Reliable Software Systems Design:
Defect Prevention, Detection, and Containment

Gerard J. Holzmann
Rajeev Joshi

Laboratory for Reliable Software
NASA/JPL Pasadena, CA 91109, USA

Introduction

The grand challenge that is the focus of this conference targets the development of a practical methodology for software verification: a practical verification tool that would work like a language compiler does today. The objective of software verification is of course to reduce the number of design and coding defects in software products, and ultimately to reduce the number of failures in the use of a product. It is safe to assume that virtually all non-trivial software in use today has defects. Some of these defects are merely cosmetic in nature, but some can also cause real damage: damage that can be measured in terms of time and money lost, and in some cases in terms of lives lost. The greater the damage that can be caused by a software defect, the greater our desire is to prevent it.

It has often been argued that with the right training, discipline, and tools it should be possible to produce zero-defect code. Very few things in life, though, are zero-defect – not even the things that can be considered life critical. Traffic lights and elevators can fail, ambulances and fire engines can fail, even the phone system and your hard-disk drive can fail. If an elevator company promised to have developed a zero-defect elevator you would have every reason to be suspicious. The reason that we trust, for example, elevators is that they are designed to explicitly take the possibility of *component* failure into account to prevent *system* failure.

Building Reliable Systems from Unreliable Parts

Hardware designers know how to construct *reliable systems* from *unreliable parts*. In building these systems, the designer starts from the knowledge that any component in the system might fail, while securing that such failures can not cause the failure of the system as a whole. When an elevator fails, the car does not come crashing down, because the system was designed to handle this type of defect. We have yet to learn how to apply similar principles in the construction of reliable *software* systems.

Although there is a strong need to improve software verification techniques, the purpose of this position paper is to point out that our ultimate objective is not necessarily to produce zero-defect software, but to produce ultra-reliable software systems. This position has implications for the type of work we plan to do, as we will outline in more detail in the remainder of this paper.

Blue Screens of Death

Non-critical software applications are often designed in a monolithic fashion. When the application crashes, e.g. when it hits a divide by zero error, the only recourse one then has is to restart the application. This approach is of course not adequate to use in the construction of systems that must be ultra-reliable, for instance because human life depends on its correct and continued functioning. When, for instance, a spacecraft experiences the failure of one of its components during a launch or landing procedure, a complete restart of the software may in itself cost the loss of the mission. In manned space flight, a few minutes spent in rebooting the crew's life support system may have similarly unintended consequences. Systems like this have to be ultra-reliable, even if some of their software parts are not.

Simplicity and Redundancy

There are two primary strategies for achieving system reliability. The first strategy is to use a design that emphasizes *simplicity* and robustness. A simple design is easier to understand, easier to test or verify, and easier to operate. The second strategy is to exploit *redundancy*. If the probability of failure of individual components is statistically independent, the chance of having both a prime and a backup component fail at the same time can be made very small. If, for instance, all components have the same probability p of failure, then the probability that all N components fail at the same time in an N-redundant system would be p^N. In a nutshell, simplicity seeks to reduce the value of p, while redundancy seeks to increase the value of N. Trivially, for all values of $N \geq 1$ and $0 < p < 1$ both of these techniques can lower the probability of failure p^N for the system.

Unfortunately, one of the basic premises used in the redundancy argument that we used above, the statistical independence of the failure probabilities of individual components, can be very hard to achieve for software components. Well-known are the experiments performed in the eighties by Knight and Leveson with N-version programming techniques, which demonstrated that different programming teams tend to make the same types of design errors when working from a common set of (often flawed) design requirements. [KL86] Independently, Sha also pointed out that a decision to apply N-version programming cannot be made independently of budget and schedule decisions. With a fixed budget, each of N independent development efforts will inevitably receive only $1/N$-th of the total project resources. If we compare the expected reliability of N development efforts, each pursued with $1/N$-th of the project resources, with one targeted effort that can consume all available resources, the tradeoffs become very different. [S01]

Redundancy in the traditional sense, in the way that has proven to work well with hardware systems, therefore cannot be duplicated easily in software systems. By combining the strategies of simplicity and redundancy in a slightly different way, though, we may be able to build larger software systems that are indeed significantly more reliable than any of their individual parts.

