Designing for failure

Pieter G. Beerthuizen
Dutch Space B.V.
Leiden, The Netherlands
Phone: +31-71-5245712
Fax: +31-71-5245388
E-mail: p.beerthuizen@dutchspace.nl

Introduction

The title of this paper is of course meant as a teaser, and both the title and the paper were partly inspired by the 1991 Turing Award Lecture by Fernando J. Corbató. It refers to two aspects of today's (embedded) aerospace system design: systems have become so complex that (partial) failure becomes almost inevitable; and conversely one must find ways to cope with these failures such that function and performance are not significantly compromised, especially when dependability is an issue. Software tends to dramatically increase complexity: anything seems possible to implement, especially when processor performance and memory capacity hardly pose real constraints to the creative minds of a system's specifier.

A vast amount of approaches and methods have been defined and taken into widespread use in the last decades to make better and more reliable software designs (whatever that may be), like modular and modifiable architectures, extensive testing in various ways, analyse systems for nominal and failure behaviour, software process improvement, extensive standardisation, and so on. However, when studying the literature and watching every day's newspaper columns, it appears that failures of complex systems are manifold, are seemingly unavoidable, and are sometimes extremely expensive.
The question now is: "Is there any way at all to avoid such occurrences?"

It will be obvious, that this paper does not present the silver bullet; nor do all those approaches and methods. Applying them (if done correctly) usually does no harm, and may even result in very good working systems. However, it is not a guarantee for being failure free, and cost and needed time tend to increase exponentially for most of them.
A number of aspects, especially related to dependable systems, will be discussed.

Complexity issues

Some twenty-five years ago, aircraft industry and certification authorities were facing the challenge of new digital avionics systems for commercial (FAR/JAR 25 certified) aircraft. Processors for embedded systems were commercially available, and compilers and assemblers had been developed and in widespread commercial use. However, although their designs were still quite straight forward and there was no indication of unreliability, the certification rules required a much more rigorous approach.
Avionics industry started to develop their proprietary (micro-programmed bitslice) processor systems, and invented computer languages and compilers that could be certified, since only then it would be possible to control all variables (commercial vendors would not guarantee stability and support of one version over time). After a few years (and an immense number of man years) this was not considered a viable approach: 100% fault free testing could not realistically be achieved, and the pace of technology overtook the activities to establish the "standard" processing system.
Eventually, initiated from the military, some (much less ambitious) solutions came in the Mil-

1750 processor and the Ada programming language as building blocks. The basis for the initiative however was different from the civil avionics one.

In the same time frame, the IRAS satellite was rescued by reprogramming the computer's memory to repair a glitch problem, which compromised the attitude control. For a long time, this has been one of the history marks leading to making satellite platform computers reprogrammable, as to enable repairs when in orbit. Born out of necessity, since a satellite can no longer be reached once up in space, realising that a space computer may contain errors here led to design decisions.
These days reprogrammability is a standard feature, but no longer for this reason only: the end user wishes to adapt the mission, hence the satellite's behaviour, with the satellite already in orbit, thereby increasing flexibility in use.

Where processors such as the early 8-bit and 16-bit ones were still using a straight forward design, making it (theoretically) possible to analyse in detail and single-step through a programme running on the microprocessor, for today's processors and related operating systems and high level languages use this is out of the question. Pipelines, cache usage on various levels, parallelism, optimisation, etc., make so many hidden paths that the old bottom-up analysis and test approach poses significant problems. One line change in a programme may cause a completely different processing sequence, which may make the difference between a system failure or not.

These are just a few examples of the two-sided interaction between technological potential and customer use.
Microprocessor technology allowed more complex functionality, but it was exactly this complexity, which caused significant problems for the traditional bottom-up validation of avionics systems in the late 70's and early 80's. More recently, something similar holds for the application of the new processor and software technologies versus the testing approaches that were common for more than a decade.
The customer always wanting more (and in aerospace usually being quite well aware of the technical boundaries) causes existing technologies to be the basis for new creativity in requirements on new systems. This may easily become so important that the original reason for a design feature (such as reprogrammability) is compromised.

