Publication III

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

Defects in Concurrent Programming Assignments

Jan Lönnberg
Aalto University
School of Science and Technology
P.O. Box 15400
FI-00076 Aalto, Finland
jlonnber@cs.hut.fi

ABSTRACT

This article describes a study of the defects in the programs students have written as solutions for the programming assignments in a concurrent programming course. I describe the underlying causes of these defects 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 and has been 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.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education; D.1.3 [**Programming techniques**]: Concurrent Programming

Keywords

Concurrent Programming, Defect Cause Analysis

1. INTRODUCTION

An important first step in improving a process is understanding where it fails to produce the desired result and why. Quantitative information is particularly helpful in this endeavour, as it allows accurate and easy prioritisation of possible improvements.

Students' solutions to programming assignments provide information that can be used to improve several interlinked processes. The purpose of an assignment is typically twofold: to allow students to learn to apply in practice what they have been taught and to evaluate how well they have learned. The assignment solutions submitted by students (*submissions*) can also be used to evaluate, indirectly, the teaching the students have received. The submissions can also be used to improve assignments and assignment grading processes to make them determine the students' skills more effectively and accurately.

Information on the defects introduced by programmers can be used as a starting point for the development of debugging methodology and tools. Studying programming assignments in education allows one to get statistically meaningful data on errors made in a specific task. By contrast, in professional development contexts, large numbers of programmers seldom implement the same specification.

In this article, I will describe the defects found in students' programming assignments in a course on concurrent programming and their causes, to the extent they can be deduced. I will then present some conclusions that can be drawn from these data that are relevant to teachers, assignment developers and graders.

Koli Calling '09 October 29 - November 1, 2009, Koli, Finland

1.1 Related Work

The work described here can be considered to belong to two different areas of research: research on defects in programs and research on students' problems with programming. The former research field aims to improve quality of software by understanding why programmers err, while the latter aims at improving the quality of teaching.

1.1.1 Defects in Software

When studying program *defects* (discrepancies between the actual program and the correct one, commonly known as *bugs*), and the underlying programming *errors* (mistakes), several approaches can be taken that support different approaches to the overarching goal of improving software quality.

One approach, used by e.g. Eisenstadt [8] with the goal of understanding and mitigating difficulties in debugging, is to concentrate on collecting anecdotal data on bugs that were hard to fix and the debugging process involved. The conclusions include types of bugs (such as writing outside allocated memory) that are hard to track down and the methods used by the programmers who tracked them down (e.g. adding `print` statements and hand simulation of execution). Natually, this approach only provides data on bugs that result in a story the programmer finds interesting enough to remember and share.

Another approach, such as that used by Ko and Myers [16] to form an understanding of errors in order to improve error prevention, detection and correction, is to set up an experiment that is videotaped and analysed in detail. This approach can be used to get very detailed information on error causes, especially if the programmers think aloud, allowing their reasoning to be examined in detail. However, this requires a lot of time for analysis (40 h for 15 h of observation), making it prohibitively time-consuming unless one only observes a few students doing a small project. Furthermore, observation of this type is often hard to do in the natural working environment of the students, which may affect their behaviour.

Defects can be classified in a variety of ways, depending on the relevant aspects. For example, Eisenstadt [8] and Grandell et al. [11] form categories based on the observed

defects; others (such as Spohrer and Soloway [26]) 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 the defect manifests itself), or on the difference on the syntactic level 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. Errors can be classified, for example, by the type of cognitive breakdown involved in the error (lack of knowledge, mistake) (e.g. Ko and Myers [16]) or the part of the program design that is incorrect (e.g. Spohrer and Soloway's goal/plan analysis [26]).

In a software development context, many different types of information related to bugs are useful, resulting in a multifaceted classification scheme such as the IEEE standard classification for *software anomalies* (deviations from expectations observed in program operation or documentation, including bugs) [15]. 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, such as requirements, data structures or data processing.

