CHAPTER 10

The Use of Formal Methods

The main area of concern to many safety-related system engineers is software. Reasons are that

- software and firmware is pervasive, part of every E/E/PE subsystem;
- the current state of software engineering does not enable a practical statistically-based approach to assessing software risk at anything like the level we need to meet our hopes for acceptably safe operation [BF93,LS93]. This also includes testing; and
- partly in consequence, the requirements on software concern mostly the software-development process rather than properties of the software itself. That is, the quality of the process used to produce the software rather than the quality of the end product itself. We surely ultimately care about the properties of the software product. But there is no reliably-demonstrated causal connection between process methods and product quality, which depends intuitively not just on what has been done, but on how well it has been done.

I shall assume that independent assessment of the fitness-for-purpose of safety-critical systems is enshrined in local law: that not only is there a developer who wishes to know how appropriate hisher system is, but there is an assessor who has to be convinced. Documentation of fitness-for-purpose in some form is required by the standard, in what many call a «safety case».

Broadly, then, improvement to the standard will occur through two pursuits:

- improving methods for statistical assessment of software risk; and

- requiring that objective properties of the software product be assessed which directly establish its fitness for purpose.

10.1 The Nitty Gritty

Besides these two general ways of improving the standard, there are some more pragmatic considerations for improvement, some lower-hanging fruit. Here is an example.

Methods are recommended for the development of subsystems of various levels of criticality. One of these methods is "Reliability Block Diagrams" (RBD) [3]. RBD is a specific technique for assigning and calculating overall reliabilities of composite system parts given reliabilities of components, and assuming independent failures of those components. Another such method is "Formal Methods", apparently a highly-recommended "technique" for the development of the most critical software. There is no IEC standard on "Formal Methods", and there is a very good reason for this, namely it is not a technique, but a name for a class of techniques which establishes objective properties of software and its documentation through the use of rigorous mathematical methods.

We can, though, be far more specific about Formal Methods. Let's see how.

Suppose you are an engineer with experience in using higher-order formal logic to specify properties of computer systems (there are such tools around, for example the SRI system EHDM or the Cambridge system HOL). You specify the requirements for your system in higher-order formal-logical language, and you also express the system design in the same logical language. You then use a theorem prover, EHDM or PVS or Isabelle, to prove that each requirement is fulfilled by the design. You conclude that the system design indeed fulfils its requirements. Have you applied Formal Methods? Most people would say: yes.

Suppose now you had given the task to a summer intern from the local university, who wanted to learn about specification and verification in PVS. She did a good job, was very pleased with her results and used them as credit for one of her classes. Would that suffice to demonstrate that your company had used "Formal Methods" as recommended by Part 3 of the standard? I suggest that this would be more problematic.

As with any engineering technique, quality control is an issue. We have good human reasons to trust the output of an experienced practitioner dedicated to the task more than the output of a neophyte who is learning on the job. Trust is not infallible, though. Better than trust is checking the results of each project explicitly.

Suppose you write your software design in UML. Have you used "Formal Methods"? Most eminent software engineers of my acquaintance would say: no. One reason is that a UML description does not necessarily have a precisely-defined unique meaning. So if you write such a description, and give it to another engineer to implement, it is not guaranteed that the other engineer will understand what behaviour it is that your UML description is supposed to specify. Suppose you write your software design in UML and use a commercial C-code generator from a trusted supplier to derive C-code directly from the UML, untouched by human hand. Have you used "Formal Methods" as recommended by the IEC 61508 standard? Maybe you have, yes. The C code generator is a functional translator from the UML into executable code. The behaviour of the program thus generated is what is specified by the UML from which you started. But do you have a good intellectual handle on that behaviour, via understanding the UML and the meaning of the C code generated? Maybe; maybe not.

Suppose you had done any, or all, of these things for some safety-related software you were delivering. You have arguable used "formal methods" as ()-highly-recommended by the standard. Does this suffice to assure the quality of your software? Most people would, I hope, say: no.

There are a number of reasons why one says no.

- The engineer experienced with using higher-order logical specification may well have shown that the software-system design fulfils the requirements specification, but nothing is said in that one task about how the system implements the design. That could be well or poorly, and the quality of the software correspondingly high or low.
- The intern may not have captured the requirements correctly, or the design. She may have made mistakes in using PVS, and thought something had checked out which didn't.
- Maybe the requirements specification you gave your engineer and your intern weren't adequate. Maybe they missed out some critical situations which needed to be covered (see for example Chapter 9).

