Industrial Challenge: Robots Should Never Explode!

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 International Conference on Program Comprehension is hosting an industrial challenge. The topic of this year's contest is "Robots Should Never Explode!" A robot leg controller is running a C program written by your company many years ago. The program is the implementation of an embedded control system which uses a data-flow algorithm to direct the robot leg to move from one desired angle to the next. The program is faulty and three customers have reported bugs, the most serious of which causes the robot leg to explode!**

We invite researchers and practitioners to learn all they can about this software, fix the bugs, and save your company from expensive litigation. You must show off what you know, along with any software tools you used or built to discover or extract that knowledge from the challenge materials. We motivate the challenge with a scenario which offers you a taste of the real-world social situations industrial practitioners find themselves in every day.

***Keywords*-**

I. Overview

This year, the International Conference on Program Comprehension's Industrial Challenge has the theme "Robots Should Never Explode!" Here is the scenario: You are a developer at a leading embedded software design firm (RobotControllers.com).[1] You have been tasked by your manager to fix a high-priority bug in a piece of embedded systems code your company wrote to control a robot leg. This code was licensed to hundreds of leading robot manufacturers, including myAwesomeRobot.com,[2] the maker of the best selling robot of all time, Mr. Awesome Robot III. myAwesomeRobot.com and two other robot manufacturers which use your company's controller have been getting complaints from their customers that in certain situations, their robots' legs malfunction and self-destruct in a cloud of smoke (and possibly harmful particles).[3] Obviously none of these companies like the publicity that these explosions have been causing, nor would they like the government to come in and examine their manufacturing processes more closely. They have each filed an incident report with RobotControllers.com, and since each is a long-time, highly-valued customer, request that a fix for their bugs be delivered by the ICPC 2011 challenge deadline, May 31, 2011.

[1]Not a real company.
[2]Not a real company.
[3]Lawsuit pending.

II. The Challenge

Your challenge, should you choose to accept it, is to 1) find the bug, 2) fix it, and 3) explain the fix to all stakeholders: your manager, the RobotControllers.com quality and assurance support person, the myAwesomeRobot.com robot product leader (and the leads of the other two companies reporting similar problems), and the myAwesomeRobot.com CEO, and convince them that the bug can never happen again.

If you believe that you can accomplish these tasks, the engineering leadership team at RobotControllers.com would like you to show them what tools or techniques (e. g., software visualization or testing tools) could help developers like you to understand the code and fix the bugs more easily. As a fourth requirement of the challenge, you must demonstrate how you used a new or pre-existing software analysis or testing tool to help diagnose and repair the bug.

III. The Software

The robot leg controller chips sold by your company contain embedded control systems code. What you know about embedded control systems design is that strategies for controlling a piece of hardware are typically 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. In the past, you have been puzzled by the low-level systems code because the (usually) hand-written, imperative implementation tends to obscure the data-flow nature of the original controller strategy.

Early in your investigation of the buggy code, you discovered that the controller was designed in 1984 by a now deceased control systems engineer[4] on an IBM PCjr computer. He wrote a specification for the controller's data-flow logic[5] and a second engineer (happily retired in Boca Raton, Florida) implemented it in C. Fortunately, the code

[4]The engineer died tragically in a department store mishap involving several animatronic mannequins.

[5]The specification was sadly lost in 1996 when both the hard drive and all backup tapes containing it suspiciously disappeared. All paper copies of the specification were "found" by the night janitor, shredded, the morning after the disappearance.

has been "modernized" over the years to compile and run on the latest embedded controllers and microcomputers.

Even without the controller's specification, your tacit knowledge gained from several years of employment at RobotControllers.com enables you to know that most of the controllers your company builds use a blackboard architecture. Inside each module, information about the various system components is read and written using global variables visible to all components. Access rights to these variables are specified and checked outside of the codebase. To ensure that customers can customize the controller's behavior to their robots' needs, a special set of customer-supplied configuration variables may be compiled into the binary. These variables' names all end in _PARAM. There may also be characteristic curve variables (whose names end in _CURVE) that can be adjusted to specific configurations.

The controller's purpose is to cause a motor to move a robotic leg to a particular angle. To make movements smooth, ensure a long life of the motor, and fulfill a number of other non-functional requirements, the motor has to be controlled in specific ways. Unfortunately, the details of that are, shall we say, "buried" in the code. As this is a controlled dynamic system, elements from control theory, such as low pass filters (e. g. PT1_Filter() in impls.c), are used to ensure that the leg reaches and stays in its (dynamically adjustable) desired state. The elements are signaled by the current state of the leg, as well as historical information about the state of the motor and the leg.