Eisenstadt [8] also classifies by two other aspects that are interesting from a debugging point of view: why the bug was hard to track down and how it was tracked down.

1.1.2 Students' Programming Errors and Misconceptions

Several studies have been made of students' errors in programming assignments (e.g. [11, 26]) and misconceptions about algorithms (e.g. [25]). The goal of these studies is usually to improve programming education by developing an understanding of students' misconceptions and errors.

The programming assignment studies mentioned above all use the students' code as data; either the final versions submitted by the students [11] or every syntactically correct version compiled by the students [26]. This code was then (mostly manually) analysed for defects.

1.2 Applications

As noted in the introduction, information on the types of defects in students' programs can be applied both in developing teaching and grading and in the development of debugging tools and methodology.

1.2.1 Teaching and Assignments

The results of an assignment can be used to determine whether students are effectively learning what they should. In particular, if a large number of students has problems understanding or applying some relevant knowledge, the teaching of this knowledge should be improved.

If students, on the other hand, produce many defects unrelated to the subject matter they are being taught, the assignment may be testing the wrong knowledge and skills. If the defects can be traced to misconceptions about the assignment or the artificial environment in which it is done (if it exists), 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.

1.2.2 Code Reviews and Manual Assessment

An experienced code reviewer can quickly spot common defects in the programs he reviews, as he knows what to look for. This applies even more strongly to a grader who reads many similar programs. Information on common defects can therefore be very useful to new graders in a course as a substitute for actual experience (both general and assignment-specific). Information on the errors underlying a defect can be used to guess the error made even in the absence of explanatory reports or comments.

1.2.3 Verification and Automatic Assessment

One of the simplest and most commonly used ways to detect defects in a program is to test it and hope that the defects cause *failures* (incorrect program behaviour). However, testing is practically never exhaustive for non-trivial programs [6] and often gives little indication of the actual location of the defect, leaving the actual *debugging* (finding and correcting the defect) to be done essentially manually, with the aid of tools that help examine program execution and detect common errors [23]. Exhaustive model checking is often applied to concurrent programs, but this requires that the entire system be modelled within the verifier and large amounts of memory and processing time.

Programming assignments are assessed automatically in systems such as BOSS [22] or CourseMarker (formerly CourseMaster) [13] by executing tests 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 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. For this reason, manual assessment (using tests and/or model checking to check functionality) or requiring students to apply model checking to their own designs before implementation (e.g. [5]) seems to be favoured for assessing "real-life" concurrent programs. However, automatic assessment has been used for small and clearly delimited concurrent programming assignments such as solving the reader-writer problem or the producer-consumer problem in the SYPROS [12] intelligent tutoring system. SYPROS goes beyond mere assessment to providing detailed feedback tailored to each student while he tries to solve the assignment.

One of the problems with automatic assessment is that it is hard to design tests that detect all common errors and distinguish between different types of error without empirical data on real students' errors. This research into error types and frequencies in concurrent programming assignments is intended to mitigate this problem.

1.2.4 Testing and Debugging Tools and Methods

Current debuggers 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. 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 (see e.g. [14]). Most debuggers can

only show the current state of the program, even though the cause of a program malfunction usually lies in the past. Concurrency also makes debugging harder, as concurrent processes often interact in unexpected ways. These problems combine to make it hard, even with a debugger, to find bugs. Only a few debuggers (e.g. RetroVue [7]) are specifically designed to aid in debugging concurrent programs, and they do not seem to be widespread.

Having quantitative data on programming errors would provide a background against which debugging methods and tools could be developed that address common real-world problems [21]. One foreseeable problem with using data from university programming assignments is that the data do not reflect the skills of professional programmers. This can be mitigated by using advanced university courses.

2. SETTING

This study is centred around the Concurrent Programming course at Helsinki University of Technology[1] 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 and concurrent and distributed systems. Most students major in Computer Science or a related subject such as Mobile Computing and have completed a Bachelor's degree or a roughly equivalent part of a Master's degree.

