Design for Validation

Sally C. Johnson
Ricky W. Butler

NASA Langley Research Center
Hampton, VA 23665–5225 *

Abstract

The use of computer hardware and software in life-critical applications, such as for civil air transports, demands the use of rigorous formal mathematical validation procedures. However, formal specification and verification will only be tractable if the system is designed in a manner that lends itself to formal methods. Likewise, accurate reliability analysis will only be tractable if the number of interacting components that must be individually included in a single reliability model is kept to a low number and if their failure behavior interactions can be modeled simply. Also, the system must be designed such that the system reliability does not directly depend on system parameters that cannot be accurately determined. This paper presents a design methodology based on the concept of designing a system in such a manner that it can be rigorously validated, or "design for validation."

Introduction

The development of the Airbus A320 marked the beginning of a new era in civil air transport technology—dependence on flight-crucial digital avionics. However, there are many indications that this step was premature, given the current state of the practice in digital systems design and validation[1]. Although the A320 was certified by the British Civil Aviation Authority (CAA), Brian Perry, head of Avionics and Electrical Systems for the CAA admits, "It's true that we are not able to establish to a fully verifiable level that the A320 software has no errors. It's not satisfactory, but it's a fact of life"[2]. Airframers perceive that increased use of flight-crucial digital avionics is an economic necessity. But how can traditionally conservative airframers, such as Boeing and McDonnell Douglas, safely make the transition to flight-crucial avionics without jeopardizing their conservative reputations?

There are numerous reports of serious incidents involving the use of computers in life-critical applications. For example, "In 1983 a United Airlines Boeing 767 went into a four-minute powerless glide after the pilot was compelled to shut down both engines," because the computerized engine-control system, in an attempt to optimize fuel efficiency, had ordered the engines to run at a relatively slow speed causing ice buildup and subsequent overheating[2]. John Garman, Deputy Chief of NASA Johnson's Spacecraft Software Division, stated, "It's as hard to predict a software failure as it is to predict what your poker hand will be in the next deal"[2].

The current procedure (RTCA DO-178A) used in certification of flight-crucial software for civil air transports is not so much a verification of the system itself as an examination of the process used in its development. The certification process consists of checking for completeness of documentation and adherence to acceptable design and development practices. According to Mike DeWalt of the FAA, "Basically, we take a slice through the whole system. That is, we pick a function like left aileron control and follow it all the way down through testing and configuration management"[2]. Thus, the testing of the system is clearly incomplete. Even after certification of the A320, "various unsettling reports have appeared in the European press, regarding: engines unexpectedly throttling up on final approach; inaccurate altimeter readings; sudden power loss prior to landing; steering problems while taxiing"[2].

There are two major reliability factors to be addressed in the design of ultra-reliable avionics: hardware component failures and design errors. Even though significant increases in the reliability of future hardware devices are envisioned, hardware component failures in the operational environment will remain unavoidable. Furthermore, industry trends towards significantly reducing the requirements for aircraft maintenance actions will mean increased dependence on the ability of systems to tolerate random hardware faults.

Design flaws are errors introduced in the development phase rather than the operational phase. These include errors in specification of the system, discrepancies between the specification and the design, and errors made in implementing the design in hardware or software.

While it is convenient to consider these factors separately, they are inexorably linked because of their strong interactions. The need for tolerating random hardware

*Published in IEEE Aerospace and Electronic Systems Journal, Vol. 7, No. 1, Jan. 1992, pp. 38-43. Also Presented at the 10th Digital Avionics Systems Conference (DASC), Los Angeles, Ca, Oct. 7-11, 1991.

component failures requires the use of redundant hardware components. The accompanying need for redundancy management functions can greatly increase the complexity of the operating system software and hardware. Complexity increases the likelihood of serious, yet latent, design flaws.

