Bugs and Ethics

Magnus Björk
mab@cs.chalmers.se
Department of Computing Science
Chalmers University of Technology

December 5, 2002

Abstract

This paper discusses bugs in safety critical software: why they exist, how one can avoid them, and who is responsible. We look more into the failures of the Therac-25 radiation therapy machine and the Ariane 5 launcher.

1 Introduction

During the last decades technology has become more and more dependent on software. Most electronical devices now contain some embedded system that runs a piece of software. At the same time we see a lot of faulty software, containing bugs. This paper adresses the issue of bugs in safety critical software, and who is responsible for them.

With bugs we don't mean just any malfunction of a computer system. There are for instance cases where the one who specified what the software should do thought wrong. When the software fails in those cases we don't talk about bugs. A bug is when the software fails to meet its specification, due to a mistake made by the programmers.

We will begin by looking at some famous software bugs, then the software crisis which is the reason for why bugs occur. Then we'll discuss ethics and software, and finally give a brief overview of some methods that exist today for finding bugs at early stages.

2 Known cases

2.1 The Therac-25 radiation therapy machine

One of the most tragic cases where software bugs has mattered is the radiation therapy machine Terac-25. The machine, which is used for treating cancer patients using X-ray and electron beams, was controlled by an erroneous program, which massively overdosed six people, killing three of them. A long report about the accidents can be found in [Lev95], which is my source for information on this case.

The accidents occured during 1985 to 1987 at four different hospitals in Canada and the US. After the first overdose, the Therac-25 was suspected to have caused the injury, but it was not until after several other accidents that this was established. It was obvious that the Therac-25 caused the second accident, but nobody could reproduce the error. The maker concluded that a faulty microswitch could cause a problem, and introduced a reduntant switch. However, the specific problem could never be reproduced, even with faulty microswitches, and the switches on the erroneous machine were working well. Even so, the company claimed that the safety of the machine had now increased with 10,000,000%.

The third accident was, as the first had been, a mystery until long time later. After both these accidents the maker of the machine claimed that it was impossible for the machine to overdose, and that there had to be some other reason. The fourth accident was obviously caused by the Therac-25, however the maker replied that they didn't know about any other accidents involving radiation overexposure by the machine. This was a bit strange, since they knew about the first three accidents, and did acknowledge that the second one could have been caused by the machine.

It was not until after the fifth accident had occured that somebody managed to identify the problem. The operator who was running the machine as the accident occured remembered clearly what he had done, and after a few day's work he was able to reproduce the overdose. It involved a number of steps, including entering all settings, then going back to correct a mistake and finally starting the treatment, within a short period of time. The maker finally admitted that there was a problem with the software, but as a fix, they recommended all hospitals to remove the "cursor up" key from the keyboard, prohibiting operators from correcting erroneous settings (and instead forcing them to enter all data once again). Also, they updated the error messages, which up until this time, all looked like `MALFUNCTION 54`.

The sixth accident occured only a few months later, and was caused by a completely different bug. The company now rewrote parts of the software, and claimed it to be safe. However, the U.S. Food and Drug Administration (FDA) was not satisfied with this, and required safety mechanisms to be built into the hardware. After these changes were made, no known accidents have occured.

We can identify a number of bad choices in the design of the Therac-25. Leveson [Lev95] makes a longer list, but I select what I consider to be the most important issues.

- **Overconfidence in old code.** Large parts of the source code for Therac-25 had been used in the Therac-6 and Therac-20, and was therefore believed to be safe. However, these machines had hardware safety systems, and knowing about the bugs, it took little effort to reproduce the same problems there. The main difference however was that for the older models, the problems only resulted in a burned out fuse, instead of an overdose.
- **Bad safety analysis.** The maker had done a thorough analysis of what could go wrong. However, with the motivation that software doesn't degrade due to wear, fatigue or reproduction process, they

asserted high safety for the software. As an example, they asserted that the probability that the software selects the wrong energy is one to 10^{-11}. This type of analysis wasn't good for determining the safety of the software, but rather the reliability of the hardware, the likelihood of a fault occuring due to worn out components.