Software Architectures for Fault Containment

Consider a standard architecture consisting of software modules with well-defined interfaces. Each module performs a separate function. The modules are chosen to minimize information flow across module boundaries. We will assume here, primarily for simplicity but without loss of generality, that the only way for modules to interact is through message passing over trusted channels. Modules execute (at least logically) on independent hardware, to secure that the crash of one module cannot affect other modules in any other way than across its module interface. A failed module may stop responding, or fail to comply with the interface protocols by sending erroneous requests or responses. We will make a further convenient assumption that module failures can be detected either through consistency checks that are performed inside a module, or by peer modules that check the validity of messages that cross module boundaries.

> One could make the argument that a failure that cannot be detected at runtime it is not a failure that can be remedied. We will have to accept that not all conceivable types of failures can be defended against with this or any other fault containment discipline. We restrict our attention to those cases where a remedy is at least in principle possible.

In our proposed software architecture each software module is provided with a backup. In normal operations, this backup module is idle. When a fault is detected, the faulty module is switched offline and the backup module replaces it. (Naturally, the backup module can have its own backup, and so on, but we will not pursue this generalization here.)

> Note that in a traditional system the failing module is its own backup. Upon a failure one simply restarts the module that failed and hopes that the cause for failure was transient. We suggest that we can defend against a substantially larger class of defects if the backup module is *distinct* from the primary module and deliberately constructed to be significantly *simpler* than the primary module.

As indicated earlier, if the primary and backup modules are constructed within an *N*-version programming paradigm, we do not necessarily gain additional reliability from this type of system structure. This system structure will not adequately defend against design and coding errors. Some of the same design errors may be made in the construction of both modules, and if the two modules are of similar size and complexity, they should be expected to contain a similar number of residual coding defects (i.e., coding defects that escape code testing and verification). Our proposal is to make the backup modules significantly simpler than the primary modules.

Simplified Redundancy

The backup modules in our proposed architecture are constructed as *simplified* versions of the primary modules. Specifically, these backup modules can be designed and build by the same developer(s) that design and build the primary modules. The primary module is build for *performance*; the backup module is build for *correctness*. The main purpose for a system architecture of this type is that the backup modules are easier to verify thoroughly. The *statistically expected* number of residual defects in a backup module should be lower than that of the primary module, because they contain less code.

The basic premise is that the backup module guarantees continuity of operation, though in a somewhat degraded state of operation (e.g., slower and likely with reduced functionality). The backup gives the system the opportunity to recover from unexpected failures: the primary module is offline and can be diagnosed and possibly restarted, while the backup module takes care of the most urgent of tasks in the most basic of ways. If code is developed in a hierarchical fashion, using a standardized software refinement approach, the backup module could encapsulate an earlier level in the refinement of the final module: a simpler version of the code that is not yet burdened with all features, extensions, and optimizations that will support the final version, but that does perform the most critical and basic duties in the most straightforward way.

If this approach can be made to work (we have yet to do a realistic case study) we would expect the backup modules to be significantly smaller in size (e.g., in lines of code) than the primary modules. By virtue of being smaller and simpler, the expected number of residual defects in this code should also be smaller. We will tacitly assume here that the number of design and coding defects is proportional to the size of a module, just like the number of syntax and grammar mistakes in English prose is proportional to the length of that prose. If now the primary module has a probability of failure due to residual defects of p and for the backup module the probability of failure is q, we would expect to have $1 > p > q > 0$ (ignoring the boundary cases where we have either certainty of failure or absolute perfection). Because the backup module contains less code, and implements less functionality, it offers fewer opportunities for design and coding defects. The module with its backup now fails with probability ($p.q$) which should be smaller than the probability p for the same module without the backup.