The design of a system entails making a series of design decisions and tradeoffs. These tradeoffs are typically made towards greater performance or lowest cost without regard for increased design complexity and thus lower reliability. For example, the developers of the Advanced Fighter Technology Integration (AFTI) F-16 decided to use triplex, asynchronous channels because it was believed that synchronous channels would be more vulnerable to a single-point failure due to electromagnetic interference (EMI) or lightning. However, this decision greatly complicated the design and integration of the system. During flight tests, the majority of the in-flight anomalies found were attributed to design oversights during integration of systems developed separately, and many of them were directly attributable to unexpected interactions between the asynchronous operation and the redundancy management system[3, 4].

This paper outlines an approach for the development of ultra-reliable digital avionics for civil air transports—a "design-for-validation" philosophy that includes rigorous application of formal methods. First, the basic concept of the methodology is introduced, and the role of formal methods is explored. The impact of the design-for-validation philosophy on the system design process is then demonstrated by two simple examples. More detail about the design-for-validation methodology is then given, followed by some concluding remarks.

Basic Concept

A commonly stated requirement for the flight critical components of commercial aircraft is a probability of failure not greater than 10^{-9} for a 10-hour mission time. This reliability region is clearly outside the domain where black-box testing is feasible. Thus, analytic techniques must be used in addition to testing to demonstrate that a system meets its requirements.

The validation problem for life-critical systems can be decomposed into two major subtasks:

1. Quantification of the probability of system failure due to physical failure.
2. Establishing that *design errors* are not present.

Since current technology cannot manufacture electronic devices with failure rates low enough to meet the reliability requirements directly, fault-tolerance strategies must be utilized that enable the continued operation of the system in the presence of component failures. The first subtask must therefore calculate the reliability of the system architecture that is designed to tolerate physical failures. This leads to the use of stochastic models of the fault arrival and fault recovery behaviors of the system. Such models depend critically upon the *correctness* of the software and hardware which *implements* the fault-tolerance of the system. For example, if the redundancy management system improperly diagnoses a good processor as failed or if a voter selects a faulty value, the assumptions of the reliability model may be violated—leading to "useless" reliability numbers. Thus, the second subtask must not only establish the absence of errors in the control laws and their implementation, but also the absence of errors in the underlying architecture which executes the control laws. Furthermore, it must be demonstrated that the reliability model is a complete and accurate model of the implemented system. Since this cannot be rigorously demonstrated through testing, analytic methods must be used. Thus, the *design-for-validation* concept consists of the following:

1. The system is designed in such a manner that a complete and accurate reliability model can be constructed. All parameters of the model that cannot be deduced from the logical design must be measured. All such parameters must be measurable within a feasible amount of time.
2. During the design process, tradeoffs are made in favor of designs that minimize the number of measurable parameters in order to reduce the validation cost. A design that has exceptional performance properties yet requires the measurement of hundreds of parameters (e.g., by time-consuming fault-injection experiments) would be rejected over a less capable system that requires minimal experimentation.
3. The system is designed in a manner that enables a proof of correctness of its logical structure. The reliability model does not include transitions representing design errors.
4. The reliability model is shown to be accurate with respect to the system implementation. This is accomplished analytically.

The Role of Formal Methods

The design-for-validation approach is based on the belief that life-critical digital systems (software and hardware) must be designed in a manner that enables rigorous mathematical analysis in order to truly meet their reliability goals. The mathematics for the design of a software system or digital hardware is *logic*, just as calculus and differential equations are the mathematical tools used in other engineering fields. The following steps are performed to accomplish a formal verification.

1. Specification of system using languages based on mathematical logic
2. Rigorous specification of desired properties as well as implementation details
3. Mathematical proof that the implementation meets the desired abstract properties
4. Use of semi-automatic theorem provers to insure the correctness of the proofs

The first two steps by themselves represent the most limited application of formal methods. Nevertheless, the use of specification languages based on mathematical logic can offer tremendous improvement in the specification process. Deficiencies and inconsistencies can be detected early in the development process when their correction is less costly. Step (3) represents the use of traditional mathematical "hand" proofs to verify that the implementation meets the specification. Step (4) represents the final and most rigorous application of formal methods—the use of mechanical theorem provers to check the correctness of the proofs themselves.