- Maybe the design you used in the product was modified from the design that your engineer/your intern checked. (Say, you had realised during development that an important requirement was missing.)
- If you used UML, maybe different people in your development team had understood the UML description differently (it is an ambiguous medium, after all).
- Maybe the C-code generator you used is faulty in some way which affected the code you generated automatically from the UML.
- Maybe, in any of these cases, the C compiler you used on the generated code wasn't as dependable as you were assuming, and generated object code which behaves subtly differently from the behaviour which someone inspecting the C code would expect.
- Maybe someone had mucked with the microcode on the processor, so that certain object-code instructions didn't work the way the compiler assumes.

Almost all of these situations are well-known to experienced software developers. There are a lot of things which can have gone wrong, despite the best of intentions. "Using formal methods" here and there is not a panacea for any of them.

Just claiming to use "Formal Methods" does not suffice to obtain the benefits of these methods. One must use them in a targeted manner, to achieve certain goals, and be clear about what one has done, what goal one has achieved. One can be more precise about exactly what kinds of formal methods were used, given that many are out there and are mature enough for routine industrial use.

10.2 Using Formal Methods

The Finsbury Group was an informal group of eminent software engineers[1], who met sporadically in London, first at City, University of London, whence the name.

There is the following list of tasks which arise during software development, for which rigorous mathematical methods exist to accomplish those tasks, and those

1 Peter Bishop, Robin Bloomfield, the late John Knight, Peter Ladkin, Bev Littlewood, Lorenzo Strigini, Martyn Thomas; John Rushby joined us in electronic discussion on this theme.

methods are industrially mature[1,2]. The results of performing these tasks establish objective, explicit properties of software and its documentation (and the relation of the software to its documentation).

It was necessary to invent some terminology. I explain why. Somewhere between design specification and object code, there is at least one place at which the sequence of actions taken by the executing processor(s) is written down in something which approximates a "higher-level language". There used to be just one such place. If a program is written in C, or Fortran, or Pascal, there is just one "source code". However, with modern languages there may be more than one stage in development at which this description occurs. Source code in C++ may be first translated into C before being compiled. One could consider either the C++ source, or the resulting C code, as "source code". There are some program development systems in which programmers write "state machines", and the system translates the internal state-machine descriptions into an executable procedural language such as C. Here there are also two possible "source code levels": one is the state-machine description; another is the resulting C code after automatic transformation. If one is programming in Java, then the Java source code can be considered as "source code". But so can the Java Bytecode into which that Java source is translated.

The tasks and objects below are predicated on there being just one source code level, which I call the "Executable Source Code Level", ECSL. If, as in the examples above, there is more than one possibility for "source code", then the ESCL can be taken to be just one of these, which may be chosen arbitrarily (that is, according to criteria which at not relevant to our purpose). Or, one may prefer to label all source code levels, say $ESCL_1$, $ESCL_2$, ..., and engage extra tasks to check that, say, $ESCL_2$ correctly implements $ESCL_1$, and so on. These extra stepts are omitted here for simplicity.

1. 1.Formal functional requirements specification (FRS)
2. 2.Formal FRS analysis

1 There are many methods which produce good results in research projects, but which do not translate so easily into everyday engineering production. Such methods do not appear in this list.

2 The list was initially formulated by the author, and elaborated in discussion with the Finsbury Group: Peter Bishop, Robin Bloomfield, the late John Knight, Peter Ladkin, Bev Littlewood, Lorenzo Strigini, Martyn Thomas. John Rushby joined us in electronic discussion and observed that runtime verification methods were industrially mature.

