Publication II

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

How Students Develop Concurrent Programs

Jan Lönnberg[1] **Anders Berglund**[2,1*] **Lauri Malmi**[1]

[1] Department of Computer Science and Engineering
Helsinki University of Technology,
Espoo, Finland,
Email: {jlonnber,lma}@cs.hut.fi

[2] Department of Information Technology
Uppsala Computing Education Research Group, UpCERG
Uppsala University,
Uppsala, Sweden
Email: anders.berglund@it.uu.se

* Temporary affiliation

Abstract

This paper describes a qualitative, explorative study of how students approach developing and testing concurrent programs. The study is based on interviews with students working on the final programming assignment in a concurrent programming course. We discuss 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 can be designed to support the development, testing and debugging of concurrent programs.

1 Long-term Research Aims

The ultimate goal of our project is to help programmers produce better concurrent programs. Our approach to this is to develop methods and tools, primarily program visualisations, to help programmers understand what a concurrent program does.

Different errors can be the result of completely different ways of thinking. Approaching a problem from the wrong perspective may lead to erroneous conclusions. The nature of the errors can also depend on the perspective or task at hand. Thus, understanding how the programmer is thinking is important in finding ways to prevent errors from being made as well as determining the errors to look for and how to look for them. For example, a programmer who misunderstands the requirements or specification of a system or module or envisions a different purposes for a system will also be testing according to this erroneous understanding of the requirements.

We focus on inexperienced programmers, in particular students, for several reasons. One is that their inexperience means they have more difficulties and therefore need more help. Another is that helping students understand their mistakes not only helps them get their programs to work; it also helps them learn. Thirdly, we can collect and analyse large amounts of data from students, with less effort than from commercial software developers. Finally, it is easier to introduce new ways of working to students than experienced professionals with ingrained habits. We have chosen to focus on studying the understandings of students for three different reasons. Based on the above, our large-scale approach is to first identify the needs of the intended users, students, and then design solutions to address them. The general questions we therefore seek answers to are:

- What kind of defects do programmers inexperienced in concurrent programming introduce in concurrent programs, and why?
- Which of these defects are difficult to locate or understand and why?
- What kind of tools can assist a programmer in finding and understanding these most problematic defects, and how well do they work?

The first results from this project were quantitative information on students' concurrent programs' defects (Lönnberg, 2007). We then proceeded to seek an explanation for the defects we found in the students' understanding of concurrent programming (Lönnberg and Berglund, 2008). This paper describes the continuation of that investigation. Here, we describe students' understandings of the tuple space concept as well as their general understanding of program development and debugging. This study can be seen as exploring the user requirements for future software development tools for students of concurrent programming (the first two questions above). At the end of this paper, we briefly discuss what sort of tools would address the issues by this study (the third question); a theme that we address further in another paper (Lönnberg et al., 2008).

1.1 Aims of This Study

The purpose of the work presented in this paper is to shed light on *how students approach developing and testing concurrent programs*. These insights can serve as a platform for exploring possible sources of errors, especially those that stem from approaches that are ill-suited to developing reliable concurrent programs. The key motivation here is that a better understanding of errors will help us design better software to support program development, understanding and debugging.

In this study, we explore the different ways in which students in a concurrent programming course approach developing and testing a concurrent program. We do this using an empirical, qualitative research approach called *phenomenography* (Marton and Booth, 1997). Phenomenography investigates the qualitatively different ways in which a group of people

experience or think about something. In our study, this provides us with a starting point for exploring many different situations where improvements to teaching or software development tools can be made.

We focus on the design, implementation and testing parts of the software development process. These are the phases in which errors are introduced that can be found by testing performed by developers based on the specification or requirements they are working from, and which require debugging to trace.

Errors in specifying or communicating requirements result in a clearly different form of defect (the program is working as specified), and they can therefore be treated as a separate problem that is not considered here. Similarly, activities performed on the finished code either have no effect on the code as such (e.g. distribution and installation) or can be considered a return to a previous phase (e.g. changing the code to meet new requirements or fixing bugs).

1.2 Students' Understandings of Concurrent Programming

Students may approach the task of developing a concurrent program with different goals in mind than their teachers. Ben-David Kolikant (2005) 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" and that roughly a third of the students were sometimes satisfied with only compiling their program to ensure it 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 (1999) 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 (2004) 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.

2 The Study

Two qualitative empirical methodologies, *phenomenography* and an informal qualitative method inspired by grounded theory, are used in this project to explore how students understand concepts in program development. The data for the study are collected through interviews (Subsection 2.2) and are then analysed in two conceptually different ways.