Several projects have already demonstrated that formal specification combined with informal design reviews and walkthroughs is useful and cost-effective for uncovering design faults. IBM's Cleanroom software experience has shown that "More than 90 percent of total product defects were found before first execution," (as opposed to the customary 60 percent), while productivity was "equal to or better than expected for ordinary software development"[5]. Likewise, the parallel development of the Transputer by two design teams concluded with the team employing formal specification techniques completing the design on time and under budget (and receiving the Queen's award in recognition of this effort). However, while the use of formal specification alone without proof is an effective method for uncovering design faults early in the design process, it is not rigorous enough for complex, life-critical applications. Numerous design faults were still uncovered during the testing of IBM software developed using formal specification teamed with informal correctness arguments.

Even when the correctness of a system is proven and checked using mechanical theorem provers, one cannot guarantee that the probability of a design fault is zero. The proofs could be based on incorrect axioms, the system requirements could be incomplete or inaccurate, or there could even be an error in the proof (e.g. the system designer makes an error in designing the system and comes up with an erroneous proof that happens to be declared a valid proof by the mechanical theorem prover because of a design fault in the theorem prover).

Formal methods is a powerful system design technique for two reasons. First, the use of formal methods provides a degree of confidence in the correctness of the system that is impossible with less rigorous methods. But more importantly, the application of formal methods forces the system designer to examine his system design in intricate detail and to keep that design simple and modular enough to be rigorously analyzable. For example, Dijkstra recognized that formal verification of software programs could be greatly simplified by restricting the programmer to a few basic control structures and eliminating the use of "goto" statements, and this was his principle motivation for introducing the idea of structured programming. Unfortunately, "Many popularizers of structured programming have cut out the rigorous part about mathematical verification in favor of the easy part about no gotos"[5].

System Design Examples

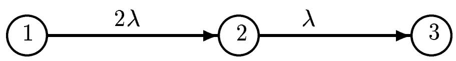

Figure 1: Over-simplified Model of Fault-Tolerant Dual Processor

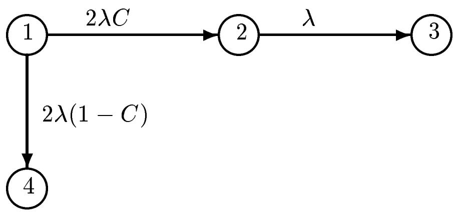

Figure 2: Accurate Model of Fault-Tolerant Dual Processor

The design-for-validation philosophy means that, ideally, how the system is to be validated should be considered from the very first moments of the system design process. The following simple examples illustrate this process:

Example System 1

Suppose we must design a simple fault-tolerant system with a probability of failure no greater than 2×10^{-6} whose maximum mission time is 10 hours. We quickly eliminate the use of a simplex processor since there is no technology that can produce a processor with this low of a failure rate. Thus, we begin to explore the notion of fault-tolerance. We next consider the use of redundancy—how about a dual? When the first processor fails, we will automatically switch to the other processor. We develop the Markov model shown in figure 1 to model such a system.

Unfortunately, our design suffers from one major problem. It would be impossible to *prove* that any implementation behaves in accordance with this model. The problem is that one cannot design a dual system that can detect the failure of the first processor and switch to the second 100% of the time.[1] Thus, we must accept the fact that there is a single-point failure in our system and include that failure transition in our reliability model (see figure 2).

Now we have a parameter in our model which must be measured—C. This parameter represents the fraction of single faults from which the system will successfully recover. We must now determine whether this parameter can be measured in a feasible amount of time (i.e. say less than year) with statistical significance. Analysis of this model using the SURE reliability analysis program[6] shows the sensitivity of the system reliability to C, as shown in Table 1. From this sensitivity analysis,

[1] There are theoretical proofs that this cannot be done.