- **Bad programming methodology.** One cannot expect that programmers would use a sophisticated programming methodology in the beginning of the 80's, but the one they used must have been outdated even then. The program was written in such a way that different parts of the program could interfere with each other in many ways, that were hard to find out. Indeed, out of tens of thousands of successful treatments, the bugs could only be identified in six treatments. Clearly, no test based verification could ever have revealed the bugs in time. Another bad programming issue was that the program did not have any functions to help operators identify errors. The error messages were all incomprehensible, and no logs were kept.

2.2 Ariane 5, Flight 501, June 4, 1996

A software malfunction turned the maiden flight of the Ariane 5 launcher into a failure. Some 40 seconds after initiation the launcher self destructed, and equipment worth $500.000.000 was lost. The bug that caused the failure is easy to understand, but was introduced under complex circumstances, and wasn't found until afterwards. After the accident, a thorough investigation was conducted by Jacques-Louis Lions et al. This investigation is described in [Lio96].

The concrete problem was the conversion of a 64-bit real number into a 16-bit integer in a system which read data from various sensors. When performing such a conversion, if the number is too large to fit in a 16-bit integer, an *exception* occurs. The programmer must include instructions on how to handle the exception, but may choose not to do this if he thinks it's impossible for the exception to occur. In that case, if the exception still occurs, the system most often crashes. On windows systems, unhandled exceptions in applications results in an error message similar to "The program ... has performed an illegal operation", or "An exception has occured at position ...", which is probably familiar to many readers.

The reason why the exception occured was that the rocket accelerated faster than the program was designed for, and some readings from a sensor then produced the fatal number which couldn't be stored in a 16-bit integer. The exception was never caught, because that specific part of the system was designed for the Ariane 4, which couldn't accelerate that fast, and had been included unmodified in the Ariane 5. Since a physical limitation of the launcher would make it impossible for the exception to occur, there was never any code included to handle that exception. Here we recognize a mistake from the Therac-25 case: overconfidence in old code. The designers assumed that the code that had always worked for Ariane 4 would still work with Ariane 5, but as with Therac-20 and Therac-25, hardware issues influenced the effects of the software.

But there are still more issues: The process in which the error occured wasn't needed after takeoff, but to make it possible to restart a halted countdown, the process was kept running for 40 seconds after lift off. The exception occured some 30 seconds after lift off. Another issue is that in order not to lose this important system, two identical systems were running in the launcher. The idea was that if one failed, the other one would continue to operate. However, that sort of redundancy only works for hardware failures. Since both systems used the same software, they both failed simultaneously. Actually, the back up system failed 1/20 second before the primary system.

So how could this bug go unnoticed? One explanation is bad specifications. When the system was built for the Ariane 4 launcher, the specifications included the predicted trajectory, and hence specifyed which values that the sensors would reasonably produce. When the system was implemented for Ariane 5, the trajectory wasn't included in the specifications. A simple test, where the software would be run with simulated sensor values would have revealed the bug (and indeed, after the failure such a test produced the exception).

2.3 Comparison of Therac-25 and Ariane 5

As already mentioned, one issue is common in the two cases described above: overconfidence in old code. Both the Therac-25 developers and the Ariane 5 developers reused code from older systems without evaluating their correctness further. In the Therac-25 case this was probably done to save time and money, while in the Ariane 5 case it was a deliberate move to *prevent* bugs. The motivation was that new code is more likely to contain bugs than old code that has been tested. However, nobody noticed that the physical conditions had changed, and therefore the previous use of the software didn't guarantee anything.

Both cases also displays overconfidence in software in general. In their fault analysis, the Therac-25 team assumed that the software wouldn't be any problem, since it would never wear out. The same mistake was made in the Ariane 5, where the back up system used the same software as the main system. When one relies so heavily on one piece of software, then one better make sure that that software is written in such a way that safety can be guaranteed with high accuracy.

There are however notable differences between the two cases. One is that the Ariane 5 team used good methodology for writing the program, by standards of 1996. The Therac-25 team used bad programming methodology, even by 1985 standards. One would then expect the Ariane 5 software to be more reliable, but they failed miserably in one aspect: the verification. While the Therac-25 overdosed six people while successfully treating tens of thousands, the Ariane 5 failed on it's *first* launch. A simple test run, with the physical sensors replaced by simulation devices that produced the data that was expected during the flight, would have revealed the bug. However, the software for Ariane 5 reported errors in a good way, and it was possible to find out which exception that had occured without much work (the on board computers actually worked after the accident, having survived the explosion and the free fall from 4km,

into a swamp), while it took five accidents with the Therac-25 before anybody could reproduce the error.