Our key research approach in this project is phenomenography. This approach aims to reveal the different ways in which something is understood by a cohort (Marton and Booth, 1997). In recent years, the interest in phenomenographic research has increased in the Computer Science Education (CSE) community, since the results that are offered, focusing both on the learners and what they learn about, have been shown to be useful within CSE (Berglund, 2006; Berglund et al., 2008). Our use is mostly consistent with this, as we aim to reveal how certain phenomena are understood by a cohort.

Berglund (2006) describes the process of phenomenographic research in computer science education as consisting of a data collection phase and an analysis phase. In the former, the researcher interviews students about the phenomena under investigation.

For most of the questions we investigate, we have found that the students' expressions can be summarised as different perspectives on certain phenomena. In one case, the standpoints expressed by the students do not refer to a single phenomenon but several related phenomena and are therefore better described as a set of differing opinions, less coherent than those that have lent themselves to phenomenographic analysis. Here, we have chosen a different method to list the opinions.

Questions of validity and our responses to them are discussed throughout this paper as they are raised by our methods and results.

2.1 Setting

The students in this study participated in the Concurrent Programming course[1] at Helsinki University of Technology during the autumn of 2006. Students could choose to do the assignments alone or in pairs.

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.

Lincoln and Guba (1985) emphasise the importance of a natural research setting in getting results that deal with real situations. In this case, the setting was an existing programming course; the only change made for this research was adding interviews. They also argue that determining the transferability of results from one context to another requires knowledge of both contexts; we provide a description of our context to allow the reader to determine which of our results apply to his or her context.

2.2 Interviews

The interviews focused on the third assignment, *Tuple space*, in which the students implement a tuple space (Gelernter, 1985), which consists of a space containing tuples that can be added, read and removed atomically, using Java synchronisation primitives (shared memory, monitors and conditional variables) and use this to construct the message-passing section of a distributed chat server. The students' message-passing code communicates with the rest of the system using method calls; a simple GUI front-end is provided for testing.

The eight students were those who volunteered for the interview out of 16 selected students. Making the interviews a mandatory part of the course for selected students was deemed both an unacceptable demand on the students and counterproductive as the interviews rely on interviewees volunteering information.

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

[1] The contents of the course, including the assignments, are described on the course web page: `http://www.cs.hut.fi/Studies/T-106.5600/2006/english.shtml`

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.

In order to maximise the variation of experiences based on the information available to us about the students, we chose groups with different types of problems with their code, as determined by the teaching assistant who graded the assignments. Ten of 31 groups that failed the assignment and two of 24 that passed the assignment were chosen and invited to an interview. Of these groups, seven of the failing groups (six single students and one pair) agreed to participate. The first author conducted interviews with these eight students, after the results for the third assignment were announced and before the resubmission of failed attempts. The focus of the interviews was on the development process, especially the students' reasoning behind their design. 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 to almost an hour. This allowed students to, in addition to the topics raised by the interviewer, talk about related issues such as their experiences with the other assignments in the course or with programming in a professional context.

2.3 Analysis

The goal of the analysis was to organise the interviewees' utterings into a form that allows the reader to understand the students' different understandings and approaches to developing concurrent programs.

The analysis was done iteratively by the first author in discussion with the second author and, later, also the third author. Each category was intended to represent one understanding or aspect of a phenomenon; the categories were grouped into outcome spaces by the phenomenon they describe. The resulting categories from the last iteration are presented in the following section.

During the analysis it became clear that one of the resulting outcome spaces is not phenomenographic in the sense that it can more obviously be seen as first-order knowledge of what the students have done than second-order knowledge; what the students experience about what they have done. We therefore consider this outcome space to be a grounded theory.

3 Results

In this section we present the results of our analysis. Quotes are used to illustrate the categories. In these, the interviewer is denoted *Int* and the interviewees are assigned, to preserve their anonymity, the names *Evgeniy* and *Elena* (interviewed separately in English), *Filip*, *Fabian*, *Fritjof* and *Frans* (interviewed separately in Finnish) and *Freja* and *Fredrik* (interviewed together in Finnish). The quotes from the interviews in Finnish have been translated into English by the interviewer. Freja and Elena are, as the assigned names indicate, female; the rest are male.

Using quotes from the interviewees allows the reader to see exactly what was said, albeit mostly in translated form. During the interview, follow-up questions were used to clarify the meaning of the interviewees' words where necessary and ensure the interviewer's understanding. The results presented in this paper are a consensus between all three authors. These measures reduce the potential for misinterpretation through e.g. researcher bias.