3. 3.Formal safety requirements specification (FSRS)
4. 4.Formal FSRS analysis
5. 5.Automated proving/proof checking of properties (consistency, completeness of certain types) of FRS and FSRS
6. 6.Formal modelling, model checking, and model exploration of FRS, FSRS
7. 7.Formal design specification (FDS)
8. 8.Formal analysis of FDS
9. 9.Automated proving/proof checking of fulfilment of the FRS/ FSRS by FDS
10. 10.Formal modelling, model checking, and model exploration of FDS
11. 11.Formal determininistic static analysis of FDS (information flow, data flow, possibilities of run-time error)
12. 12.Codevelopment of FDS with ESCL
13. 13.Automated source-code generation from FDS or intermediate specification (IS)
14. 14.Automated proving/proof checking of fulfilment of FDS by IS
15. 15.Automated verification-condition generation from/with ESCL
16. 16.Rigorous semantics of ESCL
17. 17.Automated ESCL-level proving /proof checking of properties (such as freedom from susceptibility to certain kinds of run-time error)
18. 18.Automated proving/proof checking of fulfilment of FDS by ESCL
19. 19.Formal test generation from FRS
20. 20.Formal test generation from FSRS
21. 21.Formal test generation from FDS
22. 22.Formal test generation from IS
23. 23.Formal test generation from ESCL
24. 24.Formal coding-standards analysis (SPARK, MISRA C, etc)
25. 25.Worst-Case Execution Time (WCET) analysis
26. 26.Monitor synthesis/runtime verification

Please suppose for a moment that we got this right and this is a more or less

comprehensive categorisation of state-of-the-practice use of formal methods. The question to be asked, with each of these tasks or objects is: did you do it / have you got it? The answers will give a taste to the software assessor of what might be expected from the software. The answers will equally give a sense to the developer of whether the assessor is likely to start out a more-happy or a less-happy camper!

10.3 Software Assurance

Being precise about what formal methods are out there is fairly low-hanging. But merely checking a list of things you've done does not of course suffice to assure quality of the software product. To assure the product, you will have needed to have done those things adequately and well, and you will have needed to have done some specific things; not everything in the list, but some of them. And to satisfy an assessor – indeed to satisfy yourself - you will need to have some proof that these things have been accomplished.

A simplified view of assessing a software product for its fitness for purpose is as follows.

1. 1.You want to know what the software is supposed to do and not to do.
2. 2.You want to know that the object code sitting on the processor does this.
3. 3.You want to know that the operation of the software is free from dangerous quirks.

I take it we know well that none of these steps is routine (the standard, of course, has some teens of steps where here there are just three. For discussion, see [3]).

Software is usually written in a number of steps, with each step having its own language. The first step, usually called requirements, says what the software is to achieve. The last step, usually object code, specifies precisely what actions the processor(s) is(are) to take. We are concerned with safety, that is, absence of dangerous behavior, so the requirements with which we are concerned are the safety requirements, which specify the absence of dangerous behavior. That is step 1. And this absence is what a safety engineer wishes to assure throughout the software development, to object code (and the electrical engineer beyond that). That is step 2. Step 3 is a general catch-all for those phenomena which might not have been covered in Steps 1 and 2: a general «did we do everything right?» step.

There is not usually any direct way of comparing object code with requirements. Software production is broken down into a number of smaller steps, and then one can compare the results of each step with those of its predecessor and successor steps. Most products are in language of some sort, usually formal: design specification, source code, object code. And comparisons often take the form of translation: a compiler, for example, translates source code into object code, and a code-generator translates finite-state-machines or other descriptions into, say, C source code. One wants to know that these translations are «faithful», do the job the engineers are expecting of them. However, some steps are not translations. Obtaining the safety requirements in the first place is not usually an act of translation. And neither is checking executable specifications or code against requirements, although translation will likely be involved.

So, how might step 1 look when formulated in a standard?

10.3.1 Proposal A

. An unambiguous, rigorous Functional Requirements Specification (FRS) be required. The FRS needs to be checkable for (i) consistency, and (ii) relative completeness. It be required that it is so checked and the methods and results to appear in the safety case.

Unambiguity and rigor are very important, as follows. Suppose your requirement is the door shall not be shut after the horse has bolted. What if there are two doors? Do you mean neither door, or just this one, or just that one? That is the condition of unambiguity. Where there are two or more readings, you have to say which is meant. But what is supposed to happen if the horse is in the course of bolting? Is the predicate has bolted true as soon as he has the idea in his head and has started to act on it, or is it true only when he is at full tilt disappearing over the horizon? These boundary issues are seen only through consideration of what the requirement means when it is put up against the world, as it were. Things need to be decided more precisely than they were originally formulated. That is the condition of rigor. One can see that these conditions apply to systems in general, not just to software.

