DEVELOPING INTEGRATED HARDWARE-SOFTWARE RELIABILITY MODELS: DIFFICULTIES AND ISSUES

Mark A. Boyd  
Computational Sciences Division  
NASA-Ames Research Center  
M/S 269-4  
Moffett Field, CA 94035

Christine M. Monahan  
Recom Technologies  
NASA-Ames Research Center  
M/S 269-4  
Moffett Field, CA 94035

ABSTRACT

The development of integrated hardware-software system reliability models is very difficult. This paper discusses some of the differences between hardware and software reliability modeling which make integrating them together so hard. It also discusses issues that are unique to each and common to both, and lists open problems that need to be resolved.

INTRODUCTION

Pressures exist in the field of Digital Avionics that are leading to more and more use of embedded computer control systems in which software and hardware play equally important roles. This means that the reliability of the software is as important as the reliability of the hardware to the overall performance and reliability of the system. The effect of this is particularly significant to safety critical systems (such as some that may be found on commercial aircraft) and systems that may be inaccessible after deployment (such as spacecraft intended for deep space missions). In such systems, a realistic appraisal of the overall system reliability must take both hardware reliability and software reliability into account. This is not always done, however. In some cases, only a hardware reliability analysis is performed. On the occasions where both hardware reliability and software reliability are considered, often a separate analysis is done for each. The drawback here is that there are no widely accepted or standard methods for combining results of separate hardware and software reliability analyses together into a meaningful composite result. Ideally, the best approach would be to use a method which models both hardware and software reliability in one integrated system model.

The development of system reliability models which accurately represent the failure behavior of both hardware and software components of a system in one integrated model is a notoriously difficult task. The job is complicated by the fact that the failure processes of hardware and software are intrinsically different. Furthermore, the topic of how to accurately model software reliability is itself not as well understood as is the topic of hardware reliability modeling, and currently there is disagreement within the field of software reliability evaluation about how best to measure the reliability of software. This paper discusses some of the differences between hardware and software reliability modeling which contribute to the difficulty in integrating the two of them together into one overall system reliability model. It also covers some issues that are unique to each and common to both, and lists some open problems that need to be resolved before satisfactory methods are likely to appear for developing integrated hardware-software system models of practical use.

HARDWARE RELIABILITY MODELING

There are several fundamental obstacles to integrating hardware and software reliability modeling together. One of these is the fact that the failure processes for hardware and software are completely different in nature. This section describes the failure process and lists some typical modeling techniques for hardware. The next section will do the same for software.

Failure Process for Hardware

The failure process modeled by hardware modeling techniques is primarily a physical degradation process. The typical lifecycle reliability behavior of a hardware component is illustrated by the well-known "Bathtub curve" lifecycle diagram (see figure 1). The diagram depicts the relative change in failure rate of the component during three major phases of the component's life: an initial "infant mortality" burn-in period characterized by a relatively high but decreasing failure rate during which components that were improperly manufactured fail before their indicated design lifespan expires; a relatively stable "useful life" middle phase; and a final "wear-out" phase during which the failure rate increases from increased physical degradation due to age. During the middle "useful life" phase, the component is assumed to have demonstrated a lack of manufacturing defects by its survival beyond the "infant mortality" period. Because the component is not yet old enough to experience age-based wear-out failures, all failures that do occur are assumed to be caused by stress events which occur randomly at a relatively constant rate that is determined by the hardness of the component and to some extent by the environment in which the component operates. This paradigm fits the reliability behavior of electrical components well.

Hardware reliability modeling traditionally employs an implicit assumption that a particular component is operating correctly at the beginning of a "mission" (performance interval). This implies the absence of both physical defects and design defects. Until recently, the assumption of absence of design defects often could be justified through exhaustive testing of the input-output function responses of the component to verify its design correctness. The initial absence of physical defects could be justified in the same way and through "burn-in" testing. These assumptions eliminate a significant proportion of the possible causes of component failure, leaving only failures due to random environmental stresses that are characterized by the failure rates of the various phases of the component's lifecycle. It is unreliability due to these types of failures that traditional hardware reliability modeling tries to assess. Because of the stochastic nature of these types of failures, most quantitative hardware modeling techniques are themselves stochastic, aimed at estimating a probability of failure over a period of time (mission time). Indeed, the definition of reliability for hardware is usually given as the probability that a component (or system) does not experience a...
failure during a specified period of time; i.e. reliability is defined as a function of time.