The phenomena are:

1. **Purposes of the programming task**: the different ways the purpose of the program to be produced in the assignment is understood by students
2. **Sources of failure**: the different entities that may cause failures in the program that the student takes into account
3. **Software development processes**: the overall development process of the students
4. **Approaches to testing**: how the students understand testing their program

The first two describe what the student is trying to achieve by developing a program; students aiming for a passing grade care about correctness in terms of how it affects their grades. Depending on how the student understands the teachers' priorities, this may manifest in different ways.

The latter two outcome spaces are about how the student develops and tests his or her program, like the *Developing and debugging* outcome space from our previous paper (Lönnberg and Berglund, 2008). The categories of *Tuple spaces* in that paper are also connected to different steps in a development process.

Subsection 3.2 is based on grounded theory, while the others are phenomenographic outcome spaces.

3.1 Purposes of the Programming Task

In this subsection, we present what the students perceive as the purpose of their programming task. These purposes are summarised in Table 1.

These understandings are not mutually exclusive; even apparently contradictory understandings can be applied in different contexts by the same person. For example, Fritjof mentions *Assignment (1A)* and *Ideal problem (1B)* as describing how he approached the assignment, but *Possibilities (1D)* as how systems of this type should be written (see below).

The students' aims in a project course in computer systems have been explored by Berglund and Eckerdal (2005). These findings show similarities with the purpose of the programming task discussed in this paper, as they encompass both requirements set by the university and an environment extending the formal requirements, looking toward a professional life.

The different ways in which the student understands the purposes of the programming task are also similar to the *relative correctness* of Ben-David Kolikant (2005) 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 which context the program works as intended.

The categories described here can be applied to any programming exercises in an educational setting for which a grade is given. *Assignment (1A)*, in particular, is limited to this context.

1A Assignment In this category, the programming task is understood as a task required to get a grade; i.e. as one task of many required to get a degree. The development process is focused on the demands made by the university setting (such as grading and deadlines) and the correctness or functioning of the resulting program is a secondary concern. The value of the program is understood in terms of how it affects the grade.

One symptom of this is students making design decisions based on how they affect their grade. For

	Label	The purpose of the programming task	What is in focus?	Framework
1A	Assignment	To meet the requirements of the university setting	The university setting's requirements	University setting
1B	Ideal problem	To produce a program that functions within the university setting's requirements	The program itself	University setting
1C	Working solution	To produce a solution to a problem beyond the university setting	The program itself	An environment beyond the university setting
1D	Possibilities	To solve a problem with potential for future development	Possibilities for future development	An environment beyond the university setting

Table 1: Purposes of the programming task

example, when asked why he chose to design his chat system in such a way that it may lose messages under heavy load, Fritjof explains that he "*didn't put much effort into that since it wasn't a factor in failing the assignment*". Ironically, this choice was exactly the reason he didn't pass the assignment.

Decisions about process can also be based on this understanding. When Fredrik and Freja are asked how they intend to resolve a problem found in their program, they explain that lack of time prevents them from solving the assignment properly, so they need to focus on how to pass the assignment.

Similarly, errors are understood in terms of feedback from assistants, as exemplified by Freja expressing how functional her solution seemed to her in terms of expected negative feedback from the grader. Her chat system implementation does not remove closed listening connections at all from the system and thus continues to collect messages that will never be read. Her comment on this is:

> **Freja:** I, for one, thought there were going to be complaints about those `ChatListener`s not being removed.

1B Ideal Problem In this category, the assignment is understood as constructing a program that works in an idealised school environment where practical limitations such as finite memory space and delays do not apply. The focus here is the program itself, albeit as a part of the university setting.

Fritjof provides two different examples of this. He describes how he resolved a problem with messages being lost when listeners do not read them quickly enough by allowing the buffer to expand without limit. He then comments:

> **Fritjof:** Yes, the new solution uses unlimited memory. It's sort of an ideal situation, but isn't this whole assignment a bit ideal?

This shows how Fritjof has changed from one simplification (that a fixed-size buffer will never overflow) to another (that memory is unlimited, so the buffer does not need to have a size limit), and justifying his latter simplification by commenting that the assignment is not representative of reality, so his solution need not be either.

1C Working Solution In this category, the assignment is understood as constructing a program that works correctly in a realistic scenario. The focus is on creating a program that works under normal real conditions (as opposed to the ideal or academic conditions of the previous category), such as the student's own computer. For example:

> **Frans:** I just used some normal use cases and then thought about what problems one could have now and looked if they show up, and if so, why.

1D Possibilities In this category, the assignment is understood not only as a task to be completed, but as a starting point for future development. This category extends the previous one by going beyond the program itself into the realm of possibilities for future software and development.

The possibilities may be better ways to achieve the perceived goals of the assignment. Fritjof explains that the specification prevented him from writing a program that is resilient to network errors, as it did not provide a way to detect network failure. This illustrates how he thinks a chat system should work.