What about consistency and relative completeness? Consistency first. Requirements often derive from two or more differently interested parties, and you need to know that these parties are not formulating requirements that preclude each other, for, if

they do, any system whatever which you build is going to fall short of one set or another, and if they are to do with safety then falling short is a Bad Thing. This is not necessarily a rare occurrence, as those experienced with requirements specification for even moderately complex systems will confirm. It's necessary and wise to check, hence the requirement to do so.

Relative completeness is more controversial. Here, people can – and do - argue until the cows come home as to what «completeness» is, and some people will even say there cannot be any [Safecrit10]! In fact, certain notions of completeness can be usefully applied as a quality check on the requirements specification [L10.1,L10.2]. For example, that you have written down requirements which preclude all dangerous behavior which can be expressed in the language in which you have written the requirements. So, do you shut the door before the horse has bolted? Maybe you do, so the horse won't be tempted; or maybe you don't, so he won't charge through the door and hurt himself. Relative completeness says you must consider the question and decide. What you cannot express at this point is that the lubrication on the hinges releases poisonous elements into the atmosphere which will make you ill if you open and shut the door continuously. Such a phenomenon isn't yet in the language and is therefore best left to discovery later.

It is both useful and wise to check whether you have said everything safety-wise that can be said at this point and that is why the relative completeness condition is there.

Finally, this should all be written down and presented to the assessor, indeed some would argue as I would that it be as far as possible a public document, because users or neighbors of the system might well like to know what they are getting themselves into, and may wish to check for themselves.

So much for general safety requirements. Now to Step 2.

10.3.2 Proposal B

(i) The SW Architecture/Design Spec be rigorous.
(ii) There be a formal, rigorous, correct demonstration that the SWA/DS fulfils the FRS.

Most software has a specification of some sort as to how it will fulfil the requirements (not just the safety requirements, but all requirements). This proposal entails

that if the software is safety-critical, it must have one. If it doesn't have one in the usual sense, then one would be right to be suspicious. However, if this software exists then it can be deemed to be its own SWA/DS, at the source-code level (ESCL) or the object code. Good luck, in that case, with the rigor and the demonstration! But in any case an ESCL may be defined. So the implicit condition that there be an SWA/DS is no extra burden.

The SWA/DS is the first place at which you say how your software is going to look and what it is going to be doing. Again, rigor is important. Here is the first place at which it can sensibly be asked whether the safety requirements are fulfilled. So it is asked, and B(ii) provides the answer.

10.3.3 Proposal C

(i) If the SW is written using a higher-level programming language than machine code, there be defined a language to be called the Executable Source Code Level.
(ii) The SW at the ESCL be rigorously, correctly demonstrated to fulfil the SWA/DS.

Most critical software is written using a programming language at somewhat higher level than machine code. That software in that higher-level language forms an intermediate translation step between SWA/DS and object code. You have to say what your source-code level (ESCL) is. There might be many stages at which you could define an ESCL. If you program in Java, do you take your ECSL to be the Java source, or the Bytecode? Your choice, but you have to say. If you write finite-state-machine descriptions, and then press a button to get C-code implementing your state machines, then is the state-machine-description your ESCL, or is it the resulting C-code? Again, your choice but you have to say. Say both, if you like! Then you have ESCL1 and ESCL2 and a formal translation between them. So you can check ESCL1 against SWA/DS and ESCL2 against object code, and then argue that the tool triggered by your button-press produces a «correct» translation from ESCL1 to ESCL2, whatever «correct» means. The obligation proposed is to show your system at ESCL does what you said it should do at SWA/DS.

An example of an industrially-proven approach to Proposals B and C is [B06].

10.3.4 Proposal D

Compilation is defined to be an operation that translates ESCL into object code (OC) , where OC means the executable bytes that sit on the hardware. There be a rigorous, correct demonstration that the OC fulfils the ESCL.

So, the final step from ESCL to object code.

Note that there are here just two steps from SWA/DS to object code, going through ESCL. There might be many. As I just indicated, maybe one has de facto many ESCLs. Say one has three, ESCL1, ESCL2 and ESCL3. Then just one of these will be chosen as the ESCL, say, ESCL2. The demonstration that ESCL2 satisfies ESCL1 will then be part of the demonstration in C(ii) that ESCL2 satisfies SWA/DS, and the demonstration that ESCL3 satisfies ESCL2 will become part of the demonstration in D that the object code has the right relationship to ESCL2. So the condition in C and D is fulfilled also when there are many steps between SWA/DS and object code.

