Safety Critical Coding Standards Reduce Medical Device Risks

Paul Anderson

Software is making its way into a variety of medical devices to help offer patients a higher degree of effectiveness from these devices. With the benefits the software provide, there also comes a greater chance for errors in the manufacture of the device, which can lead to costly delays or even product recalls. This article reviews coding standards to help ensure that software used in medical devices is compliant.

Paul Anderson is VP of engineering at GrammaTech, a spin-off of Cornell University that specializes in static analysis. He has worked in the software industry for 16 years, with most of his experience focused on developing static analysis, automated-testing, and program-transformation tools. Anderson can be reached at paul@grammatech.com [1].

Due to the growing use of software in medical devices, the proportion of device failures that can be attributed to software is increasing. A recent study of recalls using FDA data between 1999 and 2005 found that one in three recalls of devices that use software were due to errors in the software itself.[1] The consequences of device failure are potentially dire, as witnessed by the infamous failure of the Therac-25 radiation therapy machine, which resulted in several deaths.[2] However, even if lives are not at risk, the cost of issuing a recall can be crippling. As the use of software continues to grow, and as software itself becomes more complex, it is becoming more important to reduce this risk.

In other safety-critical domains, such as avionics, industrial control, or automotive, where the cost of failure is potentially even worse, software development activities have emerged aimed at reducing the risks of introducing software defects. Among these are standards for how code should be written. In the motor vehicle industry, the Misra-C rules for programming in C have been widely adopted,[3] and have recently been extended to cover C++ too, drawing heavily on rules developed for use on the Joint Strike Fighter.[4]

The rationale for most of these rules is to improve code clarity. Code is read more

often than it is written, and as code reviews are a very effective way to find defects, adoption of these rules makes it easier for humans to spot problems early. A second important rationale is to eliminate sources of unpredictability. The specifications for the C and C++ languages are both incomplete and ambiguous, so compilers are free to make choices that may introduce unpredictability. Other rationales are to encourage simplicity, which can make code easier to test, to foster defensive programming, and to support the use of standards.

The Power of 10 rules were developed at the NASA Jet Propulsion Laboratory for use in software developed for space vehicles.[5] This is a set of ten simple rules for the C programming language. The motivation for the small number of rules is that it is easy for programmers to remember ten broad rules as opposed to hundreds of very specific rules. The Power of 10 rules were also designed to take maximum advantage of automation. The rules are designed in a way so that violations can be detected automatically by static analysis tools.

The tenth rule codifies this principle in two ways. The first part states that the compiler should be used at its most pedantic setting, and that all warnings generated be fixed. This may seem like overkill, but it forces the programmer to use maximum care when writing code, and fosters good programming practices.

The second aspect of the tenth Power of 10 rule is that it mandates the use of advanced static analysis tools. Static analysis tools have had a checkered history. The first-generation tools, such as lint, were capable of finding some superficial lexical syntactic issues, but were difficult to use because they suffered from an excessive rate of false positives—warnings issued in error. Due to this problem, and their limited scope, their effectiveness was minimized.

However, a new generation of static analysis tools has emerged that are much more sophisticated. One such tool is CodeSonar. It can find serious semantic errors in code, such as buffer overruns, null pointer dereferences, race conditions, and resource leaks. Such defects are examples of where the fundamental rules of the language are being violated, or where an API is being used in an incorrect way. They can also find places in the code that are inconsistent or that indicate contradictory assumptions. Although such occurrences are not bugs *per se*, they do correlate well with real bugs because they often signify that the programmer has not understood some important property of the code.

These new static analysis tools are very effective at finding defects. They are generally easy to integrate into the development process and produce results that are important and actionable. A key factor contributing to their effectiveness is that they have a low false-positive rate.

The rationale for the tenth Power of 10 rule asserts that these tools should be used regularly on the code, and that the code be required to pass the analysis with no warnings generated. This is true even if the warning is a false positive result. If a warning is judged to be a false-positive, then the code should be rewritten so that the analysis considers it correct. This may seem counter-productive, but there is good justification for this approach. Analysis tools generate false positives mainly

because the code is too complex for them to analyze properly. If the tool does get confused in this way, it may consequently miss a real defect too. Rewriting the code to eliminate the false positive will help the tool find true positives. Furthermore, humans can themselves be confused about the code, and it is not uncommon for developers to erroneously label a true positive result as a false positive. This is doubly unfortunate because the defect remains in the code, and the tool will not report it again. Rewriting the code to eliminate the warning removes this possibility from the table.

Safety-critical coding standards, and advanced static analysis tools for finding defects are beginning to be used more widely within the medical devices community. Even regulatory agencies are now using these tools for investigations. Jetley et al report the results of a case study at the FDA where a static analysis tool was used as part of a post-market investigation into a device failure.[6] In this study, the software under scrutiny was approximately 200,000 lines of C/C++ code. The tool identified 127 defects that were judged relevant to the investigation. Of these, 45 had not previously been detected by the manufacturer during their own in-house verification and validation.

It is now generally accepted that the use of coding standards is effective for decreasing the risk of safety-critical development. An excellent adoption strategy is to use the Power of 10 rules as a foundation, and to adopt rules from other sets as necessary. Use automation as much as possible to detect violations, and keep the code simple and straightforward enough to maximize the effectiveness of the tools.

References

[1] Bliznakov, Z., Mitalas, G., and Pallikarakis, N., Analysis and Classification of Medical Device Recalls. *World Congress on Medical Physics and Biomedical Engineering* 2006.

[2] Leveson, N.G., An investigation of the Therac-25 accidents. *IEEE Computer* 1993. 26: p. 18-41.

[3] MISRA-C 2004 Guidelines for the Use of the C Language in Critical Systems, 2004, The Motor Industry Software Reliability Association.

[4] Joint Strike Fighter Air Vehicle C++ Coding Standards, 2005, Lockheed Martin Corporation. Document Number 2RDU00001 Rev C.

[5] Holzmann,G.J., The Power of 10: Rules for Developing Safety-Critical Code. *IEEE Computer*, 2006. 39(6): pp. 95-97.

[6] Jetley,R.P. and Anderson,P., Using Static Analysis to Evaluate Software in Medical Devices. *Embedded Systems Design* April 2008. p. 40-44.

Online

For additional information on the technologies and products discussed in this article, see *MDT* online at www.mdtmag.com [3] or GrammaTech at www.grammatech.com [4].

Source URL (retrieved on *03/10/2014 - 1:57am*):
http://www.mdtmag.com/articles/2009/03/safety-critical-coding-standards-reduce-medical-device-risks

Links:
[1] mailto:paul@grammatech.com
[2] http://www.mdtmag.com/sites/mdtmag.com/files/legacyimages/ProductImages/0903/BufferOverrun_large.jpg
[3] http://www.mdtmag.com
[4] http://www.grammatech.com