C	LOWERBOUND	UPPERBOUND
.9990	2.99600×10^{-6}	2.99900×10^{-6}
.9992	2.59660×10^{-6}	2.59920×10^{-6}
.9994	2.19720×10^{-6}	2.19940×10^{-6}
.9996	1.79780×10^{-6}	1.79960×10^{-6}
.9998	1.39840×10^{-6}	1.39980×10^{-6}
1.000	9.99000×10^{-7}	1.00000×10^{-6}

Table 1: Sensitivity Analysis of System Reliability to Parameter C.

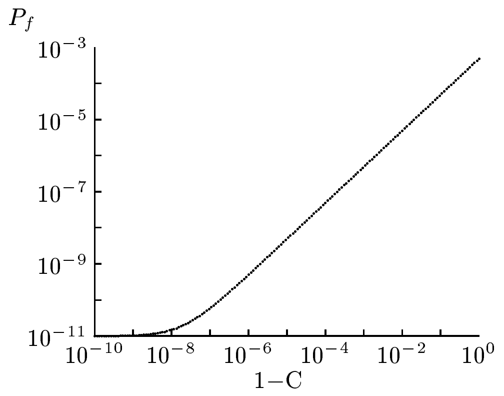

Figure 3: Failure Probability of 5MR with $\lambda = 10^{-5}$ as a Function of C

we can see that we must demonstrate that C is greater than 0.9995. It can be shown that 20000 observations are necessary to estimate this parameter to a reasonable level of statistical significance. If we optimistically [2] assume that each fault injection requires 1 minute, then this validation exercise would require 330 hours (i.e. 14 days). In this case, we decide we can live with this amount of testing and proceed to develop our system.

Example System 2

Now suppose we need to design a system with a reliability goal of $1 - 10^{-9}$. We decide to develop a nonreconfigurable 5-plex (5MR) using a processor with a failure rate of 10^{-5}/hour. We do not intend to use formal methods to verify the correctness of the fault-masking capability of the system, so we must rely on testing to validate this property. Through testing we must establish that the probability of a single point failure, say C, is sufficiently small. The probability of system failure is plotted as a function of 1−C in figure 3. The value of C must now be greater than 0.9999982.

It is easily shown that over a million fault injections would be required to measure this parameter even if we are very optimistic about the testing process. If each injection required 1 minute, this would require almost 1.9 years of non-stop fault injections.

It would be nice if we could design our system so that such an experiment is unnecessary. This is precisely the notion of design for validation. The system is designed so that a single point failure cannot cause system failure (i.e. C = 1), and this is demonstrated to be true by formal proof. Thus, one uses the power of analysis to eliminate experimental testing.

The Design-for-Validation Methodology

System design begins with a detailed description of the system requirements written in a formal, mathematical language. The system design then proceeds in a hierarchical fashion from a highest-level specification of the system down to a detailed implementation level. Therefore, formal methods are applied to the total system, not just to the individual subsystems, and all interactions between subsystems are formally described and understood. Of course, this represents the long-term ideal. In the short term, formal methods will probably be applied to individual critical subsystems first.

Although experimental methods cannot be used to measure ultra-reliability directly, there are important applications of experimental methods. The reliability models used to analyze the system will depend on accurate measurements of certain parameters, such as component failure rates and system reconfiguration rates. Likewise, the interface between the lowest level of formal system description and the actual hardware implementation of the system must be bridged by accurate descriptions and measurements of the hardware functionality and timing.

Reliability Analysis

Reliability models are constructed based on a detailed understanding of the failure modes and fault tolerance of a system. A reliability prediction is only as accurate as the reliability model of the system. Consequently, it is essential that a formal proof be constructed to demonstrate that the Markov model is an abstraction of the implementation[7]. It is important to recognize that experimental methods cannot be used to demonstrate this for ultrareliable systems. This would require as much experimentation as direct life-testing of the system.

Additionally, the reliability estimate obtained for a system is only as accurate as the parameters used in the model. Therefore, the reliability model, and hence the system behavior, must be based on parameters that can be accurately measured or estimated through analysis or experimentation. This would typically include the failure rates of the hardware components and the recovery time for detecting, isolating, and reconfiguring out a failed component.