These possibilities may also be ideas for making future projects easier suggested by the assignment. For example, Evgeniy would like to see "*a debugger or some kind of unit testing system for testing synchronised methods*" and justifies this by noting that "*with current tools you end up pretty much proving it on paper*".

Communicating the Purpose of the Assignment to Students

Different aims can easily lead to different types of errors. For example, a solution to an *Ideal problem (1B)* can, as our example shows, lead to a program that fails in practice or is unacceptable from a teacher's perspective. One can argue that students should be warned that the assignment is graded as a *Working solution (1C)* or taught to approach programming in a corresponding fashion. If the assignment is supposed to let the student practise or evaluate the students' skills at determining requirements, this is more satisfactory than explicitly listing everything a student should take into account (e.g. memory use, performance, network failure). This also illustrates how this outcome space could easily have been different if, for example, the grading criteria had been explained in detail and in advance to the students.

When designing systems to explain errors resulting from an unsuitable understanding of purpose, the underlying context must be made clear: the student must understand, for example, that his or her solution fails because there is not unlimited memory, and that this is something that must be addressed in the solution. Visualising memory usage (e.g. by showing object lifetimes and heap allocation over time) may help students understand this type of problem. Another example is assuming the underlying tuple space is always FIFO; giving the student a test setup where this is not true helps expose this assumption.

	Source	Effect on program design
2A	Systems	Tolerate other systems' failures
2B	Programmer	Minimise chances and/or consequences of programmer error
2C	User	Tolerate user error

Table 2: Sources of failure

3.2 Sources of Failure

In order for a program to be useful, it must interact with other entities. This typically involves interacting with other software, especially libraries and operating systems, and human users. As real-world entities tend to be imperfect, risks stemming from the entities can be found that may cause the program to fail.

In the following, we describe the failure sources taken into consideration by students. They are summarised in Table 2.

The three categories here are essentially the three different classes of potentially imperfect entities the student's program interacts with over its lifespan: the student, who as a *Programmer (2B)* may make errors and thus introduce defects, *Systems (2A)*, hardware and software, that the system builds on or interacts with and the *User (2C)*, who may make mistakes when using the program.

A programmer can take any combination of these sources into account. Fritjof shows awareness of the *Programmer (2B)* and *Systems (2A)*, for example, while Filip mentions the *Programmer (2B)* and *User (2C)* (see below).

It should be noted that both the *User (2C)* and *Systems (2A)* sources were de-emphasised by the teachers to avoid complicating the assignment with error checking.

These *Sources of failure (2)* apply to more or less any program (in extreme cases, such as a program with no user input, one of the risk factors may be trivial enough to be ignored).

2A Systems If the goal of the development process is to make a 'bullet-proof' solution, the program must be written to recover from failures in the systems it interacts with. Fritjof's example of *Possibilities (1D)* illustrates this category as well.

2B Programmer An obvious source of failures is defects in the program introduced by the developer.

One reaction to this is to keep things simple.

> **Fritjof:** If I start playing with optimisation, things can go wrong really quickly. I'd break the code that works. Let's just stick to basics.

He goes on to explain how simple structures (in his case, `while(!found)` {try again}) make it easy to show that code is correct:

> **Fritjof:** It was a sort of emergency solution, so you can't get out of the loop before it's really found.

2C User The third and final source of problems is the user, who may provide invalid input or make mistakes. The programmer may, however, have different ideas of what a user error is than the specification. Filip justifies deleting duplicate and empty messages (to compensate for the system occasionally duplicating messages) by arguing that he "*assumed that you don't want to put empty messages*".

Encouraging Awareness of Sources of Failure

The *Sources of failure (2)* considered by the students are closely related in that they also describe the intended context of the program. While the assignment was designed to allow the student to ignore *Systems (2A)* and the *User (2C)* as sources of failure, taking these into account makes sense in a larger perspective and should therefore arguably be encouraged, not discouraged, by teachers. *User (2C)* errors are easy to create (even unintentionally), but simulating problems in *Systems (2A)* can be difficult and is a possible area for testing tool development.

3.3 Software Development Processes

In the previous subsection, we described different ways students had to structure parts of their solution. Here, in contrast, we present different ways they structure their development process, or, in some cases, do not structure. The process understandings are summarised in Table 3.

The six categories of development process models can be seen as a progression from an unstructured or informal development process to a structured one.

Software development is traditionally divided into separate activities that together form a development *process*. The most important of these activities are *requirements specification*, *design*, *implementation* and *verification*. We use this traditional division to describe the students' development process understandings.

In a study of novice programmers, Booth (1992) 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.