![Component Lifetime](image)

Figure 1: The "Bathtub Curve" Lifecycle Diagram

The "Bathtub curve" lifecycle diagram provides an underlying paradigm for the failure process of hardware. This allows hardware failures to be modeled stochastically as responses to a sequence of environmental failure-causing stresses that occur randomly according to some probability distribution which can be characterized by mathematical equations. The form of the mathematical equations depends on underlying assumptions upon which the model is based. For example, an assumption that the component remains in its "useful life" period throughout the period of evaluation (and hence has an essentially constant failure rate) translates into a sequence of failure events with two characteristics:

- the number of failure events that occur within any fixed interval \((0, t]\) has a Poisson distribution, i.e. \(P(N, = r) = e^{-\lambda} \frac{\lambda^r}{r!}\), where \(N,\) is a random variable denoting the number of failure-inducing stresses occurring in the interval \((0, t]\). \(\lambda\) denotes the constant rate at which the stresses occur, and \(P(N, = r)\) denotes the probability that \(r\) stresses have occurred in the interval \((0, t]\).

- the time between any one failure event and the next (the interarrival period) has an exponential distribution, i.e. \(P(X = r) = e^{-\beta} \), where \(X\) is a random variable denoting the time from one failure event (which occurs at time \(t=0\)) to the next, \(P(X = r)\) denotes the probability that the time until the next failure event is greater than \(l\), which is equivalent to the reliability \(R(l)\) of the component when it is the time until the first failure event that is being evaluated.

The bottom line of this discussion is that reliability for hardware components can be estimated using models that are based on a well-defined failure process paradigm; can be expressed in terms of mathematical equations; and which model failures stochastically (i.e. in terms of probabilities).

Modeling Methods

From the mathematical representation of the failure behavior of individual components, modeling methods exist to derive mathematical representations of the failure behavior for overall systems of many components. Most hardware reliability modeling techniques are oriented toward either evaluating the effect of specific individual component failures on the overall system (quantitative methods), or on tracing the propagation of failures throughout the system (qualitative methods). Quantitative methods attempt to arrive at a numerical estimate of the probability of failure of a system or subsystem. They include methods such as fault trees[2] and reliability block diagrams[1], Markov models[3], hybrid and hierarchical models[4], and simulation[5]. Qualitative methods do not aim for a numerical estimation, but instead attempt to identify combinations or sequences of failures or events that significantly impact system reliability. Examples of qualitative methods include Failure Modes and Effects Analysis (FMEAs)[6], digraphs[7], and fault trees. Note that some methods can be used in either a quantitative or qualitative manner; specifically, FMEAs and fault trees fall into this category. An important simplifying assumption of most quantitative and some qualitative methods is that individual component failures are statistically independent.

SOFTWARE RELIABILITY MODELING

We now consider the failure process of software, how it differs from that of hardware, and some existing methods for modeling software reliability.

Failure Process for Software

The failure process that must be modeled for software is fundamentally different than the failure process that is traditionally modeled for hardware. The most obvious difference is that there is no physical degradation process associated with software. Instead, the failure process to be modeled in software consists entirely of an executing program encountering design defects (latent faults in the code) that cause incorrect functioning of the software. In a nutshell, traditional hardware reliability modeling assumes a complete absence of design defects and concentrates solely on failure due to physical degradation, whereas software reliability modeling has no physical degradation component at all and concentrates solely on failures due to design defects. The term "design defects" includes errors in the initial design, errors introduced during "bug fixes", and errors introduced when the software is enhanced to add new features.