This paper describes a study of the defects in the programs students wrote in the three mandatory programming assignments[2] of the Concurrent Programming course at Helsinki University of Technology during the autumns of 2005, 2007 and 2008. Due to differences in grading, the autumn 2006 instance of the course has been left out. All of the assignments were to be done in Java. Students could choose to do the assignments alone or in pairs; in both cases, the grading was the same. In 2005, students were allowed to retry the assignment several times. This was reduced to one resubmission in 2006. Resubmission was eliminated completely in 2008; the grading was made less severe and a test package provided to students to compensate.

Students were required to submit both the actual program source code and a brief report outlining how their solution works with an emphasis on concurrency.

As the students were required to submit their solutions through a WWW form that compiled their code, all the submissions were valid Java programs. The last submission by a student or pair of students before the deadline was assessed. Only submissions done before the initial deadline have been examined in this research; late submissions and resubmissions after receiving a failing grade have been left out.

2.1 Trains

In the first assignment (*Trains*), the students are given a simulated train track with two trains and two stations. 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.

2.2 Reactor

The second assignment (*Reactor*) is about the Reactor design pattern [24]. The students' task is to, using the synchronisation primitives built into the Java language, implement a dispatcher and demultiplexer that can read several handles that have blocking read operations at the same time and sequentially dispatch the events read from these handles to event handlers. The students then implement a simple networked Hangman game that uses this Reactor pattern implementation.

2.3 Tuple Space

In the third assignment (*Tuple space*), the student implements a simple tuple space [9] 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 part of a distributed chat server.

3. METHODOLOGY

The process applied here consists of three separate phases: data collection, defect detection and defect classification. They are described in this section.

3.1 Data Collection

The obvious source of information on defects in students' programs is the programs themselves. Furthermore, since students' programming assignments are graded by checking them for defects, the grading process already incorporates much of the necessary defect detection work.

Initial experiments with Java PathFinder [28] in which the model checker failed to complete verification even of simplified versions of the programming assignments described here, encouraged the use of testing to support our grading. Hence, the choice was made to assess the programs manually, essentially by reading the code and the students' explanations of it and checking whether it is correct. Testing was used to find situations that the programs did not behave correctly in.

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. This classification is explained further in Subsection 3.2.

3.2 Defect Classification

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 classified in this way by considering the execution of a call to a forbidden feature as a failure or by considering the operation to behave incorrectly. 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 the limitations of a notional environment are artificially introduced in the real environment, as in our Concurrent Programming course.

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

[1] Since 1 January 2010, this is the Aalto University School of Science and Technology.

[2] For details, see the course web sites at: `http://www.cs.hut.fi/~jlonnber/T-106.5600.html`

3.2.1 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 [26]. However, instead of constructing a full goal/plan tree for each program (which was found to be very time-consuming due to the size of the programs involved and not very useful), only the incorrect parts of the program are considered. Defects are classified by the incorrect or missing subgoal or subplan in the most specific correct goal or plan (assuming top-down development, this means the students' errors are assumed to be made as late as is plausible in the development process). Most defects can be explained this way as errors in a specific plan or goal. Similarly, goals are considered equivalent if a plan that achieves them both is known. Plans are differentiated by their subgoals. While this greatly decreases the amount of different errors, this occasionally results in two otherwise correct plans interfering with each other; these errors are handled separately, as are cases where the students' plans cannot be determined. With some minor refinements and additional defect classes, this classification was used as a basis for assessment in 2007 and 2008.

Previously, I performed the analysis of defects [17] 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 [8]. 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.

Because of this, a phenomenographic analysis was done [19, 20] to provide an understanding of how students understand concurrent programming in order to analyse their defects meaningfully. The resulting phenomenographic outcome spaces 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 [20, Table 3]. 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 [20, Table 4]. Understanding the requirements of the assignment can be seen as a source of difficulties that is great enough to structure one's work around [20, Table 3]. Alternative understandings of the goal of an assignment, which lead to understanding the requirements differently, exist [20, Tables 1 and 2].

