When Program Comprehension Met Bug Fixing

Jochen Quante

Corporate Sector Research and Advance Engineering
Robert Bosch GmbH, Stuttgart, Germany

Jochen.Quante@de.bosch.com

Abstract

Localizing and fixing bugs requires a certain amount of program understanding to be successful. In this paper, we report about a recently conducted "program comprehension challenge", where the task was to find and fix bugs – but the focus was on program comprehension. Some participants used fault localization techniques, others built on different kinds of static analysis techniques. We present the approaches and results of the challenge, and discuss the relation between program comprehension, fault localization, and bug fixing.

1 Introduction

Last year's International Conference on Program Comprehension (ICPC) gave the program comprehension community the opportunity to analyze some (artificial, but realistic) embedded code and see what they can find out about it with their approaches and tools [1][1]. Bosch provided an artificial robot leg control software, along with a simulation environment and some bug reports. The challenge for the participants was to find and fix all the bugs from the bug reports. The underlying idea was that a certain level of program understanding would be required to solve this task. Our expectation was that people from the program comprehension community would try their approaches on this code – a kind of code that is usually not available to the scientific community. However, we also got some submissions using automated fault localization.

2 The Challenge

The subject software was a robot controller. It was realized as an artificially created piece of C code that contained typical elements of embedded control software, such as filters, ramps, curves, and lots of application parameters [4]. As a part of the story, the documentation of this software was lost a long time ago (by an unfortunate combination of mischances), and the original developers are no longer available. Meanwhile, many people have changed the code, and no one really understands it any more. And this is where our participants stepped in. The participants got the following artefacts to work on:

- The robot leg controller code (∼300 lines of C code, ∼35 application parameters),
- a test environment, consisting of a simulation of the robot leg and a test driver (∼600 lines of C code),
- three bug reports, along with test cases that show the erroneous behaviour and the corresponding log files,
- documentation (i. e., description of general idea and application parameters), and
- an acceptance test script to validate correct controller output.

The challenge was fourfold: Participants were asked to find the bugs, fix them, explain the fix to all stakeholders, and show how they used any techniques or tools for finding and repairing the bug.

The three bugs were the following:

- Exchanged +/− signs in a calculation.
- An erroneous configuration, i. e., two parameters were set in an incompatible way.
- Erroneous use of a library function, or rather a library function that behaves differently from what would be expected.

3 Submissions

We received five submissions. Table 1 shows an overview. The most successful of them (A) was based on classical program analysis techniques such as abstract interpretation, tracing, and slicing. The participant's description showed that he gained a quite good understanding of the code, and that these techniques helped him to focus his attention on the relevant parts of the code. However, using these techniques and using the tool (Frama-C[2]) requires a good amount of expert knowledge.

Participant B used a commercial program analysis tool[3] for extracting projections of the control flow graph. However, he interpreted the tool's output in a

[1] The challenge description and material can be found at `http://icpc2011.cs.usask.ca/conf_site/IndustrialTrack.html`

[2] `http://frama-c.com/`

[3] `http://www.sgvsarc.com/Prods/CREVS/Crystal_REVS.htm`

	A	B	C	D	E
	Frama-C value analysis, abstract interpretation, tracing, slicing	Crystal REVS for C++ flow graphs, plots of trace	Diff of test drivers and configuration, plots	Ochiai bug localization, delta debugging	Test case generation, Ochiai bug localization
Bug #1	Fixed fault	Fixed symptom	Fixed fault	Fixed fault	Inelegant fix, fault still there
Bug #2	Fixed fault	Fixed fault	Fixed fault & superfluous	Fixed fault	"Fix" introduces new faults
Bug #3	Provided workaround	Fixed symptom	Provided workaround	Fixed Symptom	Fixed symptom & superfluous fix
Understanding	Good	Very limited	Manual	None	None

Table 1: Overview of submissions and results.

wrong way: It did not really show data flows (what they thought), but only reduced the control flow graph to nodes that contain a certain variable. This lead to an invalid reduction of the search space, and subsequently, the participant did not find the real underlying fault for two of the bugs. Such a tool could have been useful for understanding the code, but it would have required a better understanding of the capabilities of the tool and of the meaning of its results.

The third submission (C) was based on a quite simple technique and resulted in astonishingly good results. This participant used a simple differencing technique – in fact, a variant of delta debugging. He first checks for differences in the code and configuration of a faulty version compared to a working version, then removes these differences step by step. At some point, the error disappears, therefore the cause has to do with this last removed difference. In a subsequent step, the surroundings of this difference (e. g., the places where the differing parameter is used) are checked manually in detail. This is the key to success: He does not stop when the working replacement has been found, but rather starts detailed program comprehension at this point. The technique delivers some good starting points for that and thus effectively reduces the search space.

Submissions D and E both used the spectrum-based bug localization technique Ochiai [2, 3]. This technique analyzes the execution profiles of positive and negative test cases and then identifies those parts of the program that are primarily executed by failed test cases. Participant D successfully used this technique to locate one of the faults. For the other two bugs, he used delta debugging [5]. This technique was successful in one case, but not successful in the other case. The author of submission E stated that he also used the Ochiai technique. However, he was not successful and provided bug fixes that either did not fix the bugs or introduced new ones. Unfortunately, he did not provide a description of how exactly he proceeded to locate the faults. This shows that the success of using fault localization techniques depends on the people who use it.

Overall, the results are quite diverse. A lot seems to depend on the user: He has to know enough about the used techniques in order to apply them in a successful way. Also, those participants who gained a good understanding of the program – even when it was reduced to the relevant parts – provided the best bug fixes. Others, who did not invest much into program understanding, often fixed only symptoms or even introduced additional errors.

4 Conclusion

The different submissions showed that developers cannot successfully fix bugs when they have not understood the code. From the technical side, (abstract) interpretation, tracing, slicing, and differencing provided the most helpful information, which provides guidance for future program understanding research. Also, some bug localization techniques were successfully applied – but there need to be guidelines and heuristics for how to use these techniques. Blind adoption of advanced techniques is not a good idea and can lead to disastrous results. A certain amount of program understanding is inevitable to assure the correctness of a fix.

References

[1] A. Begel and J. Quante. Industrial program comprehension challenge 2011: Archeology and anthropology of embedded control systems. pages 227–229, 2011.

[2] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Proc. of ESEC/FSE 97, LNCS, Vol. 1301*, pages 432–449, 1997.

[3] A. J. v. G. Rui Abreu, Peter Zoeteweij. On the accuracy of spectrum-based fault localization. In *Proc. of Testing: Academic and Industrial Conference – Practice and Research Techniques*, pages 89–98, 2007.

[4] V. Schulte-Coerne, A. Thums, and J. Quante. Challenges in reengineering automotive software. In *Proc. of 13th CSMR*, pages 315–316, 2009.

[5] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.