From the "Bathtub curve" paradigm, stochastic models of the physical degradation failure process of hardware can be formulated that are good abstractions of the relevant failure behavior and are also widely and generally applicable across a broad range of hardware component types. However, to our knowledge there is no comparable lifecycle paradigm for software's "design defects" failure process that is widely accepted, comprehensive, and generally applicable across a broad range of software types. The "design defects" software failure process includes aspects that are complex enough and diverse enough that there exist no modeling methods that capture the effects of all of them: sufficient accuracy to be able to predict the failure behavior of a piece of software solely from its physical characteristics and development process (environment). This is a consequence of the fact that software is a product of human creative activity and fallibility. The presence or absence of design defects in a piece of software can be influenced by such factors as: the development control process used (lone hacker vs. traditional structured design and analysis vs. formal methods), the software architecture type (procedural vs. object-oriented, single-thread vs. multithread, localized vs. distributed (client-server)), the implementation language, the type of program and type of use (i.e. is it a real-time flight control system with independent periodic control loops, or is it a commercial program like Microsoft Word(TM), budgetary and time schedule constraints, etc. Even the skill level and experience of individual programmers can play an important role. The mechanisms by which all of these factors influence the reliability of the software are not understood, so generally it is not known how to translate an arbitrary assumption about one of these factors into mathematical equations describing a stochastic model of the software failure process. Formulating a stochastic model that incorporates the combined effect of assumptions...
about all these factors together is an even more difficult task. As a result, many of the techniques for modeling software reliability tend to rely on assumptions covering only some aspects that influence the software failure process and not others. The practical consequence of this is that many different software modeling techniques have been developed, and individual modeling techniques each tend to be applicable only to specific, relatively narrow classes of software types (e.g., software of a particular architecture, aimed at a particular application, category of use, developed with a specific type of development process, etc.). Usually it is not obvious which modeling method is the best one for a given piece of software, and a major challenge for the software reliability analyst has been matching the most appropriate of the available modeling methods to a specific piece of software under analysis. Currently this must be done empirically using statistical analysis. There are several practical difficulties inherent in this approach. We will discuss these and other problems in a following section.

Software Reliability

Reliability for software is usually defined in a similar manner as that for hardware: the probability of failure-free operation of the software for a specified period of time in a specified environment[8]. However, the relationship between failure manifestation and time is different. For hardware, an operating component is assumed to experience an outside stimulus which causes it to fail, i.e. a defect is induced that did not previously exist. What is modeled stochastically is the time to the occurrence of the outside stimulus that induces the failure. However, for software there is no creation of physical defects during operation. All defects in the software component are latent defects which are present within the software at the time operations begin until the end of the performance period. A latent fault manifests itself when the right conditions of input data and execution sequence cause the fault to be "activated", i.e. cause an erroneous internal state in the program which then causes the program to deviate from its intended behavior. For software, then, what should be modeled stochastically in order to be consistent with hardware modeling practice is the time to manifestation of any latent faults present in the software. This is extremely difficult to do in practice. The time to manifestation of latent faults depends on at least two characteristics of the software: the number and location of the latent faults present in the software, and the frequency with which the various execution paths through the program are executed (which in turn can depend on how the software is used, by whom, and what the skill level of the user is). We are aware of no comprehensive techniques for modeling either of these directly with high accuracy. Instead, methods have been developed that attempt to model them indirectly. For example, one can attempt to characterize the number and location of latent faults by estimating a measure of failure intensity of the software (the average number of failures encountered throughout the program per unit execution time[8]). The execution path frequency is essentially what Musa[8] defines as the operational profile (the set of run types that the program can execute along with the probabilities with which they occur). The operational profile can be characterized either in terms of sets of inputs, sets of execution paths, or (more abstractly) in terms of types of user and expected use. Currently all of these are modeled with statistical approximations, and the parameter values of these models (i.e., the failure intensity, the operational profile probabilities, etc.) generally have a significant uncertainty associated with them.

Modeling Methods

There are three general methods of measuring software reliability, all of which depend upon software testing[9].

The first, Fault Seeding[10], is the analog for software of fault injection for hardware. It involves the artificial introduction of new faults into a program before subjecting the program to a testing regimen. The proportion of the seeded faults (for which the total number is known) discovered by the testing regimen is then used as an estimate for the proportion of latent faults present in the program (for which the number discovered is known, but the total number is not known).

A second method, Sampling, involves testing the program with a random sample of points from its input domain tailored to be consistent with the expected operational profile. The number of failures is then observed (no faults are removed from the software during testing), and the reliability of a single run of the program is estimated by: 

\[ R = 1 - \frac{1}{n} \]

where \( n \) is the number of failures observed, and \( n \) is the total number of runs executed during the test.