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 aspects 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 instead of 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 make a clear distinction between creating an algorithm and an implementation, these errors 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 common, surprising or illustrative of students' understandings of concurrent programming.

3.2.2 Classifying Defects by Failure

An alternative classification is by the type of failure; this is relevant for testing and debugging. Some students showed an awareness of this distinction [20, Table 4].

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.

4. RESULTS

In this section, I describe the defects found in the students' programs in terms of the defect classifications described in Subsection 3.2. Detailed lists of defects are also available [18].

4.1 Trains

An interesting aspect of the Trains assignment (described in Subsection 2.1) 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. 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 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.

4.1.1 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. The trains could also 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 simulator 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 is 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.

4.1.2 Programming Environment-related Errors

In three cases in 2005, students had clear misunderstandings of the Java language or API, such as accidentally generating negative random numbers or leaving out the `break` statement at the end of a `case` of a `switch` and insisting that the fall-through is a compiler bug. Before 2004, introductory programming was taught at Helsinki University of Technology using Scheme instead of Java, so some students may have been unfamiliar with Java.

4.1.3 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.

4.1.4 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; there is no indication that the students considered the possibility that the order could be different. 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

	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

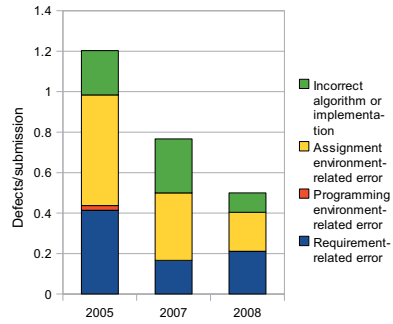

Figure 1: Defects found in Trains assignment

initialising a semaphore to the wrong value; many of these appear to be implementation slips since a correct solution is described and similar situations are handled correctly in the same program. A few unnecessarily complex solutions introduced the possibility of deadlock by ignoring the possibility of a sequence of semaphore operations being interleaved with operations made 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.

4.2 Reactor

The Reactor design pattern in the second assignment (see Subsection 2.2) 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 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.

4.2.1 *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 did 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 [24] 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.

The sharp decrease in deterministic errors in 2008 is almost entirely due to failures to comply with the specified APIs and I/O formats (about 80 % of the deterministic errors) being essentially eliminated by the test package.

4.2.2 *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.

4.2.3 *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; again, there is no indication that these students considered this sequence of events. 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. In some cases, status variables were set at the wrong time or not at all. Two circular locking dependencies were found, which can be seen as two subsolutions conflicting. In some solutions, the defect involved notifying the wrong thread or at the wrong time; again, with no indication that such a possibility had been taken into account. In some case, messages were overwritten or lost, either due to possible interleavings not being considered or because the student did not consider it relevant to handle certain situations, such as messages appearing faster than they can be dispatched or before the main loop is entered. Only a few cases of using collections or variables without the necessary synchronisation were, however, found, mostly involving a flag variable or the list of active handlers not being protected by a `synchronized` block with no explanation given.

A few obvious implementation slips were found, such as having the Hangman server and client connected to different ports, starting the same thread twice, declaring

	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

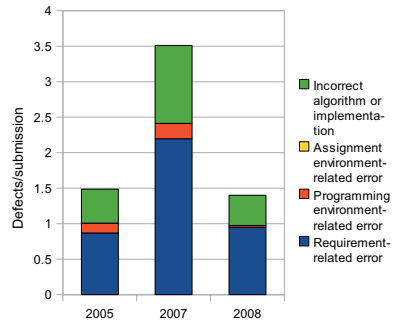

Figure 2: Defects found in Reactor assignment

an array that was one element too small and using a stack instead of a queue.

4.3 Tuple Space

In the tuple space assignment (described in Subsection 2.3), 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 3.