Some common approaches

As mentioned above, many different approaches with differing original purpose are used to cope with complexity and dependability. Standardisation of development, in phases, iterations, documentation, traceability, testing, … has been part of all aerospace projects, defined in e.g. the ESA PSS and in RTCA standards for space and aircraft industry respectively. Above all, these standards provide *management* tools aiming at controlling the processes and methods, with the assumption that, when properly applied, this will lead to better (i.e. less defects, more maintainable, cheaper, …) systems.
Although it is without doubt that a *systematic* approach, as required through the standards, avoids many problems, and also is a prerequisite for managing complex developments, it is not enough to prevent (fatal) failures to occur. These are most commonly due to occurrences of multiple coinciding error situations and/or the result of a conceptual error in design, verification, etc. Many fatal failures in systems appear to originate in strikingly simple matters, if one would only have thought of it.
So why don't we think of them before?

A common way of dealing with dependability is by applying redundancy to systems. Normal civil aircraft have dual or triple systems as the most flight critical ones are concerned. The space shuttle's flight control system goes quite beyond this amount. In satellites there are severe restrictions in mass and power, so there is more effort put in smart solutions such as

skewed reaction wheels to limit the amount of redundancy. A current trend is to combine functions from attitude control and data handling into one (redundant) single processor system, which traditionally were divided up into two separate systems monitoring each other.

Redundancy aims at reducing the probability of failure of the total system, under the assumption that the redundant systems will fail totally independent of each other, i.e. there are no common cause failures possible. This is mostly not true, and it has been the basis for requiring “dissimilar programming” for the most critical system functions. In the extreme, it also requires that dissimilar algorisms shall be used, and even dissimilar hardware may be required. Of course, the use of runtime kernels or operating systems is largely compromised by such requirements. Real dissimilar development is very difficult to achieve and extremely costly.

Testing has long been the ultimate proof of correctness of many systems, despite Dijkstra’s statement long ago, that it can only show the occurrence of an error but never the absence of errors. Testing hardware systems like aspects of mechanics, chemical effects, electrical disturbance is of different nature than testing software. Though hardware-software interaction is a possible error source, most software errors are essentially designed-in. Therefore testing software-based systems involves finding those specific instances by all kinds of analysis and logic (path) testing on various levels of integration. Testing a rather small software programme of say 30kLOC for 100% decision coverage (which is still not covering all possible paths through the software) easily results in many tens of thousands of test cases for that purpose alone. However, introducing e.g. an algorithmic error or making a design error is much more likely, and the impact is much larger.

Failure analysis methods for dependable systems may be top-down or bottom-up.
Traditionally (and still required in many occasions) bottom-up analyses like FMECA’s shall be performed. In today’s software based systems this is a sheer impossibility. Experiences already in the early 80’s showed that for small (assembly level) programmes on early microprocessors could be done, but one should have full insight in every instruction being executed. Multiple errors are of course not analysed. As already mentioned above, the “disturbing” effects of pipelines, caches and the like in modern processors make failure analysis on the lowest levels practically useless.
Top-down analysis such as FTA (supported by higher level FMECA) is a common method to design fault tolerance into a system in a very early stage. It also allows the design of very effective high-level barriers against (categories of) failures and limits the amount of hardware and software and their respective verification and validation. The method however is easily flawed when some aspects are overlooked, such as human error (commanding errors, …) or unanticipated subsystem behaviour (sensor errors, …).

System engineering aspects

This chapter will deal with the elaboration of some system engineering concepts to achieve a high level of dependability. It will focus on a number of aspects such as architecture choices, and constraints put by those choices.

The basic starting point for coping with dependability requirements here is the assumption that errors and failures will occur, and the focus (apart from trying to avoid them being a priori introduced into the system) should be on smoothly dealing with the occurrence such that the system functions and performance are minimally compromised. I.e. design for failure.

Discussion

This chapter will discuss the pro’s and con’s, blessings and pitfalls of the system engineering concepts discussed in the previous chapter, in relation to other approaches.
Also aspects like requirements and tests being sensible or senseless (in specific cases) will be addressed.

Conclusion

Complexity of today’s software based aerospace systems is increasing dramatically. Especially when dependability is at stake, many traditional approaches are no longer sufficient. While the pace of technological developments is very high, and the creativity of the users and customers in specifying new application areas for these new possibilities is virtually unlimited, there is no silver bullet to this challenge.
Sound system engineering approaches, paired to careful analysis together with the users and customer on the real needs, are conditions to possible solutions for this challenge.