Fault Detection and Secure Fall-Back

We have assumed that we can tell, in a sufficiently broad number of cases, when a software module fails to perform its intended function due to a design or coding error. There are several ways in which this could work, at least in principle, but none are truly satisfactory. The module code can contain assertions that check for the validity of inputs and outputs (standard pre and post-condition checks), and verify that essential invariants are maintained in the module code. But if we assume that the nature of the residual software defects is unpredictable and to first approximation will exhibit itself as a random divergence of the intended or desired code, the conclusion will be inevitable that a module cannot reliably detect all occurrences of defects in its own code. Modules can, however, be reasonably expected to check each other. If a module, for instance, detects that faulty input is provided to it across its module interface, the module could declare the peer module that provided the input to be faulty, reject the input, and command the suspect module to switch-over to its backup. There is a close correspondence here to security related problems in mainstream software design: how can a module trust that its peer is reliable? [R98, W89]

There is also another problem that has to be addressed. Even supposing that we would have, or will be able to develop, a reliable defect detection discipline, how precisely can we arrange things in such a way that the switch-over from a primary module to its backup (or vice versa) does not itself introduce a system failure? cf. [AB85, RL81] We do not have answers to these questions, but suggest them as a potentially fruitful area of research in reliable software systems design.

Synopsis

We suggest that to achieve software reliability we should not only be investigating ways to achieve zero-defect code, but also more broadly ways to produce fail-proof systems, that is the art of building reliable software systems from unreliable software components. The principal method of structuring code we propose to investigate is fairly simple. The code is structured into modules that can fail largely independently. Modules communicate only via well-defined interfaces. Each module is provided with at least one backup that can take over basic operations when the primary module fails. The backup module is constructed to be significantly simpler, smaller, and more reliable than the primary that it supports, possibly performing less efficiently and providing less functionality.

This basic mode of operation is used today in the hardware design of spacecraft. Spacecraft typically do not just have redundant components on board, but also components of different types, providing different grades of service. Most spacecraft, for instance, have both a high-gain and a low-gain antenna. When the high-gain antenna becomes unusable, the more reliable low-gain antenna is used, be it at a significantly reduced bit-rate. Perhaps not surprisingly, this same principle has also been applied on a modest scale in the design of mission critical software, though not always systematically. The MER rover software, for instance, was designed to support two main modes of operations: the fully functional mode with all its features and functions enabled and a minimal basic mode of operation that has been referred to as the "crippled mode." It was precisely this "crippled mode" that made it possible for the software engineers to recover from a serious software anomaly that struck one of the rovers early in its mission. [RN05] Our proposal is to use these principles more systematically, throughout the software design and all safety or mission critical components.

Acknowledgement

The work described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Specifically, this work is part of a NASA funded four year project SISM-160, titled *Reliable Software Systems Development*, which targets the development of new tool based methodologies for reliable software development.

References

[AB85] T. Anderson, P.A. Barrett, D.N. Halliwell, M.L. Moudling, "An evaluation of software fault tolerance in a practical system", Proc. Fault Tolerant Computing Symposium 1985, pp. 140-145.

[KL86] J.C. Knight and N.G. Leveson, "An Experimental Evaluation of the Assumption of Independence in Multi-version Programming," *IEEE Transactions on SoftwareEngineering*,Vol. SE-12, No. 1 (January 1986), pp. 96-109.

[RN05] G. Reeves and T. Neilson, "The Mars Rover Spirit Flash Anomaly," *IEEE Aerospace Conference*, Big Sky, MT, March 2005.

[RL81] R.D. Rasmussen, and E.C. Litty, "A Voyager attitude control perspective on fault tolerant systems," Proc. AIAA Conf., August 1981, Alburquerque, NM, pp. 241-248.

[R98] J. Rushby, "Partitioning in Avionics Architectures: Requirements, Mechanisms, and Assurance. Draft technical report, Computer Science Laboratory, SRI,1998.

[S01] L. Sha. "Using Simplicity to Control Complexity," *IEEE Software*, July-August 2001, pp. 20-28.

[W89] D.G. Weber, "Formal specification of fault-tolerance and its relation to computer security", Proc. 5th Int. Workshop on Software Spec. and Design, pp 273-277, Pittsburgh, PA, May 1989