The third method, Reliability Growth Models, attempts to characterize the change in failure intensity of the software as it undergoes testing and debugging. This method appears to be the most promising for integrated software-hardware modeling because it can directly produce an estimate of failure intensity (needed to link into the hardware reliability model). The models express failure intensity with mathematical equations in terms of parameters, such as the number of faults initially present before testing, failure intensity decrement per fault corrected, etc. There are many different models available which differ with respect to underlying assumptions about the various factors that influence the presence of faults in the software and the effect of the fault removal process employed during testing. These differing assumptions manifest themselves as differing forms of mathematical equations, differing numbers of and types of parameters, whether execution time data (time between faults) or interval data (number of faults between fixed intervals of time) are used[11], etc. Musa[8] describes two models, a basic execution time model and a logarithmic Poisson execution time model, for which the major difference is the decrement of the failure intensity functions. Both assume failure intensity varies with time since faults can be both fixed and introduced as testing progresses. However, the failure intensity function decrement is constant in the basic model, whereas in the logarithmic Poisson model it becomes smaller with each successive failure experienced. The various assumptions give the models' predicted failure behavior different properties which, depending on the specific software, may fit the actual failure behavior well or not at all. For example, Musa indicates that programs with a uniform operational profile are better modeled with the basic model than the logarithmic Poisson model, whereas programs with non-uniform operational profiles are better modeled with the logarithmic Poisson model. This property was empirically observed rather than predicted analytically[8], and it is not clear how the degree of operational profile uniformity causes the per fault change in failure intensity over the testing period to range between constant and logarithmic decrements. This exemplifies the current need for empirical matching of model types to specific software, since it is often not yet possible to determine model-software matches directly from the characteristics of the software.

INTEGRATED HARDWARE-SOFTWARE RELIABILITY MODELING

What is needed to be able to produce a reliability model that integrates hardware and software failure behavior? The foremost requirement is to have a common concept through
which hardware reliability modeling methods can be linked to software reliability modeling methods. For hardware, component failure rates derived from the "bathtub curve" paradigm are used as a basis from which to develop stochastic models that can estimate reliability. A comparable concept of failure rate for software would permit integrated hardware-software models to be constructed, using many of the same modeling methods commonly used for hardware reliability modeling, in which software would be treated like any other (hardware) component. The best technique available seems to be to approximate the software failure rate using failure intensity, which in turn is estimated from data on observed failures obtained during testing of the software. Care must be taken that the time scale used in the software failure intensity is consistent with the time scale used in the hardware failure rates in order to avoid distortions that can occur, for example, as a result of different processor execution speeds[8].

The validity of the software failure intensity estimated from a reliability growth model depends heavily on how well the failure behavior of the software matches the assumptions underlying the model. In view of this, a second requirement for developing integrated hardware-software reliability models is having available ways of accurately matching the most appropriate reliability growth model to any specific piece of software. In addition, an important use of reliability modeling for life-critical embedded systems is to help compare different system architectures during the design phase before any system is actually built. For these types of systems, this adds a more stringent requirement of accurately performing the software-to-reliability growth model match without needing to use statistical data obtained by testing the software under study.

Failure intensity estimates from reliability growth models currently are dependent on data obtained during testing and debugging of the software. In addition, it is widely accepted that the outcome of testing depends on the operational profile[8]. This points to a need for efficient ways of determining accurate operation profiles.

Most of the hardware modeling methods mentioned earlier assume that components fail independently of each other. This may not always be the case when there are both hardware and software components in an integrated model. For example, a failure of a processor on which a software component is executing will certainly cause the failure of the software component. Conversely, a failure in software that controls hardware reconfiguration and fault recovery can affect system hardware behavior in potentially complex ways. This raises the need for modeling methods that can accurately represent situations where hardware and software components interact and can influence the failure behavior of each other. Recent work[12-15] addresses this topic.

ISSUES AND OPEN PROBLEMS

A second obstacle to integrated hardware-software reliability modeling lies in the fact that the field of software reliability evaluation is less mature than the field of hardware reliability evaluation. There are two major reasons for this: it is a younger field than hardware reliability evaluation (by several decades), and the software failure process is arguably much more complex to model than the hardware failure process. As a consequence of this relative immaturity is less consensus on definitions of some basic concepts and more controversy over approaches for measuring metrics and validating modeling techniques for software than for hardware. This section describes some resulting unresolved issues and open problems that contribute to difficulty in developing models for software reliability, which in turn contributes to difficulty in meeting the requirements for integrated hardware-software reliability modeling outlined in the previous section.

Difficulties in the Software Reliability Field

Concepts - A major indication of immaturity in a field is a lack of agreement on some basic concepts. An example of this is pointed out by Hamlet[16] when he observes that there is no established theory of software dependability, nor is there an underlying theory for testing. He further notes that disagreements exist about what software testing is supposed to do. He identifies four different common approaches to testing which can lead to very different assessments about what kind of testing best establishes that software can be trusted. All four approaches have some kind of significant theoretical or practical drawback that diminishes confidence in their applicability. This is a significant problem because all software reliability estimation methods currently depend on some form of testing.