The process models shown here can, in principle, be applied to more or less any programming project.

3A No Design Needed In some cases, the implementation may be obvious enough from the requirements that the programmer feels no design is needed; no discernible parts can be found in constructing the code from the requirements.

> **Fabian:** The tuple space implementation was done quite mechanically. It was already mostly defined how tuples are put in there and in what form.

Lack of time is another reason to cut a process down to the bare essentials.

> **Fabian:** I didn't think about it, I figured I'd spend very little time on getting it done, what with the deadlines and all.

In this category, only the essential parts of the assignment, the requirements and the end result, the program, are taken into account, and the focus is on producing the end result.

3B Trial and Error Some students showed no signs of having a planned process. Instead, they described their development process as *trial and error*. As in the previous category, there are no discernible intermediate steps in the process. However, a simple structure can be seen: code is repeatedly written

	Label	What is the process understood as?	What is in focus?	Framework
3A	No design needed	Writing code directly based on requirements	Writing code	Requirements and code
3B	Trial and error	Writing code to find a solution that meets requirements	What code works?	Requirements and code
3C	Coding to understand	Writing code to understand the requirements	Understanding the requirements	Requirements and code
3D	Inertia from previous work	Writing code based on own previous work	Writing code	Requirements, code, own experiences
3E	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
3F	Adapt known solution	Writing code based on others' previous work	Structuring the solution	Requirements, code, solution archetypes

Table 3: Software development process models

(trial) and found inadequate (error), until the solution is "good enough"

One reason for this is unfamiliarity.

> **Evgeniy:** After a few tries, you realise some things about your previous design and you try to improve it. I wasn't really trying to design it too much ahead, because tuple space was a new thing for me.

Another reason is experiencing problems getting your implementation to work. Frans's approach was "*mostly through trial and error*" after he failed to find a pre-existing solution he could use. He later explains that:

> **Frans:** This trial and error method was a bit bad in the sense that you don't really have a clear picture of what the idea behind it is, how it works, and when you try to fix it, it's kind of hard because you don't know what it's supposed to do, what you have been thinking about.

The distinguishing characteristic of trial and error seen here is lack of planning leading to work that must be redone when it is found inadequate.

3C Coding to Understand While trial and error is often used as a way to reach a solution, misreading or having difficulty understanding the requirements may also lead to trial and error. This category is similar to the last one, except that the purpose of the repeated coding is to understand the requirements. This further separates understanding the requirements (which involves writing code that may solve them) from producing the code that forms the end product.

Some trial and error is inevitable, as one can not understand the development task fully until one has attempted it, making it hard to design ahead.

> **Fredrik:** First I've had to start coding something just to little by little get a sort of grip on what it's about, and not until you start coding the whole thing for the second time does it end up even close.

It's also possible that the student, like Freja, simply "*didn't notice the requirement*".

3D Inertia from Previous Work This category involves solving the assignment the same way as one has done other programming tasks in the past. Compared to the previous categories, a source of approaches, the programmer's experience, is added. This introduces a way to find solutions: reuse your old solutions to old problems.

Fritjof explains that assignments may be "*hard to see as separate entities*" if one has "*previously done the sort of code where you do these big lumps, which is bad*". In his case, he "*just had to put together a monolithic system*".

3E Apply Known Technique Many design techniques such as diagrams exist that can make it easier to design a program. Compared to the previous category, the programmer now has formal design methodology to draw on in solving his or her problem. These are often intended to help split the problem into subproblems by structuring the intended system first.

For example, Fabian "*quickly drew a collaboration diagram containing essentially all the classes worth mentioning*" and then looked at how to build it, in the sense of which message goes where.

3F Adapt Known Solution Instead of unconsciously using one's own previous work as a guide, one can consciously use a known solution as a starting point. Like the previous category, this category adds more knowledge from which a solution can be constructed: other people's solutions to similar problems.

Experiences vary. For example, Frans tried to find something he could adapt from "*some book on concurrent programming [he] borrowed from the library*", but ended up using trial and error, and ended with a problematic solution, of which he said:

> **Frans:** I figured one ought to carefully approach it by thinking of a sort of existing method to use there, since that's the sort of mess where you don't know what it does really.

Fredrik, on the other hand, found:

> **Fredrik:** For many problems, even hard ones, a good and efficient solution has been invented. Usually, even the tough concurrency stuff is abstracted away from the actual business logic that you're implementing.

Why Should Teachers Care about Students' Development Processes?

Many of the process models described here, like the categories of *Development and debugging* in our previous paper, are based on or engender ignorance or

lack of understanding, as the students themselves admit. This is worrying considering that the students are not novices at programming. They are also simplistic compared to those described in the software engineering literature, although a complex process is hardly justified in such a small project.

