Hierarchical Approach to Specification and Verification of Fault-tolerant Operating Systems*

James L. Caldwell II & Ricky W. Butler
NASA Langley Research Center
Hampton, VA. 23665-5225

Benedetto L. DiVito
Vigyan, Inc.
Hampton VA. 23666

9 June 1990

Abstract

The goal of formal methods research in the Systems Validation Methods Branch (SVMB) at NASA Langley Research Center (LaRC) is the development of design and verification methodologies to support the development of provably correct system designs for life-critical control applications. Specifically, our efforts are directed at formal specification and verification of the most critical hardware and software components of fault-tolerant fly-by-wire control systems. These systems typically have reliability requirements mandating probability of failure $< 10^{-9}$ for 10-hour mission times. To achieve these ultra-reliability requirements implies provably correct fault-tolerant designs based on replicated hardware and software resources.

1 Introduction

The application of theorem provers to verification of critical properties of real-time fault-tolerant digital systems is being explored at NASA Langley. Specifically, we are interested in fly-by-wire digital avionics systems. Typically these systems continuously read sensor values, perform computations

*Presented at the *Workshop on Software Tools for Distributed Intelligent Control*, organized by Lawrence Livermore National Laboratory for the Defense Advanced Research Projects Agency (DARPA), Pacifica California, July 17-19 1990.

implementing the desired control laws, and output the resulting values to actuators. Sensor values might include airspeed or input on the attitude of the aircraft. The actuators control engines, flaps, and/or rudders.

The reliability requirements for commercial aircraft are very high — probability of failure less than 10^{-9} over 10-hour mission times. This level of reliability is often referred to as *ultra-reliability*. If quantification of system reliability to this level seems a questionable endeavor, consider the problem of latent design errors. Design errors affect system reliability in unpredictable ways and measuring their effects in the lab is infeasible. In systems containing latent design errors, failures of individual replicated processors are not independent and render the reasoning behind replicated strategies for fault-tolerance impotent.

A current approach to solving the problem of latent design errors is based on notions of design diversity. This approach is typically implemented by independent design groups working from common specifications. However, in an often cited paper [2], Knight and Leveson have shown, at least in the software domain, that design diversity does not necessarily ensure independence of design errors. Moreover, quantification of software reliability in the ultra-reliability range is not feasible in the presence of design errors [5]. Historically, quantification of hardware unreliability due to physical failure has not been viewed as a problem and, reliability analysts assume hardware components are immune from design errors. However, as we move into the nineties, hardware description languages, silicon compilation, ASIC's, and microcoded architectures are blurring the boundaries between hardware and software development methodologies. Based on this observation, we believe caveats regarding quantification of unreliability attributable to design errors now apply to hardware as well.

Hence, we argue for verification through mathematical proof, rather than design diversity, as a partial solution to the serious problem of design errors in digital systems. Our approach is to formally specify and verify the correctness of mechanisms that implement the required fault tolerance. Management of the distributed resources that implement the required fault tolerance is a complex systems problem. Considering the obvious requirement that the voted results produced by the replicated processors must be voted in a fault-tolerant fashion seems to lead to a vicious circle. A second difficulty arises from the fact that voted results mask errors only if each replicate receives the same inputs; thus sensor values must also be distributed to each processor in a fault-tolerant manner. Ingenious algorithms have been developed to perform these tasks [4]. Verifying that these algorithms have been correctly incorporated into the fabric of a distributed operating system is at the heart of reliable fault-tolerant system design.

2 A Science of Reliable Design

Mathematical reliability models provide the foundation for a scientific approach to fault-tolerant system design. Using these models, the impact of architectural design decisions on system reliability can be analytically evaluated. Reliability analysis is based on stochastic models of fault arrival rates and system fault recovery behavior. Fault arrival rates for physical hardware devices are available from field data or empirical models [7]. The fault recovery behavior of a system is a characteristic of the fault-tolerant system architecture.

The justification for building ultra-reliable systems from replicated resources rests on an assumption of the failure independence between redundant units. The alternative approach of modeling and experimentally measuring the degree of dependence is infeasible, see [5]. The unreliability of a system of replicated components with independent probabilities of failure can easily be calculated by multiplying the individual probabilities. Thus, the assumption of independence allows fault-tolerant system designers to obtain ultra-reliable designs using moderately reliable parts. Often complex systems are constructed from several ultra-reliable subsystems. The subsystem interdependences (e.g. due to shared memories, shared power supplies, etc.), can still be modeled (assuming perfect knowledge about the failure dependencies) and the system reliability can be computed. Of course, the reliability model can become very complex.

The validity of the reliability model depends critically upon the *correctness* of the software and hardware that implements the fault tolerance of the system. If there are errors in the logical design or implementation of the fault-recovery strategy or in the design of individual system components, failures between redundant units may no longer be independent. The quantification of system unreliability due to physical failure would be meaningless.