Software testing - The reliance of current methods for estimating software reliability metrics on testing of the software means that any inherent limitations in software testing methods can have a major impact on reliability estimates. Simply put, testing of software involves executing a program many times, each time with a set of input data, and observing whether the program produces correct or incorrect behavior or output for each set of input data. The sets of input data either may be chosen randomly from the program's input domain, or they may be chosen systematically[10]. Several complications may arise as a part of the testing process. For example, for each run of the program using a set of input data, a determination must be made as to whether the program executed correctly or incorrectly. This implies the need for an "oracle" which is capable of making that determination. Depending on the function of the program, such an oracle may be difficult to implement, or be costly in terms of execution time, or both[17,18]. Secondly, the failure behavior of the program can depend on the operational profile from which the sets of input data are generated[8]. Depending on the program, obtaining accurate operational profiles for programs can be difficult to do[18]. Finally, the effectiveness of a testing regimen to detect faults in a program is dependent on a property called testability, which is defined as "the probability that, if a program P contains fault(s), P will fail under test[17]." However, testability analysis is in its infancy[17]. As a result, there are no methods currently available for accurately measuring testability. As a consequence of this, an outcome of no program failures detected during a testing regimen does not guarantee that there are no faults in the program; it only demonstrates that any faults that are present were not detected by the testing regimen. This can be an important limiting factor on the validity of an estimate of software reliability derived from testing data for the program. It is an important point because all reliability estimation techniques currently available rely on testing, and hence all of them are subject to this limitation.

A final point applies in particular to life-critical software that requires very high reliability. The current state-of-the-art in testing is not sufficiently advanced to be able to measure software reliability in the ultrareliability range (<10^-7 failures per hour execution time)[19]. Some researchers are attempting to address this problem[17].

Software reliability modeling methods - The primary unresolved problem is that of accurately matching the software with the best reliability growth model to get an accurate failure intensity estimate. This is important precisely because most existing models are not broadly applicable. The various
factors involved in developing software can have a potentially wide influence on the reliability of the software through mechanisms that are not understood and cannot yet be directly modeled. Consequently, the various sets of combinations of these factors must be empirically matched to the available reliability growth models. Because of difficulty in identifying all the relevant factors for an individual piece of software, this empirical matching should be performed for every program analyzed. Currently this matching can only be done by statistically fitting observed failure data to predicted behavior of the various reliability growth models, and painstakingly verifying that each assumption underlying a candidate model is valid for the software under analysis. This can only be done effectively for software developed with a strictly controlled, well-defined, and well documented development process[20]. This means that no allowances are possible for shortcuts inspired by budget and schedule constraints. This is not usually consistent with reality in actual software development as commonly practiced today.

The avionics software developed for the Space Shuttle by IBM is an example of software that is ideally suited to the use of reliability growth models to estimate failure intensity (and other metrics) for use in integrated hardware-software reliability models[21]. A well-defined, strictly controlled development process is used to maintain and enhance the code. A reliability growth model[21] was matched to the software and rigorously validated as accurately modeling the actual failure behavior of the software. Data generated by the software during development and testing were used to do the validation. All of the assumptions underlying the model were closely examined and their applicability verified. For software that doesn't have the benefits of all of these characteristics, however, software reliability growth models may be of more limited use in predicting failure intensity at points in the future. Lack of a well-defined development process can make validation of assumptions underlying growth models difficult and hence make it hard to select an appropriate growth model. Lack of a strictly controlled development process can let excessive variation in the various fault-influencing factors prevent any one model from accurately representing the actual failure behavior of the software. Lack of an established development process and existing history of failure data can make validation of a selected growth model difficult or impossible. Even when an established process exists, lack of a history of failure data (as is the case for new projects) can make matching an appropriate reliability growth model to the software uncertain, especially early in the testing and debugging phase. This was the experience of a recent project conducted within our research group that analyzed failure data from the flight control software for the X-29A experimental aircraft program during full-up system testing[22]. An attempt was made to match various software reliability growth models available in the SMERFS[23] modeling package to the flight control software at various points along the system testing schedule, duplicating what program managers might have seen had they performed the analysis as the testing schedule evolved. It was found that the small number of data points (failures) led to sufficiently large statistical "noise" errors during the curve fitting that none of the growth models available in SMERFS provided usable predictive estimates of future values of failure intensity, including the model later selected as best fitting the full set of failure data from the completed test.