In cases where students structure their solution in well-known ways (*Apply known technique (3E)* or *Adapt known solution (3F)*), it would seem useful to express information about the student's program using the similar structures. If a teacher can infer the structure behind a student's solution, the teacher can explain in the student's terms where (in the process and the solution) the student has gone wrong. Similarly, a student is likely to understand visualisations that show the student his or her program's execution using notation and a partitioning he or she is familiar with.

3.4 Approaches to Testing

In this subsection, we examine the approaches to testing taken by students in terms of what they understand testing to consist of; in other words, how they structure their testing. These views are summarised in Table 4.

Verification, especially in sequential software, typically relies heavily on *testing*. However, the unpredictability of interaction between concurrently executing processes also introduces many pitfalls in the software development process that may result in software defects that are hard to find through testing. There are, therefore, several approaches to ensuring correctness despite nondeterminism, including *deductive proofs* (usually manually constructed) and *model checking*. These formal methods, however, have limited ability to cope with large and complex programs.

Thus, the categories in this subsection can be seen as steps from an undeveloped testing process to a developed one.

Unplanned (4A) testing can be seen as a base that *Breaking the system (4B)* and *Covering different cases (4C)* extend by targeting testing, while *External testing support needed (4D)*, *Testing inadequate (4E)* and *Proof necessary (4F)* are successively clearer pictures of the limits of tests.

While these categories are all applicable to sequential programming, the last two categories are motivated by concurrency issues.

4A Unplanned Unplanned testing involves running the program and passively observing the output to see if anything goes wrong. In this category, input is provided to get the program to do something, but the testing is not directed toward making defects manifest themselves; it's more a matter of convincing oneself that one's program works.

For example, when asked to clarify what he means by "normal use cases" in his quote about a *Working Solution (1C)*, Frans explains that in order to test both tuple space and chat system:

> **Frans:** I just opened two or three chat windows and then wrote some stuff or used the built-in flood feature.

4B Breaking the System The goal of testing can be understood as getting the program to fail, and the testing process then involves setting up cases in which the system is likely to fail. This category extends the previous by adding a goal to the testing: getting the program to fail.

When asked whether he noticed his chat system's failure to enforce message order, Fabian explains the he "*tried to get it to break using the provided user interface*".

Stress testing is one particular way to attempt to break the system. For example, Freja tested her chat system implementation by running many clients and servers with heavy traffic. However, she goes on to explain:

> **Freja:** When I got that running, it worked nicely, so I thought we might even pass this, but, what do you know, there was another "fail".

Her solution relied on the tuple space being FIFO; no testing using Fredrik's (FIFO) tuple space would have exposed this.

4C Covering Different Cases In this category, a strategy for choosing test cases is added: many different cases or ways in which the program can behave are tested. The underlying assumption is that other, untested, cases are similar enough to be covered by these tests. Stress testing is then only part of the cases tested.

Diversity in testing can involve both choosing different data for the program and studying the program's behaviour in different ways. For example, Fritjof says his testing "*was just sort of trying things out with all sorts of cases*". He "*started from the basics*" and moved on to stress testing. Finally:

> **Fritjof:** I went line by line through the lines of code, stopping at certain points in the code and looked at the innards of the program at that point.

Covering different cases does not preclude focusing on likely problems, as Frans explained in *Working solution (1C)*.

4D External Testing Support Needed In this category, limitations of the student's own testing ability appear; the student realises he or she cannot find all his or her own errors and wants outside help.

One reason is being blind to one's own mistakes, like Fritjof:

> **Fritjof:** I found the problem almost directly based on the teaching assistant's explanation. I guess it was a really clear error, and I just couldn't spot it in my own tests; I was blind to that error. That sort of thing is really hard to test without a fancy testing facility or something.

Fritjof also mentions the importance of quick feedback, like in the Goblin (Hiisilä, 2005) programming course management and assessment system he has used in introductory programming:

> **Fritjof:** What I especially like is that you can submit and see what it looks like, red or green, and it sort of gives an impression whether my solution is close to the right one now.

Regarding the API test package provided by the course staff, he says "*That's just 10 % of the assignment; the big problem is the concurrency management*".

4E Testing Inadequate As in the previous category, an awareness of the limitations of testing is added here. In this case, testing in general is seen as insufficient.