There are practical and effective computational approaches available today for calculating the reliability

[2] Theoretically one would have to observe the system for a long time in case the fault has a large latency period. If one assumes that fault latency is less than 1 minute one can censor the experiment.

of Markov models[6, 8]. The main area of concern is that reliability models are often constructed with many parameters that would require exhorbitant amounts of testing to measure accurately. If rigorous validation is to be accomplished, systems will have to be designed differently—even if this means adding additional hardware to the system to make the validation task tractable.

Design Faults

Reliability modeling techniques are satisfactory for validating the failure probability due to random hardware failures given that accurate component failure rate data is available. The primary obstacle in validation of ultra-reliable systems concerns design faults in functionality, not random hardware failures. With random hardware failures, the failures are assumed (and generally accepted to be) independent between electrically isolated redundant channels, and the failure probabilities of the replicated units can be multipled, greatly increasing the overall system reliability prediction. When considering design faults such as software bugs, however, it has been found that errors in replicated versions, even though created by different programmers using different programming languages, are not independent; i.e. the programmers tend to make the same kinds of mistakes[9].

The concept of different replicated versions is called "design diversity" and has been applied to both software and hardware. It is generally accepted that design diversity can result in increased reliability, but it is not possible to quantify the increase in the ultra-reliable regime. These considerations leave validation of life-critical systems in a quandary: testing is not appropriate because of the exhorbitant number of tests required. The design-for-validation philosophy leads us to the approach of formally verifying the correctness of each and every element of the design. There is no attempt to measure the probability of system failure due to design faults. Once proven correct, the design is assumed correct for all analyses.

Although formal verification can conceptually be carried down to deeper and deeper levels of refinement (say to the quantum-physics level), ultimately one reaches a point where the cost/benefits do not justify verification at a level any lower. For example, it is typically believed that gate-level design is sufficiently low. At lower levels CAD synthesis tools seem to be adequate to develop fault-free designs. The implementation consequently is built in terms of "atomic" components such as NAND gates, crystal oscillators, latches, etc. These components are described mathematically. The demonstration that these components are described properly must be done experimentally. For example, the drift rates of the clock crystal oscillators is obtained by measurement.

Performance Analysis

Avionics systems typically consist of a number of tasks that execute periodically. The flight-crucial avionics tasks must reliably calculate the outputs needed to control the airplane according to strict real-time deadlines.

Figure 4: Histogram of Task Execution Times

Fundamentally, the validation must establish that all of the flight-crucial tasks meet their deadlines.

Although probabilistic/statistical methods have been successfully utilized to model general purpose operating systems, they have limited application to the performance validation of ultrareliable, hard real-time systems. In fact, the majority of performance analysis tools being developed today are useful for estimating the average performance levels of a system, but are of little use in estimating in the tails of the performance distribution. Simulation is of little value in such estimation for the same reason that software reliability cannot be quantified—you cannot estimate what you cannot observe.

Since the set of tasks are constant and their schedule is almost always static, the performance problem reduces to a demonstration that each task's execution time is bounded. Unfortunately, experimental methods cannot establish this property to the required level of reliability. When one measures the execution times of a task one obtains a histogram like the one shown in figure 4. Collecting enough measurements to estimate with sufficient statistical significance the probability that the hard deadline would be exceeded is infeasible. Consequently, one must use formal code analysis to demonstrate that the execution times are strictly bounded. However, in many cases such analytical methods will also be infeasible unless the code is developed (or redesigned) in such a manner as to support the required analyses. In recognition of this problem, the proposed 00-55 British defence standard defines strict coding practices that avoid implementations whose execution times cannot be analytically bounded.