Object code isn't the end of the story. There are things which go wrong when running programs on processors that are not necessarily embodied in the meanings of the programming languages, including object code. Such as attempting a divide operation when the denominator is zero, or trying to put a number in a register that is bigger than will fit. These are collectively known as run-time errors. Run-time errors are generally bad; the processor stops, or continues with bad data. If your software is running a safety monitoring program, then stopping that program is a Bad Thing. If your software is calculating and monitoring safe levels of some possibly dangerous phenomenon, then injecting bad data is a Bad Thing. Generally, run-time errors are Bad Things in critical systems. The state of the practice is that there are mature techniques which will preclude run-time errors. They should be used, and this should be documented.

10.3.5 Proposal E

There be a rigorous, correct demonstration that run-time errors do not cause or contribute to causing dangerous failures.

One may do so by eliminating whole classes of run-time errors, as with SPARK, or by trapping and handling the raised exceptions. And so on. Avoiding run-time errors is state-of-the-practice, so it should be standard.

10.3.6 Proposal Cluster F: Testing

Here, many remain somewhat befuddled. Well-known mathematical results say that appropriate software assurance is just not to be had through practical statistical testing [BF93,LS93]. But one does get some kind of assurance of SW fitness for purpose through even routine testing. The hard part – the problem – here is to make precise what kind of assurance this is, and specify how it is achieved. To cut to the chase, I have here only partial improvements to offer, and not yet a comprehensive proposal.

Given partial improvements and lack of an encompassing approach, can we best proceed by ignoring this issue for the standard? No, there has to be something here to reflect the necessity of good testing, in whatever that necessity consists.

I have a personal story and a moral, like most people who have written software. When I was writing mostly declarative code in the language REFINE, I wrote a time-interval calculation system over a few months. I did unit testing of the functions as I was writing them, performing (a) sanity checks, and, more thoroughly, (b) boundary-case calculations. Integration of the entire system took a programmer, who had no idea what I had written, two hours, mostly performing (a) and (b) at the integration level. She found one boundary case I had missed, I fixed it, and the code was on demo at a major conference the next day, in 1986. It has been used, I don't know how much, by the client in a system which for many years had an annual conference devoted to it, and I have not heard of any further errors. My part was not big, but it would have been far, far more consumptive of time and effort to develop it in C. My unit testing was goal-directed, semantics-directed, and effective. Semantics-directed testing for (a) and (b) seem to me intuitively to be needed for any system. One could argue that they would be supplanted by effective SPARK-like formal methods; maybe so. But they or an equivalent are somehow needed; routine testing is essential for ultra-highly dependable systems. That is my view. But where are these methods? How can we recommend them to all in a standard if they don't exist, or only appear through happenstance, or with the use of particular toolsets, and so on? We can't.

There needs to be some kind of story on what unit testing and integration testing achieve and what is to be shown concerning that achievement. There are some ways, see below, in which certain kinds of testing produce proof of concrete properties of the system. Such methods need to be in the standard.

10.4 Summary

I have proposed how we may begin to assess critical software based on documented properties of the software, rather than through the process by which it was developed. I have also proposed that a requirement to do so be in the 61508 standard and have indicated some approaches.

I have addressed the vexing question of what one can know about software through running it, either in test situations or as conclusions from previous use. I have suggested that we are not very far along the road to practical methods of assessment, but that there are pioneering techniques which hold promise. Progress in this area might allow us to set and assess acceptable rates of dangerous failure for software and other designs, which is currently a major unanswered question in the risk-based assessment of critical systems.

Bibliography

[1] J. Barnes with Praxis, *High Integrity Software*, Addison-Wesley 2006.

[2] International Electrotechnical Commission, IEC 61508, *Functional safety of electrical/electronic/programmable electronic safety-related systems*, 7 parts, 2010.

[3] International Electrotechnical Commission, IEC 61078 Reliability Block Diagrams, 2016.

[4] Peter Bernard Ladkin, *An Overview of IEC 61508 on E/E/PE Functional Safety*. Available at http://www.causalis.com/IEC61508FunctionalSafety.pdf , Causalis Limited, 2008.

[5] Bev Littlewood and Lorenzo Strigini, *Validation of Ultra-High Dependability for Software-based Systems*, Communications of the ACM, vol. 36(11), pp. 69-80, November 1993.