3 The software crisis

Software engineering is still an immature discipline and *the software crisis* is a well-known concept among software developers (see [Gib94]). Better programming languages and methodologies are constantly developed, but nothing can really cope with the desire to build even larger software systems. There are always many aspects of a software project, and the more advanced a system is, the harder it is to find bugs in it. It is important for software companies and companies working on projects which use software to realize this (see for instance [Bow00]).

When the decision to use software has been made, an analysis of how crucial the functions of the software are, and an appropriate programming style must be selected. One cannot write a word processor with all the bells and whistles that modern word processor comes with, and expect it to be completely free from bugs. People who use word processor must realize this. Indeed, the manufacturers know it, and their response is to sell the software under *limited warranties* (which perhaps should be called *anti-warranties*), where they only promise that the software works mainly as it is supposed to, up to three months after the purchase.

But the main problem doesn't lie in bugs in word processors, although lots of people find them annoying. The most important thing to realize is that safety critical software cannot be written in the same way as non critical software. For non critical software we wish that there are no bugs, however with safety critical software, it is crucial that there are no bugs. Therefore safety critical software must be developed using safe programming methodologies. The control software for the Therac-25 was written in a general purpose programming language, without any fancier programming constructs to increase safety. The problem with the Ariane 5 was not so much the programming tools used, since it was an active decision not to handle the exception that caused the accident. It was rather a problem with relying blindly on old software, without reviewing it completely for its new use. I will discuss later what the current state of the art methods are, but as success stories I would already like to mention the control software of modern Airbus passenger aircrafts, which is written in Lustre and verified formally, and of Paris metro system, which is developed using the B method, which relies heavily on formal verification.

4 Ethics and software

There are many sources for software ethics available, see for instance `http://computingcases.org` and `http://www.onlineethics.org`, and indeed there are many aspects of software ethics. The two major organizations for software engineers, ACM and IEEE, have codes of ethics that members must agree to follow. Another example is *The Ten Comman-*

dements of Computer Ethics, by the Computer Ethics Institute at The Brookings Institution, Washington DC. Many of these codes of ethics are more general, and applies to all uses of a computer. On `onlineethics.org` there is a story called *The Story of the Killer Robot*, which describes a fictitious scenario where a factory worker is killed by a malfunctioning robot. The reason for the malfunction is a software bug, and as the story develops, the blame is put on many different persons involved in the making of the robot: the programmer, his manager, the verification staff, people who decided on the budget, and so on. The purpose of the story is to show the complexity of software development, and how complicated the ethical issues are.

An interesting student project [BDLY96] adresses the issues of liability law and software development. It claims that we don't want to pay the price of risk-free software, and must make legal distinctions between safety critical and normal software applications. In case of an accident, resulting from a software error, someone must be held responsible, and the responsible part, according to [BDLY96], must be the one who had the power to prevent the harm from occuring. This can be both the makers of the program and the one who was using the program. In case of the Therac-25 accidents, we can at least state that the operators were innocent, since the only hint they got that something was wrong was the message `MALFUNCTION 54` (where about 40 similar malfunctions occured each day). But still, we cannot blame anyone more specific than the company that made the software. Neither the individual programmers who wrote the code, the team leaders who chose the programming languages to use, the people who decided to reuse old code, nor the verification staff who tried to find bugs can be blamed separately. Therefore, the approach described in [BDLY96] might be good to determine who should pay fines to whom after an accident has occured, but it doesn't help software developers avoid making bugs.

An ethical framework that proves more useful for the software developers is the ImpactCS framework [Com95]. It describes seven different ethical issues related to software development: Quality of life, use of power, safety, property rights, privacy, equity and access, and honesty and deception. For each new software project, each of these issues must be evaluated at four different levels of social analysis: individual, group, national and global. At `computingcases.org` there is an analysis of the Therac-25 software using ImpactCS, which I will summarize here. In many aspects, the software wasn't unethical, e.g. the aim of the software was actually to aid quality of life in different levels (by building a cheaper radiation therapy machine, and therefore making the treatment available to more patients).