System Modification

In an ideal world, the system requirements would be completely defined at the start of the project and frozen—changes in the system requirements during design and implementation would be forbidden. However, this is simply not a realistic scenario for large development projects. The plea that John Garman of NASA

Johnson directs to the academic and software engineering community is to "help us to find ways to reliably modify software with minimum impact in time and cost." Garman continues, "Maintaining software systems in the field, absorbing large changes or additions in the middle of development cycles, and reconfiguring software systems to "fit" never-quite-identical vehicles or missions are our real problems today"[10]. The reason that modifying systems is difficult and expensive is because the interactions between subsystems are subtle and hard to determine. When a change is made to one subsystem, it is extremely difficult to determine all of the other subsystems that are impacted by that change. However, if a system has been formally verified using an automated theorem prover system, then whenever a system modification is made, the user can determine which other subsystems are impacting by rerunning the proofs. The proofs for subsystems not impacted by the change will remain valid, while the proofs of correctness of the impacted subsystems will be reported as "unproved." The user then modifies the affected subsystems and their accompanying proofs, confident that no unexpected interactions have been overlooked.

Concluding Remarks

Most of the formal methods research sponsored in the United States has been targetted towards application of formal methods to security applications. We believe that application of formal methods will be the state of the practice for civil air transports in 10 to 15 years. To achieve this, much work must be done to develop formal methods technology. Methods and tools for developing formally verified fault-tolerant system hardware architectures, operating systems, and avionics application software must be developed and demonstrated. Current formal methods tools are tedious and difficult to use, and they can only be effectively used by persons skilled in formal mathematical reasoning. Over time, it is expected that tool developers will come through with creative breakthroughs to automate some of the tedious steps that are now required. However, the development of tools powerful enough to allow persons that are not skilled in mathematical logic to perform rigorous proofs of systems is very unlikely in the foreseeable future. The logical thought processes needed to prove a system correct are far beyond the capabilities of today's artificial intelligence research. Therefore, if formal methods are to gain widespread use, there must be a supply of logicians to practice the craft.

As formal methods becomes the state of the practice, reuse of proven hardware and software and reuse of proofs themselves will become cost effective. Software reuse today has gained only minimal acceptance for three reasons: 1) development of new software is perceived as being relatively cheap, 2) most software is not built with sufficient modularity to make its reuse practical, and 3) rigorous specification is crucial to reusability. Formally verified software is expensive, is typically built in a more modular fashion to facilitate the proof effort, and is rigorously specified. Therefore, one is more likely to try to reuse formally verified software. Even in situations where new software or hardware must be developed, existing proven designs can be modified and parts of their original proofs reused.

References

[1] N. G. Leveson, "Software safety: What, why, and how," *Computing Surveys*, vol. 18, June 1986.

[2] J. Beatson, "Is america ready to 'fly by wire'?," *Washington Post*, Apr. 1989.

[3] J. Rushby, "Formal specification and verification of a fault-masking and transient-recovery model for digital flight-control systems." To be published as a NASA Contractor Report, 1991.

[4] D. A. Mackall, "Experiences with a flight-crucial digital control system," Technical Paper 2857, NASA, Nov. 1988.

[5] H. D. Mills, M. Dyer, and R. C. Linger, "Cleanroom software engineering," *IEEE Software*, pp. 19–24, Sept. 1987.

[6] R. W. Butler and A. L. White, "SURE reliability analysis: Program and mathematics," Technical Paper 2764, NASA, Mar. 1988.

[7] L. Moser, M. Melliar-Smith, and R. Schwartz, "Design verification of SIFT," Contractor Report 4097, NASA, Sept. 1987.

[8] R. W. Butler and S. C. Johnson, "The art of fault-tolerant system reliability modeling," Technical Memorandum 102623, NASA, Mar. 1990.

[9] J. C. Knight and N. G. Leveson, "An experimental evaluation of the assumptions of independence in multiversion programming," *IEEE Transactions on Software Engineering*, vol. SE-12, pp. 96–109, Jan. 1986.

[10] J. R. Garman, "The bug heard 'round the world," *ACM SIGSOFT Software Engineering Notes*, vol. 6, pp. 3–10, Oct. 1981.