Student Errors in Concurrent Programming Assignments

Jan Lönnberg
Helsinki University of Technology
P.O. Box 5400
Finland
jlonnber@cs.hut.fi

ABSTRACT

This poster abstract describes the ongoing work at Helsinki University of Technology on observing defects in concurrent programming assignment submissions and examining the underlying causes of these errors and the methods used to find them. The work is part of a larger endeavour to find problematic aspects of debugging concurrent programs and develop approaches to aid programmers in this task.

Keywords

Concurrent Programming, Computer Science Education, Bugs, Errors, Defect Cause Analysis

1. INTRODUCTION

Students' solutions to programming assignments provide material that can be used to improve several interlinked processes. The student-submitted assignment solutions (or *submissions*) can be used to evaluate and improve the students' learning, the teaching and the assignments. Information on defects in students' programs can also be used as a starting point for the development of debugging methodology and tools. Concurrency further complicates the programming process by introducing nondeterminism and its effect on debugging has seen little research.

For the reasons outlined above, I am examining submissions from the three programming assignments on the concurrent programming course at Helsinki University of Technology. The first involves writing control code for simulated trains that communicate using semaphores. In the second and third, the student implements and applies the Reactor pattern [5] and tuple spaces (respectively). The first data was collected during the Autumn 2005 course[1], and more detailed data will be collected during the Autumn 2006 course[2]. In 2005, 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-related behaviour. Defects were found in the programs using a combination of testing and manual analysis and students' explanations of how their code works were used to deduce the underlying mistakes.

The work described here can be considered to belong to two different areas of research: research on defects in programs (e.g. [4]) and research on student errors in computer science assignments (e.g. [3, 6]). The former work aims to improve the quality of software by understanding why programmers err ("What errors do programmers make? Why do they make them? How can we get rid of them?"), while the latter aims at improving the quality of teaching ("What are the deficiencies in the students' knowledge and skill and in the teaching? How can we detect and eliminate them?").

2. APPLICATIONS

As noted above, information on the types of defects in students' programs can be applied to developing teaching, the assignments and the assessment thereof, and to the development of debugging tools and methodology.

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 and/or applying some relevant knowledge, the teaching of this knowledge needs to 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. Therefore, it is important to recognise or eliminate these defects.

An experienced grader can quickly spot common defects in the assignments he grades, as he knows what to look for. Information on common defects can therefore be very useful to new graders on 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.

Automatic assessment of programming assignments is typically done by executing test runs on the code to be assessed and assigning a grade based on the number of tests that passed [1]. One of the problems with automated assessment is that it is hard to design tests that detect all common errors and distinguish between different types of error without empirical data from real students.

2.2 Testing and Debugging

Concurrency makes debugging harder, as concurrent processes often interact in unexpected ways that can be hard to trace (e.g. race conditions). Only a few debuggers (e.g. RetroVue [2]) are specifically designed to aid in debugging concurrent programs, and they do not seem to be

[1]`http://www.cs.hut.fi/Studies/T-106.420/main.html`
[2]`http://www.cs.hut.fi/Studies/T-106.5600/english.shtml`

widespread. Having quantitative data on concurrent programming errors provides a background against which debugging methods and tools can be developed that address common real-world problems related to concurrency. This information is hard to get from commercial development.

3. METHODOLOGY

In this research, the actual debugging and some of the defect detection is done using a set of automated tests and traditional debugging tools (especially print commands) as a complement to reading the code and trying to understand how it works and looking for common mistakes. This is part of the assessment process for the submissions, as knowledge of the defects in the program is required to give a grade that accurately reflects the proficiency of the student in concurrent programming and to provide constructive feedback on the defects (if any) in the submitted program.

The assignments have been designed in such a way that the solutions will be similar in structure. This means that many of the defects in the submissions can be considered to be the same defect in the algorithm that the program implements, providing a natural way of grouping many of the defects. The defects are further grouped by the part or aspect of the program affected and what the underlying error (mistake or misconception) was.

The errors made by students on the 2005 course were determined by examining the explanations provided by the student(s) in the form of code comments and the report explaining their reasoning. This information was collected to determine whether the defects were caused by slips in the implementation phase, design errors or misconceptions about concurrency or programming in general. In practice, only a few causes could be *confirmed* based on the students' explanations; in many cases, due to insufficient explanations, only a probable cause or probable causes could be determined.

4. PRELIMINARY RESULTS

The following preliminary results are based on the data from the Autumn 2005 course. Roughly half of the defects found (40 %, 60 % and 36 % in the respective assignments) appear to be cases of students misinterpreting what they are supposed to achieve. In the first assignment, only 15 % of these apparent misinterpretations could be confirmed based on student's explanations in their reports and comments (the others may be e.g. slips), but 44 % of them were confirmed in the second and third assignments. Based on the observed defects and explanations, most goal misunderstandings involve writing solutions that work correctly but use inter-thread communication methods other than those allowed in the assignment. In the second assignment, misconceptions about the Reactor pattern [5] are also clearly a problem. The latter can probably be mitigated by providing clearer material on the Reactor pattern. The former can be strongly discouraged by changing the environment in which the assignments are done to reflect the requirements instead of having seemingly arbitrary limitations on what the student may do (e.g. by setting up a real distributed tuple space in assignment 3 instead of requiring the student to write code that runs in a single process but communicates only through the tuple space). Such modifications have been made to the 2006 assignments; the defect statistics will show whether the modifications are successful. Decreasing the amount of misunderstandings of the goals of the assignments is useful in many ways: it allows information relevant to debugging of concurrent programs to be gathered more effectively (less irrelevant defects) and less of students' and teaching assistants' time is wasted on problems irrelevant to the learning goals of the course.

The simple train simulator used in the first assignment is directly related to 41 % of the defects in that assignment; 89 % of these are off-by-one errors in sensor positioning with many possible underlying causes. Only a few (5 %) of these could be confirmed to be misunderstandings of simulator behaviour or slips. This assignment also had the lowest proportion (12 %) of defects clearly related to concurrency (resource allocation between trains and other incorrect assumptions about interactions between trains). The other two assignments had 29 % (52 % confirmed) and 27 % (22 % confirmed) respectively: mostly failures to take into account all possible orderings of operations in different processes or limit them using synchronisation constructs.

The number of defects for which an exact error could be confirmed was quite low (23 %, 45 % and 34 % for the respective assignments); this suggests that asking students to explain the reasoning behind their entire solution in a written report does not give enough information to reconstruct their errors. In order to improve this in the 2006 data, students submitting corrected code after failing the assignment will be required to explain the reasoning behind the defective code. In order to get more information about their debugging methodology and the difficulties they face, they must also provide information about the measures they took (tools and debugging approach used, time taken) to track down and correct the defects.

5. ACKNOWLEDGEMENTS

This work was supported by the Academy of Finland (under grant number 210947) and Tekniikan edistämissäätiö.

6. REFERENCES

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

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.