	Label	What is testing understood as?	What is in focus?	Framework
4A	Unplanned	Trying out the program to see if it works	How program reacts to input	Features, test inputs and outputs
4B	Breaking the system	Trying to get defects to manifest as failures	Finding inputs that make the system fail	Features, test inputs and outputs
4C	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
4D	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'
4E	Testing inadequate	Part of ensuring the program is correct	Limitations of testing	Own testing ability and others'
4F	Proof necessary	A complement to a correctness proof	Limitations of testing	Testing and proving correctness

Table 4: Testing approaches

Some defects may be very hard to get to manifest on a normal system. Filip says that in his case all the reasons for failing the assignment were such that they couldn't be found with any decent testing, because "*they were mostly hypothetical*". "*If you're just testing on a home computer, it's really hard to get them to show up*".

In *Covering different cases (4C)*, Fritjof also points out that concurrency-related problems are hard to track with debuggers.

4F Proof Necessary Correctness proofs are considered an important and powerful way to verify a program. This category introduces a solution to the limitations of testing: supplementing it with proofs. Evgeniy's comment about *Possibilities (1D)* is an example of this.

Encouraging Better Approaches to Testing

Many of the students' testing approaches are, like their development processes, superficial and, as the quotes illustrate, easily allow problems, especially related to concurrent programming, to slip through. However, the more advanced testing approaches show an awareness of concurrency-specific factors that affect testing, in particular nondeterminism. As suggested by some of the students, providing testing and debugging facilities that support finding concurrency problems by providing ways to generate and study different types of nondeterministic behaviour would be useful. For example, generating and visualising interleavings both with and without failures for the same input could help students understand why their code is unreliable. The visualisation should emphasise synchronisation, for example by displaying interactions between threads through locking and shared data as e.g. a sequence diagram. Another possible approach is to use static analysis or model checking to search for incorrect synchronisation solutions and point out the relevant parts of the code to the student.

4 Conclusions

In this paper, we present the different ways in which students understand the purpose of concurrent programming tasks, the sources of failure they take into account in doing so, the process models they base their development work on and their approaches to testing. They are found to have a wide range of different understandings ranging from simplistic to advanced. In order to cope with these, teachers should be aware of them and adapt their assignments, grading and tools they provide to the students to take these understandings into account. One of example of such tools would be concurrency testing tools to help students find incorrect assumptions about the execution environment and interleavings that cause failures and visualisations to help them understand their errors and learn by correcting them.

Most of the outcome spaces appear to be applicable to other programming contexts, particularly programming exercises in an educational context. On the one hand, this bodes well for applying the results to other programming courses. On the other, this raises the issue of whether the results say anything about concurrent programming specifically.

4.1 Understanding Goals

The *Purposes of the programming task (1)* and *Sources of failure (2)* uncovered here suggest that many of the errors made by students are misunderstandings of the environment in which their program is expected to run and what it is supposed to do rather than actual misunderstandings of concurrent programming itself. This is in line with our quantitative analysis of students' defects (Lönnberg, 2007). This suggests that teachers should make goals more explicit and provide students with ways to explore problems related to these goals.

Providing students with tools to study memory allocation would help them understand how their programs use (or misuse) memory. This could be as simple as showing them how to use a basic 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.

Similarly, it is not realistic to expect students to explictly ensure message order if they never see messages get out of order even though they have ignored the issue completely.

The students' *Software development processes (3)* are often disorganised, partially because they do not understand what they're supposed to do. This could also be mitigated by more explicit goals and subgoals or by teaching ways to structure a software development process and a program.

4.2 Generating Test Cases

The students' *Testing approaches (4)* are quite weak. One possible reason is that the students do not take

concurrency into account properly in their testing. Another is that, as discussed in the previous subsection, the students do not understand the environment in which their program is to function. This situation could be improved by teaching testing and complementary forms of verification of concurrent programming.

Another approach to helping students test their programs is providing testing tools to generate scenarios that are hard to discover using normal testing procedures. In particular, students need to be able to study how their programs behave when concurrent execution threads interleave in different ways. One possible approach would be to allow students to manually control how their programs' instructions are interleaved, allowing the student to examine known problematic cases in detail. Another is to automatically generate the problematic cases using e.g. a model checker, which helps when students are not aware of a possible problem.

4.3 Understanding Program Behaviour

Debuggers traditionally focus on behaviour on the level of individual statements, as in the implementation category. However, the *Solving technical problems* category (Lönnberg and Berglund, 2008) as well as the *Apply known technique (3E)* and *Adapt known solution (3F)* categories suggest an alternative perspective on debugging: that it would be useful to provide supporting tools, such as execution visualisations, that show program behaviour in ways that support the user's understanding. This could be done by allowing the user to group together parts of the code or execution to correspond to his or her understanding, similarly to the ability to change between program- and algorithm-level behaviour suggested by Price et al. (1993). The tool would then visualise the behaviour of the program in a fashion closer to the programmer's view. For example, if the programmer understands his or her program as a set of communicating entities, the tool should be able to display 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. Similarly, familiar notation is preferable; if a student has designed a solution using collaboration diagrams, he or she should have less trouble understanding a description of a failure expressed as a collaboration diagram than using an unfamiliar notation.