Based on this analysis, the validation of the reliability of life-critical systems can be decomposed into two major tasks:

- Establishing that design errors are not present.
- Quantifying the probability of system failure due to physical failure.

The first task is addressed by formal specification and mathematical proof of correctness. The second task is addressed by the use of reliability analysis models and tools to analytically evaluate the effects of individual component failure rates on the overall system reliability.

3 Formal Methods

The major difference between the approach advocated here and approaches used for design of more traditional fault-tolerant operating systems is in the application of formal methods. This approach is borne from the belief that the successful engineering of complex computing systems requires the application of *mathematically based analysis* analogous to the structural analysis performed before a bridge or airplane wing is built. The mathematics for the design of a software system is *logic*, just as calculus and differential equations are the mathematical tools used in other engineering fields.

The application of formal methods to a development effort are characterized by the following steps.

1. Formalization of the set of *assumptions* characterizing the intended environment in which the system is to operate. This is typically a conjunction of clauses $\mathcal{A} = \{A_1, A_2, \ldots, A_n\}$ where each A_i captures some constraint on the intended environment. Typically $\mathcal{A}$ has many models although the author of a specification generally has a particular model in mind.

2. The second step is the formal characterization of the system *specification* in the formal theory. This is a statement $\mathcal{S}$ characterizing the properties which any implementation must satisfy.

3. The third step is formalization in the theory of an *implementation* $\mathcal{I}$. Typically, an implementation is a decomposition of the specification to a more detailed level of specification. In a hierarchical design process there may be a number of implementations, each more detailed than its specification.

4. The final stage is a proof that the implementation $\mathcal{I}$ satisfies the specification $\mathcal{S}$ under the assumptions $\mathcal{A}$. Formally, this is a *proof* of the statement $\mathcal{A} \supset (\mathcal{I} \supset \mathcal{S})$, where $\supset$ denotes logical implication. That is, under any model of $\mathcal{A}$, $\mathcal{I}$ is an implementation of the specification $\mathcal{S}$.

Some comments are in order. If the set of assumptions proves to be inconsistent, i.e. there is no model of $\mathcal{A}$, then any implementation satisfies all specifications and the entire effort is in vain. This suggests a strategy of minimizing both the number and complexity of the assumptions. The assumptions can be seen as constraints on the operating environment in which the specified component is to be placed.

The author of the formalizations typically has some specific model in mind which he is trying to characterize in the formal statements $\mathcal{A}$, $\mathcal{S}$, and $\mathcal{I}$. From the perspective of methodology, it is a good idea to prove some *putative theorems* about these statements to ensure that the intended model has been faithfully captured. For example, in a formal characterization of a memory, say $\mathcal{M}$, it is important to ensure that the specification correctly captures notions of reading and writing. One property of interest might be that reading the contents of address a at times t_1 and t_2 will yield the same value, v, as long as there is no write to a of a value $u, u \neq v$, during the interval (t_1,t_2). This property should surely hold in any model of $\mathcal{M}$. Proving such a theorem builds confidence that $\mathcal{M}$ correctly characterizes the intended models.

It should be noted that, strictly speaking, this property could *only* be shown by reasoning about the specification; no amount of testing can establish that this property holds. In fact, many of the properties of interest in fault-tolerant design are within the domain of formal methods and their verification depends on reasoning as opposed to testing-based approaches. The existence of formal characterizations of a system provides a basis for such reasoning.

3.1 Hierarchical Proof

The methodology outlined here is inherently hierarchical. Under the assumptions $\mathcal{A}$, if implementation $\mathcal{I}_1$ is shown to be an implementation of a specification $\mathcal{S}$ and $\mathcal{I}_2$ is shown to be an implementation of $\mathcal{I}_1$ we conclude that $\mathcal{I}_2$ is also an implementation of $\mathcal{S}$. The sentence above can be formally restated as an inference rule:

$$\frac{\mathcal{A} \supset (\mathcal{I}_2 \supset \mathcal{I}_1),\ \mathcal{A} \supset (\mathcal{I}_1 \supset \mathcal{S})}{\mathcal{A} \supset (\mathcal{I}_2 \supset \mathcal{S})}$$

Logically, this is a simple consequence of the transitivity of implication. Its significance for a hierarchical verification strategy is obvious; it provides formal justification for linking together a chain of formal proofs of correctness to show the lowest level decomposition of a series of decompositions is an implementation of the original specification.

3.2 Levels of Application

Formal methods are the applied mathematics of computer systems engineering. In other engineering fields, applied mathematics are utilized to the extent that they are required to achieve acceptable levels of assurance for safety, performance, or reliability. It is often assumed that the application of formal methods is an "all or nothing" affair. This is not the case. There is a useful taxonomy of *levels of application* identified here.