For software that is to be part of a high-reliability, fault tolerant embedded system, the model matching problem is particularly difficult. Reliability modeling for such systems is often used iteratively to evaluate different candidate designs before any system is actually built. However, no history of failure data exists for such a system that has not been completely designed or built. In this kind of situation, the best that can be achieved is to use experience gained from an earlier "sufficiently similar" system as the basis for making a model match for the new system. "Sufficiently similar" should mean that many of the fault-influencing factors as possible should be the same in both projects. For example, the architecture of the software and the system should be very similar to the existing system for which the required history of failure data is available, and the new system should be developed by an experienced team following a strict development process that was used on the existing system and had previously yielded a known failure data profile. These criteria can be difficult to meet, especially for new state-of-the-art systems that are pushing existing design envelopes.

Controversies in the Software Reliability Field

Contrasting Approaches - There are two obvious ways of reducing the number of faults in software: you can find and remove them from the finished program, or you can make sure the faults don't get into the program in the first place. The former approach is the one traditionally followed in software engineering and has been the basis for the discipline of software testing. The latter approach has been emphasized only relatively recently, and is the basis for using formal methods[24] in software development. The emergence of formal methods has given rise to a controversy within the software community about which approach achieves the best reliability. Some proponents of formal methods believe that formal methods are potentially capable of producing defect-free software so that testing will no longer be required, whereas some proponents of testing reject this notion as confusing the following of procedures with real success, and consider formal methods as merely the latest fad in software development[16, 17]. A third faction advocates a methodology called Cleanroom Software Engineering[25], which includes elements of both testing and formal methods and is also very controversial. In practice, many software engineers are very reluctant to abandon testing and rely on procedural methodology alone. However, it is reasonable to expect that use of formal methods can reduce the variability in the fault-influencing developmental factors and consequently increase the accuracy of failure intensity estimates derived from models. Formal methods may also provide a means for achieving actual reliability that is provably greater than what is measured by testing techniques[17]. This is likely to be a key element in validating ultra-reliable software, for which current testing-based validation methods are inadequate[19]. In view of all this, we feel that it is likely that both testing and formal methods will retain important roles in software reliability estimation.

CONCLUSION

We have identified some important obstacles to developing integrated hardware-software reliability models, especially when life-critical embedded systems are involved. One obstacle involves a fundamental difference in the nature of the failure processes of hardware and software which gives rise to metrics and modeling methods for the two that are difficult to integrate together. A second obstacle involves the relative immaturity of the field of software reliability evaluation compared to the field of hardware reliability evaluation, as evidenced by controversies and disagreements between experts in the software field over approaches and definitions of concepts needed for model integration. Software failure intensity appears to be the most promising unifying concept for linking hardware and software modeling techniques together. However, all current methods for estimating failure intensity rely on statistical fitting of failure data obtained during testing of the software, and the validity of results obtained from testing is itself dependent on the testability property of the
software, for which no accurate methods of measurement yet exist. In addition, many factors that influence the presence of faults in software do so by mechanisms that are not understood and cannot yet be modeled directly. Consequently, software failure intensity must be estimated from means such as reliability growth models, which must be empirically matched to each piece of software under analysis. That process of matching a piece of software to the growth model that most accurately represents the software’s actual failure behavior is a difficult and uncertain task. This matching is particularly difficult when reliability modeling is used to compare competing design options before a system is actually built, when no failure data upon which to estimate a failure intensity exists.

What research topics need to be addressed to help overcome these obstacles? In the short-intermediate term it would be helpful to have a repository of knowledge describing which reliability growth models match best with various combinations of developmental factors (software architecture, development process, etc.), and how sensitive the predictions of a specific model are to variations in the individual factors. In the long term, research needs to be directed at understanding the mechanisms of how the factors influence the presence of faults in software, so that it becomes possible to predict reliability directly from the characteristics of the software and development process without relying on any failure data collected during execution and testing of the program. Until that is possible, estimation of software failure intensity by statistical fitting of failure data from testing will be required. In view of this, research on testability analysis aimed at developing accurate means for measuring testability is needed. Work is also needed to improve testing methodology to enable certification of reliability greater than what is actually measured by the testing regimen[17]. This appears to be the only way that the limitations of testing ultrareliable software[19] can be overcome to enable practical verification of ultrareliable hardware-software embedded systems. Further research is also needed in the areas of obtaining accurate operational profiles, and developing methods for modeling hardware-software interactions.

REFERENCES