Your company's controllers have real-time guarantees, so their execution is tied to time slices. Functions in the code are called every few milliseconds, providing them enough time to read from the global variables, compute instructions for the hardware, and write to the global variables in order to pass information to the next time slice. Your controller testing framework simulates this real-time execution with a loop in main().

IV. Contest Requirements

The customers' bug reports each include the configuration file they use for their robots, and a description of the robot leg controller commands they determined lead to incorrect behavior (the worst of which blows up the robot). Your test driver can reproduce these situations on a robot leg simulator your company built to help (safely) diagnose bugs. Each execution of the simulation produces a table of values that report the time slice, engine voltage, leg angle, a boolean that indicates if the leg angle is invalid (meaning the robot leg sensor has not yet been calibrated), and a boolean that tells you if the robot controller is currently active.

To fix the bugs, you may modify any of the code in the controller and the customer's configuration values, but you may not change the commands issued to the robot leg. These have been written explicitly by the customers to enable their robots usage scenarios. After you fix each bug, execute the program, and use a Python script supplied with the contest package to check your output for compliance with the customers' commands and report any errors.

As part of your bug-fixing process, you must build a new software tool, reuse a tool you built before, or find a software tool written by someone else, that can help you understand the code and its behavior. It can be a program understanding tool, a program visualization tool, or perhaps, an actual robot leg… You may use whatever you like, however, you must document how you used the tool to comprehend the program and fix the bugs. Please include full-resolution screenshots[6] of the tool in action.

You guess that helpful visualization tools would reveal how data flows through the imperative low-level code. Better tools might show how the data flows interact with data dependencies during program execution. Since a lot of information is passed from one time slice to the next, there may be value in filtering the information flow to enable you to concentrate on semantically-related clusters of dependent variables. Extracting the potential program states and demonstrating how they relate to one another may help identify design flaws. Better test drivers might systematically, but efficiently, vary the values of the (many) external configuration variables over numerous test runs to explore their influence on the robot leg's behavior. Linguistic analysis tools may try to extract domain knowledge from the program to identify how such a (possibly generic) controller algorithm may have been selectively customized for its robot leg application.

Once you have fixed the bugs, you must write four emails that document what you did.

1) **Your manager**: Explain in technical detail how the bugs manifested themselves in the code, how you tried to debug them, how long it took you to fix them, and how you are sure that your fixes are correct.
2) **RobotControllers.com Quality Assurance**: Explain to this non-engineer how the code caused the customers' problems, and what effect your bug fixes had on the program's output.
3) **myAwesomeRobot.com Robot Project Leader**: Describe in terms suitable for an engineering manager[7] how the bug caused the robot leg to explode. Explain the fix, and convince him that because of this fix, the bad behavior could not possibly happen ever again.
4) **myAwesomeRobot.com CEO**: Since the lawsuits filed over the robot explosions have the potential to bankrupt myAwesomeRobot.com, write a letter to the CEO of myAwesomeRobot.com explaining who you are, how you came to be assigned to his company's bug, what you did (in layman's terms) to fix the

[6] We want to be able to read the text that is on the screen.

[7] He spent 10 years programming in C, but for the last 10 years has only overseen code being written; he has not written any by himself.

problem, and how you have changed the engineering culture at your company to require the use of your program comprehension software tool to fully understand the behaviors of all future robot leg controller software. Finally, apologize for the error, and convince him not to drop your company for your nearest competitor as the sole supplier of robot leg controllers. In this letter, you can be creative, and invent whatever backstory and marketplace dynamics you need to create a coherent narrative.

V. Logistics

The contest package includes the problematic code (artificially created, but similar in nature to industrial embedded software code), the three reported bugs, developer documentation, and sample program executions that exhibit both good, and bad, behavior. An acceptance test written in Python accompanies the package to validate correct controller output from the test driver.

A completed challenge submission should include the diff of your fixed source code, the corrected output logs for the three bugs, a writeup of your use and/or creation of a software tool (including screenshots) that helped you understand the code and solve the bugs, and the four emails you wrote to explain all of this to the people in charge. If you only have a partial solution, please submit it anyway – you might just have an interesting approach to propose.

The deadline for entry is May 31, 2011. Submit your package by email to the Industrial Challenge authors, Andrew Begel and Jochen Quante. Winning entries will be chosen by June 10, 2011. We encourage you to submit early and often before May 31, in case we identify any significant deficiencies in your submission that need to be corrected.

All acceptable entries will be invited to present a poster of their solution and tool at the ICPC conference, and will have a chance to personally demonstrate their solution and tool to all attendees at the Industrial Challenge session.

Good luck!