JOURNAL OF AIRCRAFT
Vol. 29, No. 3, May–June 1992

Software Reliability for Flight Crew Training Simulators

George E. Stark*
MITRE Corporation, Houston, Texas 77058

Flight crew simulator failures are costly and may have an impact on the timing or efficiency of a mission; thus, reliability is one of the most important issues facing simulator developers today. The reliability of a simulator is the probability that a training session of length t can be completed without a failure. This paper defines simulator failure and then identifies and compares the three sources of simulator failure: hardware, software, and human, focusing on the cost of software failure. The paper next describes a model for software reliability measurement and proposes a method for establishing a software reliability objective. Data from the NASA Shuttle Mission Training Facility illustrate the technique. Finally, the paper examines the implications of using the method on the software testing successes.

Introduction

OVER the years flight crew training simulators have grown in cost and complexity as have the real world systems they model. The modern simulator is a large-scale real-time man-in-the-loop computer system designed to provide the flight crew with a safe, realistic environment in which to exercise procedures related to normal and anomalous conditions. The flight crew simulator depicted in Fig. 1 is a typical arrangement of a simulation host computer with distributed nodes that provide visual scenes, motion to the crew station, instructor capabilities, and operator controls.

Flight crew training simulators often require integration with both actual vehicle avionics hardware and software models of supporting avionics systems to reproduce real-world scenarios. In general, software models are used instead of the actual hardware if a set of interconnecting real-world systems can be simplified into a single model, or if the actual hardware is too expensive (in terms of dollars or computational resources).

Simulator software involves time-critical operations synchronized to external events and processes. It is expected to have a predictable response to inputs from a pilot, to malfunctions inserted by an instructor, and to inputs from external interfaces. The software models (outlined in the simulation host shown in Fig. 1) are programs that generate realistic outputs from the inputs provided. These programs are usually subdivided into tasks that contain individual entry/exit points so that the execution frequency can be scheduled in an order that ensures minimum calculation latency. Tasks must be completed at predetermined intervals so that time-critical input/output may be serviced with accurate and complete data. A real-time operating system is required for responding to hardware interrupts and maintaining control over the various software tasks. Clearly and completely specifying the interfaces, timing, and scheduling of the individual programs and tasks is necessary for the success of the simulator implementation.

Specification of simulator software requirements involves identifying the relationships that should exist between the inputs from the external events/internal processes and the outputs to the crew and instructors. These relationships may be explicit or implicit in a specification. An example of a requirement is that no response to crew input should take more than 150 ms longer than the actual vehicle response lag.[1] A more complicated example is a visual display that accepts input from several sources and produces a detailed visual scene with ten moving objects, texturing, and occlusion.

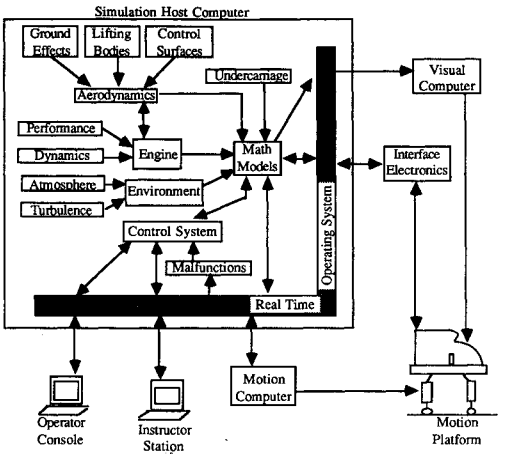

Fig. 1 A typical crew training simulator architecture.