When communicating a concurrency-related failure to a student, describing the sequence of events leading to the failure can be difficult. Understanding the exact order of events and how this affects the interactions between threads is often crucial to understanding the underlying defect and error and eliminating it. For this reason, the ability to store a particular interleaving for further study is important; this can also be helpful in debugging in general.

4.4 Understanding Errors

The results of this study can also help teachers determine students' errors based on code defects and explanations, by showing the different ways students understand concurrent programming. This is useful when assessing students' work in many ways. One is that it allows grades to reflect the student's understanding and skill better; instead of deducting points based on failures or defects (which may have little to do with the student's skills), they can be deducted for errors that are direct consequences of lack of understanding or skill.

Similarly, the results of this study will help us determine the errors underlying students' defects, allowing more meaningful analysis of these defects. This also helps in explaining the student's defects and errors to him or her.

5 Summary

Many understandings of concurrent programming can be found among students that cause them to write programs that do not work properly:

- Producing a working program is not seen as the purpose of a programming assignment.
- Design is unnecessary or impractical, so developing a program relies on trial and error.
- Testing is cursory and does not take nondeterminism into account.

Our response to these issues is threefold. First, we suggest that students need more explicit and detailed guidance on how to apply different verification techniques in practice and that assignments should be designed to encourage careful development practices. Second, we argue that showing students the consequences of the decisions they make due to their understandings will help them form more useful understandings. Third, some aspects of debugging concurrent programs are difficult, especially testing and debugging. We intend to address this by developing software to help find concurrency-related defects and visualise the failure to facilitate debugging and allow the student to understand his errors and misconceptions.

References

Ben-Ari, M. and Ben-David Kolikant, Y. (1999), Thinking parallel: The process of learning concurrency, *in* 'Fourth SIGCSE Conference on Innovation and Technology in Computer Science Education', Cracow, Poland, pp. 13–16.

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

Ben-David Kolikant, Y. (2005), Students' alternative standards for correctness, *in* 'The Proceedings of the First International Computing Education Research Workshop', pp. 37–46.

Berglund, A. (2006), 'Phenomenography as a way to research learning in computing', *Bulletin of Applied Computing and Information Technology* **4**(1).

Berglund, A., Box, I., Eckerdal, A., Lister, R. and Pears, A. (2008), Learning educational research methods through collaborative research: the PhICER initiative, *in* Simon and M. Hamilton, eds, 'Proc. Tenth Australasian Computing Education Conference (ACE 2008)', Vol. 78 of *Conferences in Research and Practice in Information Technology*, Australian Computer Society, Wollongong, NSW, Australia, pp. 35–42.

Berglund, A. and Eckerdal, A. (2005), 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', pp. 65–72.

Booth, S. (1992), Learning to program: A phenomenographic perspective, Acta Universitatis Gothoburgensis, doctoral dissertation, University of Gothenburg, Sweden.

Gelernter, D. (1985), 'Generative communication in Linda', *ACM Transactions on Programming Languages and Systems* **7**(1), 80–112.

Hiisilä, A. (2005), Kurssinhallintajärjestelmä ohjelmoinnin perusopetuksen avuksi (Course management system for basic courses in programming), Master's thesis, Helsinki University of Technology. In Finnish, abstract in English.

Lincoln, Y. S. and Guba, E. G. (1985), *Naturalistic Inquiry*, Sage Publications.

Lönnberg, J. (2007), Student errors in concurrent programming assignments, *in* A. Berglund and M. Wiggberg, eds, 'Proceedings of the 6th Baltic Sea Conference on Computing Education Research, Koli Calling 2006', Uppsala University, Uppsala, Sweden, pp. 145–146.

Lönnberg, J. and Berglund, A. (2008), Students' understandings of concurrent programming, *in* R. Lister and Simon, eds, 'Proceedings of the Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007)', Vol. 88 of *Conferences in Research and Practice in Information Technology*, Australian Computer Society, Koli, Finland, pp. 77–86.

Lönnberg, J., Malmi, L. and Berglund, A. (2008), 'Helping students debug concurrent programs', Accepted to Koli Calling 2008.

Marton, F. and Booth, S. (1997), *Learning and Awareness*, Lawrence Erlbaum Associates.

Price, B. A., Baecker, R. M. and Small, I. S. (1993), 'A principled taxonomy of software visualization', *Journal of Visual Languages and Computing* **4**(3), 211–266.