The most interesting ethical issue of ImpactCS in the case of Therac-25 is the safety issue. Lets start by looking at the individual level. The analysis at `computingcases.org` mentions two persons: the programmer and the operator. Of course we can think of other persons, but they are often easy to discard quickly. E.g. the patients couldn't have done anything to prevent the hurt. Another example is the people on the verification team (if such a team existed); the only way they could be held responsible is if they acted negligent, i.e. performing bad tests. However, we don't know anything about them, and therefore cannot include them in our analysis.

But let's consider the programmer (or programmers). What responsibilities did he have, and to whom? His responsibilities to his employer were not only to write his part of the software, but also to make his superiors aware of the dangerous programming style. Maybe he did not realize how dangerous the method used was, or maybe he told his superiors that they wouldn't listen to him. In any case, ImpactCS helps us to see who is responsible easier. The programmer also had responsibilities to the users of the machine, namely to make a safe program. Clearly, he failed here, and could have avoided it by using a safer programming style. The operators of the Therac-25 also had responsibilities, toward the hopsitals and the patients. In general however, we cannot find anything the operators did that let us believe that they failed in their responsibilities.

Let's consider the group level. The company that made the machine and the hospitals are two distinct groups. The company claimed that they had done a safety analysis of the machine, when they had only analyzed the likelihood of failure due to worn out parts. This is an example of a responsibility the company failed to meet. The hospitals had a responsibility to only use safe machines, but they don't have the capability to investigate whether a machine is safe or not, and thus have to rely on the maker's own analysis. At the national level, we can study what the FDA did, but we won't go into that here. At the global level, little can be said. Since some accidents occured in Canada, one can analyse the cooperation between US and Canadian authorities on the global level.

5 Conclusions

Software developers must be aware of the danger of bugs and choose an appropriate programming methodology for each application. Safety critical system must be written in a programming style that allows good verification methods. This choice is really important in an industry where designers often make a passive decision to use the same language that they have been using for many years, or managers encourage programmers to use the latest fashion programming language or methodology that look good in the PR.

6 Current research aimed at making software more reliable

Plenty of research in the field of safe programming methodologies is currently conducted, e.g. at the Department of Computing Science at Chalmers University of Technology. A few examples are:

- New programming languages with stronger type systems, that catch more bugs at compile time, at the same time as they allow more general code in a safe way.
- Formal methods, where tools can automatically check that programs fulfill certain properties, with 100% certainty. For instance, a relevant property of the Therac-25 could have been that the accelerator

must never emit more energy than some upper limit, or that the ray can never be turned on before the machine is in place.

- Automatic testing, where a computer handles the testing of the software. This can be applied where formal methods aren't applicable. A computerized testing means that many more test cases can be evaluated than when humans perform thet tests.

One of the main challenges of the computing science research community is to educate the software developers about these methods.

References

[BDLY96] Ravi Belani, Charles Donovan, Howard Loo, and Jessen Yu, *Liability law and software development*, Student project, 1996, http://cse.stanford.edu/classes/cs201/Projects/liability-law.

[Bow00] Jonathan Bowen, *The ethics of safety-critical systems*, Communications of the ACM **43** (2000), no. 4, 91–97.

[com] *computingcases.org*, website.

[Com95] ImpactCS Steering Committee, *Consequences of computing: A framework for teaching*, Tech. Report May, George Washington University, http://www.student.seas.gwu.edu/ impactcs/, 1995.

[Gib94] W. Wayt Gibbs, *Software's chronic crisis*, Scientific American (1994), 86ff.

[Lev95] Nancy Leveson, Safeware: System Safety and Computers, ch. Appendix A: Medical Devices: The Therac-25, Addison-Wesley, http://www.cs.washington.edu/research/projects/safety/www/papers/therac.ps, 1995.

[Lio96] J. L. Lions, *Ariane 5, flight 501 failure, report by the inquiry board*, Tech. report, Academie des Sciences (France), http://java.sun.com/people/jag/Ariane5.html, July 1996.

[onl] *onlineethics.org*, website.