	0	0
`get` does not remove tuples from space	2	0	0
Assignment environment	3	0	0
Waiting for GUI-only extra message	3	0	0
Incorrect algorithm/implementation	70	51	36
Pattern matching wrong	6	0	3
List changed while iterating through it	1	0	0
Pattern overwritten	1	0	1
Unique ID is not	0	4	3
Tuple not replaced after read	0	0	3
Never removed read messages	3	8	2
Messages not deleted on listener close	15	18	7
Messages not deleted under heavy load	0	1	0
Message deletion trigger wrong	0	0	1
Off-by-one errors in initialisation	3	1	2
Missing part of initialisation	0	6	3
No locking of buffer	13	0	0
Sequence number or index used without locking	10	1	0
Delete unread messages when buffer full	7	3	1
Write operations may interleave badly	3	3	0
Circular locking dependency	1	2	6
Messages lost when listener added	3	4	0
Memory leaked when listener added	0	0	1
Header processed as part of list	0	0	1
Wrong extent of `synchronized` block	2	0	2
`notify` instead of `notifyAll`	2	0	0

Appendix B

Interview guidelines

- Preparations:
  - As a lot of the questions reference the student's code, I'll print it out in advance (one copy each for me and the interviewee) with line numbers.
  - Kill interrupts (phone, email client...)
  - Check recording equipment.
- General note: As everyone has their own papers, I can mark parts the interviewee references on my copy or mention the file and line number to put the information on the recording.
- Introduction
- Explain the research project
  - Course development aspect.
  - Concurrency misconceptions and errors, debugging the resulting bugs.
- Interview setup:
  - "This is not a test; this is not going to affect your grade" (unless we, as in I and the interviewee find something wrong with the assessment, of course, but it's the interviewee's decision to do something about that).
  - Check that the interviewee knows that I intend to record him, knows the recording is for me but will be transcribed and anonymised and distributed to others in that form and, most importantly, that he is happy with being recorded. Backup plan if he doesn't want to be recorded: unplug the mic and start taking notes.
  - Don't worry about speaking directly into the microphone, the signal is OK at several meters.
  - Feel free to bring up anything you think is relevant.
- Start recording, mark with audio "[name] [student number] [time]"
- "What are you studying [in the general sense]?"
  - Study program, major/minor if applicable

  - Year or stage of studies ("How far are you in your studies?")

- Prior concurrency experience?
- "Can you describe a tuple space for me?"
  - Ask follow-ups to get details of operation behaviour.
    - "What does the [get/put] operation do?"
    - "What input does it take?"
    - "What happens if there's no match?"
    - "How does this work in a distributed environment?"
      - Value passing semantics.
      - Use concrete example of shared tuple space and tuple put on one machine and got on another if necessary.
- For tuple space and chat server:
  - "How did you design your solution?"
    - "Why did you [do it this way/choose this approach]?"
    - "What ideas from the lectures, books or other material did you use?"
    - Focus on problematic parts (as seen from assessment).
- "How did you test your solution?"
  - "What programs or commands did you use?"
  - "Did you find any problems?" [mention any already noted in comments]
    - "What did you do about them?"
  - Did you try to do a correctness proof, verify the program or something like that?
  - "Were you happy with the result?"
- For problems in assessment:
  - "What do you think about what the assistant says? Is this really a problem?"
  - If not clear yet, ask how interviewee thinks the problem was caused, if he agrees it is a problem.
  - "What have you done about it/intend to do about it?"
  - "How did you track it down?" [if applicable]
    - Methods, tools...
- "Any other comments about the assignment?"
- "Any other comments about the course?"
- "Anything else?"
- Immediately afterwards:
  - Save audio recording (Audacity project and FLAC).
  - Jot down summary of interview.
- Transcribe notes as soon as possible.

P1

Jan Lönnberg and Anders Berglund. Students' understandings of concurrent programming. In Raymond Lister and Simon, editors, *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007)*, volume 88 of *Conferences in Research and Practice in Information Technology*, pages 77–86. Australian Computer Society, 2008.

P2

Jan Lönnberg, Anders Berglund and Lauri Malmi. How students develop concurrent programs. In Margaret Hamilton and Tony Clear, editors, *Proceedings of the Eleventh Australasian Computing Education Conference (ACE2009)*, volume 95 of *Conferences in Research and Practice in Information Technology*, pages 129–138. Australian Computer Society, 2009.

P3

Jan Lönnberg, Lauri Malmi and Anders Berglund. Helping students debug concurrent programs. In Arnold Pears and Lauri Malmi, editors, *Proceedings of the Eighth Koli Calling International Conference on Computing Education Research (Koli Calling 2008)*. Uppsala University, 2009. To appear.