0. No application of formal methods.
1. Formal Specification of all or part of the system.
2. Paper and pencil proof of correctness.
3. Formal proof checked by mechanical theorem prover.

Significant gains in assurance are possible in existing design methodologies by formalizing the assumptions and constraints, the specification, and the implementation. Experience shows that application of *level 1* alone often reveals inconsistencies and subtle errors that might not be caught until much later in the development process, if at all. It is generally accepted that the later a design error is identified the more costly is its repair, therefore this level of application can provide significant benefits.

The use of paper and pencil proof in the design process adds another level of assurance in design correctness. *Level 2* application forces explicit consideration of the relationships between the implementation and the specification and often reveals forgotten assumptions or incorrect formalizations.

A proof of correctness is only as good as the prover. Even stronger evidence for correctness can be established by forcing proofs through a mechanical theorem prover. This is *level 3* application of formal methods. It must be noted that there is no guarantee that the implementation of the mechanical prover is correct or that the hardware on which the mechanical verification was performed was not faulty. Thus, there is no absolute guarantee of the correctness of an implementation even after a mechanical proof has been performed. What is gained by the additional effort is a detailed argument for the correctness of the implementation. The process of "convincing" a mechanical prover is really a process of developing an argument for an ultimate skeptic who must be shown every detail.

Partial application of any of the levels is possible for different parts of the system. We advocate the application of level 3 formal methods only for the most critical (and hopefully reusable) system components. What is classified here as level 1 and level 2 formal methods are being widely applied in the U.K.

4 Architectural Approach

In our research at Langley on provably correct fault-tolerant control systems we consider architectures consisting of four or more electrically isolated processors that can communicate with one another. Typically these systems run with a static multi-rate schedule with tasks scheduled periodically. Each processor synchronously executes the same schedule, and the system votes all actuator outputs to mask individual processor faults. The fault models used are worst case models in which faulty processors can maliciously cooperate in attempts to defeat the fault tolerance of the system. Under this worst case model, $3m + 1$ processors must be working in order to tolerate m faults [1]. If we assume the existence of a fault-tolerant basis providing clock synchronization and interactive consistency, then a simple majority of working processors suffices to out vote any minority of faulty processors.

Empirical evidence indicates that transient faults are significantly more common than permanent faults. If designed correctly, these systems are able to recover gracefully from transient faults. Each computation generally only depends on a short part of the input history and typically have only a minimal amount of global state information. If the global state is voted periodically and internal state is recoverable from sensors, it is clear that after some finite time errors can be flushed from the system.

The approach adopted here for the design of the distributed aspect of the system is motivated by Lamport's paper [3]. At the base of the system is a distributed clock synchronization algorithm, allowing the system to be viewed as a synchronous system. Under contract to NASA, Rushby and von Henke [6] formally verified Lamport and Melliar-Smith's [4] clock synchronization algorithm[1] providing a key system building block. In a system relying on exact match voting it must also be ensured that each processor receives the same inputs from the sensors. This is accomplished by a Byzantine resilient interactive consistency algorithm running on the

[1]Interestingly, they found at least one error in the published proof that had remained undiscovered through the social review process.

distributed system. With these algorithms as a base, the voter ensures that as long as a majority of the processors are working then the replicated system produces the same results as an *ideal* non-faulty processor would.

5 Conclusion

It has been argued that quantification of system reliability in the ultra-reliable range depends on the provably correct implementation of fault tolerance. Absolute correctness is unattainable. However, formal methods provide added assurance of correctness by forcing detailed consideration of the assumptions, the specification, and the implementation in a formal setting. Hierarchical design proofs provide a formal framework to allow consideration of these details at the appropriate level of abstraction. These methods are being applied in research efforts underway at NASA LaRC. A NASA technical report outlining the first phase of design specification and proof of a fault-tolerant operating system for control applications will be available in the near future.

References

[1] D. Dolev, J. Y. Halpern, and H. R. Strong. On the possibility and impossibility of achieving clock synchronization. In *Proceedings of 16th Annual ACM Symposium on Theory of Computing*, pages 504–511, Washington, D.C., April 1984.

[2] J. C. Knight and N. G. Levenson. An experimental evaluation of the assumptions of independence in multiversion programming. *IEEE Transactions on Software Engineering*, SE-12(1):96–109, Jan 1986.

[3] Leslie Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Transactions on Programming Languages and Systems*, 6(2):254–280, April 1984.

[4] Leslie Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, January 1987.

[5] Doug Miller. Making statistical inferences about software reliability. Technical Report CR-4197, NASA, December 1988.

[6] John Rushby and Frieder von Henke. Formal verification of a fault tolerant clock synchronization algorithm. Technical Report 4239, NASA, June 1989. Contractor Report.

[7] U.S. Department of Defense. *Reliability Prediction of Electronic Equipment*, January 1982. MIL-HDBK-217D.