Definition of Failure

A simulator fails whenever the produced outputs do not satisfy the input/output relationships specified in the requirements. Three sources of failure exist in any simulator: hardware (HW), software (SW), and human. Table 1, adopted from Shooman,[2] displays comparative failure mechanisms for each of these failure sources. This table indicates that all three sources have different failure mechanisms, but that each does contribute to the system failure rate. Because all three components are required to work for a successful training session, the probability of a successful training session of length t (i.e., the reliability of a simulator) can be calculated as the product of the three reliabilities, or

$$R(\text{sim}_t) = R(\text{HW}_t)*R(\text{SW}_t)*R(\text{human}_t) \qquad (1)$$

Software developers assume that the computer instructions are recorded on a stable medium and that the processor will always execute a given machine instruction exactly as stated in the processor specifications. Whenever these assumptions are violated, the software developer claims the hardware has failed. Furthermore, software developers assume the operations personnel always follow correct procedures in loading the software and that instructors use correct command se-

Presented as Paper 90-3122 at the AIAA Flight Simulation Technologies Conference, Dayton, OH, Sept. 17–19, 1990; received Aug. 2, 1990; revision received April 23, 1991; accepted for publication April 23, 1991.

*Lead Scientist, Data Systems Engineering Department. Member AIAA.

Table 1 Comparative failure mechanisms—hardware, software, and human

Failure mode	Hardware (HW)	Software (SW)	Human
Fabrication	Bad solder joints Bad component installed Mechanical misalignment	Typographical error Wrong version subroutine Incompatible operating system	Wrong disk mounted Wrong switch position set Reload button pushed in error
Design error	Component rating too low Metal parts exposed to corrosion An address load clears the accumulator	Incorrect re-initialization Interchanged branches Series expansion does not converge for all values	Enter wrong data in response to a request Illogical key data sequence entry Operator follows incorrect user guide steps and clears memory
Component overload	Capacitor with 50-V rating is used in circuit with 100-V transients HW cannot keep up with 1200-baud input even though the specs call for operation at that rate	Control system designed to handle 100 targets, drops targets without warning when more than 100 enter the control zone Input module of a text editor cannot keep up with a fast typist	Operator cannot handle more than 50 targets without overloading vigilance capacity An operator forgets the proper command sequence because there are too many steps
Wear out	A mechanical clutch slips after 5000 h Insulation cracks on wires after 10 yr, causing shorts	No analogous effect	Errors due to cumulative fatigue

quences while interacting with the software. Whenever these assumptions are violated, a human failure is said to occur.

The input and output spaces for flight simulator software are usually so large and the required relationships between them so complex that, even applyling the best software engineering technique, situations will arise in which the requirements are not met. Whenever this happens, a software *failure*[3] is said to occur. The cause of a failure is termed a *fault*,[3] and it is the responsibility of the software engineer to eliminate it either by substituting correct instructions for erroneous ones or by providing missing data. The *reliability*[3] of the software is, thus, defined as the probability of failure-free operation for a specified time interval in a specified environment.

Cost of Software Failure

Training simulator software failures result in a directly quantifiable cost as well as a qualitative cost. The directly quantifiable cost has three components: 1) the cost of lost training time; 2) the cost of restoration; and 3) the cost to repair the fault in the software. The lost training time cost includes not only the cost associated with failure isolation and equipment downtime but also the cost of the instructor and the crew time. Restoration cost includes the cost of the time it takes to reinitialize the simulator and return it to a point at which the training session can continue. For example, assume a simulation fails 2 h into a simulated mission, after restoring the simulator to operational state, the simulator must be flown to the point at which the failure occurred to continue the training session. The repair cost is generally an off-line cost, because the repair is usually completed by a staff of programmers at a remote location and may be handled through a separate discrepancy reporting process that has a time lag associated with it.

In the case of the Space Shuttle Mission Training Facility (SMTF), located at the NASA Johnson Space Center, the estimated cost of running a training session is $6450/h.[4] During the first quarter of 1986, the downtime associated with a *critical* software in the facility averaged 1.21 h, including the restoration time, which averaged 10 min.[5] (SMTF failures are classified as *critical*, *degraded*, and *no lost time*. A *critical* failure is a failure that results in a hard stop of the simulation or requires that the simulator be brought down and repaired prior to continuation of the training session.) Actual repair data for the SMTF was not available. For the sake of illustration it is assumed repairs take an average of 5 programmer hours for problem analysis, at a rate of $50/h, and 5 programmer on-line coding hours (with terminal access to a developmental facility), costing an average of $75/h, were required to repair each fault. Thus, the total direct cost of a critical software failure would be $8429.

