Industrial Program Comprehension Challenge 2011: Archaeology and Anthropology of Embedded Control Systems

Andrew Begel
Microsoft Research
Redmond, WA, USA
Email: andrew.begel@microsoft.com

Jochen Quante
Robert Bosch GmbH, Corporate Research
Stuttgart, Germany
Email: jochen.quante@de.bosch.com

***Abstract*—The Industrial Program Comprehension Challenge is a two-year-old track of the International Conference on Program Comprehension that provides a venue for researchers and industrial practitioners to communicate about new research directions that can help address real world problems. This year, 2011, a scenario-based challenge was created to inspire researchers to apply the best "archaelogical" techniques for understanding the complexity of industrial software, and foster appreciation for the delicate "anthropological" scenario which drives the behavior of the software engineers, management, and customers. Participants had two months to work on the challenge and submit writeups of their solutions. The best submissions were presented and discussed at the conference. This new challenge format gives researchers the opportunity to present their novel techniques, tools and ideas to the community.**

I. Introduction

ICPC's Industrial Program Comprehension Challenge is an opportunity for researchers to gain insight into real world problems faced by engineers in the software industry and develop new ideas for their program comprehension research. This year's industrial challenge explored the domain of embedded software programs in a realistic social context.

Outside of modeling and code generation, embedded software is rarely encountered in software engineering research, yet, given sheer numbers, it is possibly the most prevalent kind of software running in the world today. Developed over the last half-century, much of it is legacy code whose original authors and documentation have long since disappeared. When bugs arise in code that cannot easily be abandoned or rewritten, new software developers must understand the design intent of the original algorithms, how these designs were translated into code, the normal, correct behavior of the software in the context of its surrounding hardware, how to reproduce the bug and localize it in the software, and fix it, respecting the impact that changes in the code will have for the system as a whole.

The subject of this year's challenge was a realistic control algorithm for an artificial robotic leg which exhibited three erroneous behaviors that customers objected to, the worst of which was to blow up the robot leg. The research community was invited to develop and demonstrate program understanding and visualization tools that could help developers fix these bugs, explain how the use of these tools might fit into the software process lifecycle, and help corporations responsible for the buggy software to mollify their customers.

II. The Challenge Scenario

There is a (ficticious) company that has been writing embedded control system software for over 60 years. Some controller products that are still sold by the company and contain fairly ancient software code, have recently become popular as controllers for robotic legs. Three (fictitious) companies who sell robot legs based on controllers have reported problems they believe are caused by bugs in the controller software, the worst of which causes any robot leg that experiences the bug to explode into many shrapnel-like pieces. Assuming the role of the software engineer assigned to look into these very high priority bugs, challenge participants were charged with the challenge of finding the cause of the bugs, fixing them, and explaining the fixes to several important stakeholders: the engineer's manager, the customer support engineer, the customer's robot leg project leader, and the CEO of the customer's company.

Along with the software code, researchers were provided with customer configuration files, (realistically) minimal software documentation, bug reports describing the customers' robot leg commands and the problem behaviors that ensued, and log files of test runs that illustrate incorrect and correct controller behavior.

This challenge required quite a bit of progam understanding, primarily due to the obsfucation of control flow in the software code caused by its designed reliance on a clock-based timestep, and by its manual low-level implementation from a specification that itself was manually developed from a high-level abstract model of a controller algorithm.

Researchers were also asked to find or create a software tool or method that could be used to help an engineer visualize or understand the program, its algorithm, its runtime behavior, and/or the bugs. The use of tool had to be documented and submitted by the researcher.

The final challenge for researchers was to address the software's stakeholders and expose them to the issues faced by company representatives when they must grapple with the sometimes unpleasant, real-world consequences of supporting buggy software.