4.3.1 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 log for delivery to new listeners into one message that looked the same to the user of the provided GUI, arguing that they could ignore 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. One error of note of this type (which the test package did not detect) is storing references to tuples in the tuple space rather than copying their contents, which only 2 students in 2007 did, but 10 in 2008. This suggests that students rely on the test package to detect errors in conforming to requirements such as these.

Again, most of the decrease in deterministic errors in 2008 can be attributed to the test package helping students understand they have misinterpreted the specification (more than two thirds of the deterministic errors in 2005 and 2007).

4.3.2 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.

4.3.3 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, for various reasons including implementation slips, understanding returning an object to mean returning its contents and using library classes incorrectly.

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 test package may account for the decrease in 2008. Circular locking dependencies, on the other hand, became much more common in 2008, typically in the form of the locking for different operations, such as writing messages and closing listeners, interfering with each other.

5. DISCUSSION

	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

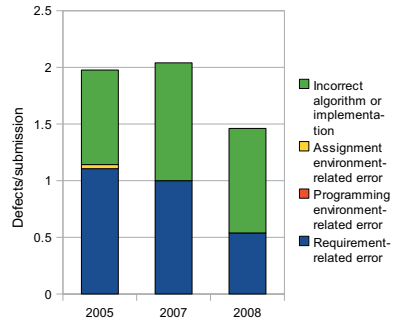

Figure 3: Defects found in Tuple space assignment

This study suggests several areas in which students have problematic understandings that lead to incorrect concurrent programs. These problematic understandings are related both to assignment goals and to the concurrent programming concepts or development practices that are being taught.

The quantitative results appear to show dramatic decreases in certain types of defect in the students' programs as the intended result of certain changes to the assignments. In particular, providing tests helps students notice their problems with understanding assignment requirements. It is possible that other changes to the course (for example, changes to the resubmission policy) or the participating students (for example, the course no longer being mandatory except for international master's students) may also have affected the results.

5.1 Understanding Program Execution

The large amount of defects found in students' programs that cause failures nondeterministically is not surprising, since these defects are both hard to find and correct. Testing software that helps make these defects manifest will help students find such 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. One plausible reason for this is that the students were not capable of debugging their programs despite knowing that they contain bugs. Reasons cited by students include not understanding the tests and being unsure whether the tests timed out due to deadlocks or their code being too slow. The latter can be mitigated by providing debugging tools that can clearly show pending and previous operations on semaphores, monitors and tuple spaces, allowing students to determine what the exact failure is.

Many students introduce defects in their programs that appear to be caused by misunderstanding or reasoning incorrectly about concurrent program execution. In particular, many difficult concurrency bugs the students introduce appear to stem from two different parts of their programs interacting badly. Students should either be encouraged to consider their program as a whole or design it in such a way that interaction between parts is minimised. Another common source of bugs is that some possible orderings of events have not been taken into account. It may be helpful to increase the emphasis on designing programs to avoid unexpected interactions between processes. In both cases, the bugs can also be found, naturally, during verification.

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 the one by Ben-Ari [4] used in the course do not match the concurrency model of e.g. Java [10] 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, students should be shown how their programs really behave so that they can realise that their understanding of concurrency is incomplete and correct it.

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.

Giving students 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 Verification

Students have a wide range of approaches to testing. Some students used completely unplanned, cursory, testing. Some tried to 'break' the system, 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. [20, Table 4]

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 [5], 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. [27]). This is the approach used by the automated testing system of our concurrent programming course.

5.3 Communicating Goals

Students may 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 [20, Table 1]. 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 [20, Table 2].

These purposes of the programming task and sources of failure of the students 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.

