IEEE DISTRIBUTED SYSTEMS ONLINE 1541-4922 © 2006 Published by the IEEE Computer Society
Vol. 7, No. 5; May 2006

From the Editor: Real-Time and Embedded Systems--Teaching Reliability

John Regehr, *University of Utah*

Can we teach students to build reliable embedded software? Although it would be rash to say that a general agreement exists on how to teach embedded systems, there's certainly a growing understanding of the issues. For example, the excellent August 2005 issue of *ACM Transactions on Embedded Computing Systems* devoted 182 pages to embedded systems education. However, surprisingly few of these pages discuss the problem of teaching students to build reliable software systems.

Why is that? For one thing, reliability is hard to quantify; for another, computer science education has endemic problems in teaching software reliability. I argue that reliability should be a first-order concern in embedded systems education instead of an afterthought.

The problems

Computer science students routinely practice just-in-time software engineering that results in solutions to programming assignments that barely limp through the test cases. Worse, when they have access to the test suite used for grading, students who have reached an impasse will often resort to a kind of evolutionary programming where they incrementally tweak parts of a program, test the code, and repeat. This random walk through the program space can move programs away from correctness rather than toward it.

Of course, the students don't deserve all of the blame, or even most of it. A coding-before-thinking approach to solving programming problems is a rational time- and energy-saving strategy that intelligent students appear to develop early in their careers. This happens in response to the countless toy programming assignments in first- and second-year programming courses, where the approach works quite well. In other words, if the project is small and well defined, if there's no real possibility of unforeseen interactions between components, and if the consequences of a bug are low (because the edit-compile-test cycle is rapid and because nobody's actually going to use the software), then why not just jump in and hack, cowboy-style?

Group projects tend to exacerbate the cowboy programming problem. Commonly, when a project isn't too difficult and when there's a significant programming skill gap between the strongest and weakest students in a group, the far-and-away most efficient route to a working project is for the best programmer in the group to just do everything. An instructor who's not paying close attention will miss this because the students have no incentive to fess up.

Possible solutions

It's not forward-thinking to produce students who know a single programming style—the cowboy approach—and an embedded systems course is a good place to show them a better way. Clearly, the motivation exists: you can't upgrade most embedded systems after deployment, and embedded software is finding its way into more and more safety-critical applications. Furthermore, it seems clear that software liability will increase in the future.

So how can we encourage students to think before hacking? To design a system rather than being surprised at how it evolves? To test software aggressively and skeptically? Of course, no single answer exists, but a variety of partial solutions, taken together, seem to work pretty well. They can augment existing embedded programming assignments, and they can be fun.

Code reviews

After turning in a programming assignment, each group of students prepares a short presentation of their design and implementation strategies. They have to show actual code. Subsequently, the class gets to ask questions about the design and implementation with the goal of satisfying themselves—or not—that the group has created a solid, efficient, maintainable system. Generally, no moderating is necessary to keep the process constructive.

Correctness arguments

Rather than simply demonstrating a working system, I require students to convince the TA or me that they have met certain correctness requirements. For example, if there's a requirement that idle time exists between successive iterations of a signal-processing loop, students must demonstrate this by attaching a logic analyzer to their running system. If correctness depends on the interrupt stack not overflowing, they must count stack memory usage on all paths through their compiled code to argue that it doesn't exceed the allowable bound. One of the insights behind this requirement is that if students know ahead of time that they must make these correctness arguments, they start to take this into account while designing and implementing their system. Simple designs are easier to reason about. This is absolutely a step in the right direction.

Using tools

Software correctness tools can ferret out difficult bugs in programs. A reasonable start is to require students to use assertions and to create software that generates no compiler warnings at the maximum warning level. In one class, I required undergraduates to use a model checker to show that their synchronization protocol was correct before I permitted them to implement it. This implicitly discouraged overly complex solutions, for which the model checker wouldn't terminate. Also, it eliminated the problem of synchronization protocols that are obviously incorrect but pass the test cases. This worked pretty well, and the students were impressed with the powerful way in which the model checker could break their code.

Emphasizing testing

Software testing techniques, such as boundary testing, random testing, and stress testing, are still an art form. These techniques only work well when the tester genuinely wants to make the system fail. Because forcing students to wish to break their own code is hard, a possibility is to have them test other groups' code and to give credit for finding bugs.

Emphasizing interoperation

Small groups of students can usually create working systems. However, interactions between groups seem to result in problems out of proportion to the task's actual difficulty. In one embedded software course, I let the students act as a standards body tasked with defining protocols such as data aggregation and dynamic leader election for wireless sensor network nodes. These started out simple but ended up involving multihop routing and other complications. After defining the standards, the students implemented them separately. Then we had bake-offs where we placed collections of nodes in proximity and checked them for conformance. The resulting interactions were priceless. Students were constantly upgrading their protocol implementations, devising experiments to pinpoint incorrect nodes, and bickering back and forth. Afterwards, we all felt that a lot of learning had occurred.

Understanding software architectures

When students feel an intuitive need for some feature such as threads, heap allocation, priority inheritance, or nested interrupts, they will likely start using the feature without fully working out the consequences of their choice. I devote a considerable fraction of my embedded software course to describing the available embedded software architectures, each of which comes with a very specific set of tradeoffs. The lesson I try to impart is that the choice of software architecture should be deliberate, rather than evolving out of a sequence of greedy choices. This is a fundamental lesson of embedded software development.

Learning from failure

Embedded systems, with their concurrency, resource limitations, flaky tools, and all-too-frequent debugging through LEDs and logic analyzers, provide the perfect environment for students to experience some truly difficult debugging. When a talented student is burned by an error that requires days to debug, he or she will remember the experience for a long time. After being burned by perhaps a dozen such problems, the student is probably well on the way to recognizing the need for a careful development approach emphasizing assertions, testing, tools, incremental implementation, and thinking in advance. Since a dozen such problems is perhaps too many for a single semester, I humbly hope that each student has two or three such experiences while taking my class.

Conclusion

Embedded systems must be reliable, but computer science students aren't in the habit of creating reliable software. Our graduates will be better citizens and more valuable developers if we explicitly teach them techniques that can increase software reliability—even though this reduces the amount of time we can spend on more traditional technical material.

John Regehr is an assistant professor in the University of Utah's School of Computing. Contact him at regehr@cs.utah.edu.

Related Links

- DS Online's Real-Time and Embedded Systems Community, cms:/dsonline/topics/embedded/index.xml
- DS Online's Education Archives, http://dsonline.computer.org /portal/site/dsonline/menuitem.bc6cc7000140 e2ec2587e0606bcd45f3/index .jsp?&pName=dso_level1_article_list &TheCat=1050&
- "Reliability Evaluation for Dependable Embedded System Specifications: An Approach Based on DSPN", http://doi.ieeecomputersociety.org/10.1109/MEMCOD.2003.1210102

Cite this article: John Regehr, "Teaching Reliability," *IEEE Distributed Systems Online*, vol. 7, no. 5, 2006, art. no. 0605-o5002.