III. The Software

The challenge software is an embedded control system for directing motors that move a robot leg. Typically, control algorithms are first formulated and simulated in a graphical modeling environment. They are then translated to a specification and implemented in a low-level systems language (e. g., C or a restricted subset) that is compiled to run on a controller chip. The resulting systems programs often are hard to understand because the hand-written, imperative software implementations tend to obscure the data-flow nature of the original controller strategies.

The code in the challenge is completely artificial, but is very similar in nature to embedded control software found in all sorts of electro-mechanical devices in use today. Several aspects of the controller algorithm in the challenge are similar to those found in the automotive industry [1]:

Blackboard architecture: These systems typically use a blackboard architecture. Inside each system module, information ("messages") from other parts of the system are directly read from global variables. Information intended for other modules is written *to* global variables. The access rights to these variables may be restricted, but this is specified and checked outside the code.

Time slices: Embedded controllers are often real-time systems, which use time slices. Every function is repeatedly invoked, e. g., every 5 or 10 milliseconds. Not only does information flow to and from the other functions, but it flows to and from the previous and next invocations of the function. In the challenge code, this functionality is simulated by a loop in the main function. However, it ignores the complexity that arises when many other functions are called in between and in different overlapping time slices.

High level of variability and configurability: Since the same basic software code has to work for a variety of customer environments and needs, it is designed to be highly configurable. Global variables ending with "`_PARAM`" or "`_CURVE`" are placed in the program to be adjusted by the customer when the code runs as part of an actual controller.

Control systems: The functionality of controllers is often based on data-flow algorithms and elements from control theory. We have included some of these elements in the challenge code, for example, low pass filters, ramps, and timing components. The controller's purpose usually is to influence the controlled dynamic system in a way that it reaches and stays in a desired (dynamically adjustable) reference state — in this case, a particular position of the robot leg.

The final components of the challenge code are some required library functions, a test driver, and a robot leg simulator for testing the control function in isolation.

IV. Submissions

This report went to print before submissions were sent in to the challenge organizers. Valid submissions had to have three components, though partial solutions were accepted if they used an interesting approach.

1) Identify the cause of each of three bugs found in the controller code, as evidenced by logs of incorrect behavior created by the robot leg simulator. Include diffs for the fixes for each of the bugs and demonstrate, with new logs, that the customers' bugs have been fixed without having changed the customers' control instructions.

2) Build a new software tool, reuse a previously-built tool, or find a software tool written by someone else that can help a software engineer understand the code and its behavior. It can take any form, including a program understanding tool, a program visualization tool or something else. A report must be written (with screenshots) that documents how the tool was used to comprehend the challenge program and fix the bugs.

3) Compose four emails to various stakeholders that document how the bug was found, understood, and fixed. Each stakeholder represents a different audience that software engineers in industry must communicate with about the software development and maintenance process. These emails have to contain a significant amount of technical detail, presented appropriately for the audience — for example, low-level programming and debugging details for the manager and the customer's project leader, and user-oriented, behaviorally-focused explanations for the customer support representative and the customer company's CEO. Creativity in the messages was encouraged to inspire challenge participants to fully flesh out the scenario for themselves.

The authors of all acceptable solutions were invited to present a poster of their solution at the ICPC conference, and were able to personally demonstrate their solutions and tools to all attendees at the Industrial Challenge session.

V. Conclusion

Industrial challenges like this one can be beneficial for all participating parties. Researchers may promote their solutions, discover new research topics to explore, and gain an appreciate for the complexities of the social context in which these problems arise. Practitioners may learn about new techniques or tools that could be directly applicable to problems they see in their own work. Finally, since a large amount of legacy software using similar structures to the challenge software runs in many contemporary embedded systems, embedded control systems engineers would benefit from improvements in the state-of-the-art for understanding such code during their maintenance and reengineering activities.

References

[1] V. Schulte-Coerne, A. Thums, and J. Quante, "Challenges in reengineering automotive software," in *Proc. of 13th Conf. on Software Maintenance and Reengineering (CSMR)*, 2009, pp. 315–316.