In a course that, like our Concurrent Programming course, has as its goal to teach students software implementation techniques with an emphasis on reliability and correctness, it is desirable to have programming assignments with clear and specific goals. One reason is to guide the students into applying the techniques that they are expected in the course to learn to apply. Another reason is that it is hard to say how correct a program is if it is not clear what it is supposed to do. Hence, requirements should be self-contained in the sense that they should be unambiguous and not require specific knowledge of a (hypothetical) usage context or users. The teachers and students can then focus on issues more relevant to the learning goals of the course, such as correctness and efficiency. Finally, if the requirements are specific enough to be expressed as test cases or some other form that can be checked automatically, it is much easier to use automation to assist in assessment and in helping students determine whether they are solving the right problem and whether they are solving it correctly. All this suggests that teachers should, when designing programming assignments for implementation-oriented courses, make assignment goals more explicit and concrete. Naturally, in courses that are intended to teach students to determine user requirements or design systems to meet user requirements, this approach is not applicable; there is a clear need for students to be able to cope with vague or unknown requirements.

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. The student should be able to see his program clearly fail to work correctly rather than be told afterwards that he 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. Naturally, giving students pre-written tests can easily eliminate their motivation for designing their own test cases. Introducing a test package is similar to introducing automatic assessment and allowing students to resubmit each assignment many times. Even when unlimited access to automatic assessment has been given, it seem that only a small minority of students make use of repeated reassessment rather than trying to correct their mistakes independently [2].

6. CONCLUSIONS

The analysis of defects found in students' concurrent programs described in this paper shows that students often have difficulties understanding requirements and taking them into account and in noticing defects that lead to nondeterministic failures. It seems that both issues can be addressed by providing students with test packages that show them how their programs fails to meet requirements. Nondeterministic execution is also difficult for students and debugging tools based on capturing and visualising execution histories can help address this.

7. ACKNOWLEDGEMENTS

I'd like to thank the teaching assistants who did much of the hard work of finding the students' bugs: Teemu Kiviniemi, Kari Kähkönen, Sampo Niskanen, Pranav Sharma, Yang Lu, Ari Sundholm and Pasi Lahti. I'd also like to thank the people who've given me feedback on this work, in particular Lauri Malmi and the reviewers.

References

[1] K. Ala-Mutka. A survey of automated assessment approaches for programming assignments. *Computer Science Education*, 15(2):83–102, 2005.

[2] K. Ala-Mutka and H.-M. Järvinen. Assessment process for programming assignments. *Advanced Learning Technologies, 2004. Proceedings. IEEE International Conference on*, pages 181–185, 30 Aug.-1 Sept. 2004. doi: 10.1109/ICALT.2004.1357399.

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

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

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

[6] I. Burnstein. *Practical Software Testing*. Springer, 2003.

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

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

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

[10] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Sun Microsystems, third edition, 2005.

[11] L. Grandell, M. Peltomäki, and T. 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.

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

[13] C. Higgins, P. Symeonidis, and A. 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. doi: http://doi.acm.org/10.1145/544414.544431.

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

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

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

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

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

[19] J. Lönnberg and A. Berglund. Students' understandings of concurrent programming. In R. 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.

[20] J. Lönnberg, A. Berglund, and L. Malmi. How students develop concurrent programs. In M. Hamilton and T. 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.

[21] J. Lönnberg, L. Malmi, and A. Berglund. Helping students debug concurrent programs. In A. Pears and L. Malmi, editors, *Proceedings of the Eighth Koli Calling International Conference on Computing Education Research (Koli Calling 2008)*, pages 76–79. Uppsala University, 2009.

[22] M. Luck and M. Joy. A secure on-line submission system. *Software - Practice and Experience*, 29(8):721–740, 1999.

[23] R. C. Metzger. *Debugging by Thinking*. Elsevier, 2004.

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

[25] O. Seppälä, L. Malmi, and A. Korhonen. Observations on student errors in algorithm simulation exercises. In *Proceedings of the 5th Annual Finnish / Baltic Sea Conference on Computer Science Education*, pages 81–86. University of Joensuu, November 2005.

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

[27] S. 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.

[28] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232, Apr. 2003.