In addition to the quantifiable costs of a software failure, there are the qualitative costs related to the timing of future missions and to negative training. For example, if a crew member must complete a specified number of simulator hours to gain certification in some mission maneuver, the reliability of the simulator impacts that member's ability to accumulate the required hours. This may cause the mission to be postponed. If the mission is executed on schedule, the inability to complete training increases the risk of the maneuver during the mission. The impact of negative training on the crew members, although difficult to measure, is perhaps the most important cost. A software failure that repeatedly disrupts the flow of the task being trained may cause the student to execute incorrectly a maneuver in the real vehicle, thereby jeopardizing the success of the mission.

Given these costs, it becomes apparent that software reliability should be specified during procurement of training simulators. If the software performs below the specifications, the developing contractor should be assessed a penalty. The penalty should be proportional to the schedule impact experienced by the user due to the rework required to meet the requirement. Furthermore, the reliability should be measured during training operations so that corrective action can be taken if changes to the simulator cause the reliability to drop below the required level.

The following sections describe a method for measuring the reliability of simulator software and propose a method for establishing a reliability objective based on minimizing the costs of the software throughout the operational life of the simulator. Finally, impacts on the developer's testing process are discussed.

Software Reliability Measurement

The inputs to be processed by the simulator software arrive in random order and at random times. In practice, an operational piece of software operates successfully for the vast majority of its inputs. It is only under a specific set of input conditions that some inherent fault manifests itself as a failure. Therefore, software failure occurrence can be considered a random process. Furthermore, even if the software is perfect, there is no way to prove this with certainty on large systems. Proving software correct has been demonstrated on small programs, but is virtually impossible on large systems.[6] The best that can be done is to establish (with an appropriate level of confidence) that the probability of failure is small compared to the consequences of such a failure. That is, a higher probability of failure is acceptable if a software failure results in a small monetary loss than if the failure results in the loss of an expensive spacecraft or human life. In either case, however, the probability of failure must be verified.

Over the past twenty years, more than forty models have been proposed for estimating the reliability of large software systems.[7,8] Among these, the nonhomogeneous Poisson process (NHPP) model is described in this paper. The NHPP was chosen for three reasons: simplicity, applicability, and tool availability. The model is simple in concept: an extensive mathematical background is not required to understand the nature of the model or its assumptions; its parameters have readily understood physical interpretations; and the data required for the model is simple to collect. The model is applicable to a wide variety of software development efforts including real-time flight simulators. It is a reliability growth model. This means that as the test/debugging phases of software development discover and correct design flaws, the probability of failure decreases. Finally, while these attributes apply to several models, the NHPP was a convenient choice because a tool is available to calculate the model parameters and performance measures.[9]

The assumptions behind the NHPP model are as follows:

1) The total number of faults to be found and corrected in the software is a Poisson distributed random variable that approaches a constant (say, N) if debugging is carried out indefinitely.

2) Tests represent the environment in which the simulator will be used.

3) Each failure is caused by one fault. The number of faults detected during nonoverlapping time intervals are independent of one another. This assumption allows the words failure and fault to be used interchangeably during the rest of the discussion.

4) The expected number of failures in a small time interval is proportional to the length of the time interval multiplied by the number of faults in the software at the beginning of the time interval.

Using assumptions 1)–4), the expected number of software faults detected and corrected by some time T, $f(T)$, is given by

$$f(T) = N[1 - \exp(-\lambda T)] \tag{2}$$

where N is as described in 1), and λ is the proportionality constant from 4). λ takes on the units failures/time, and, thus, can be interpreted as the rate at which faults are removed from the system (i.e., fault reduction factor) in the reliability growth model.

Furthermore, the probability of experiencing a total of x software failures by time T is given by

$$Pr(\#\text{ failures by } T = x) = [\exp(-f(T))f(T)^x]/x! \tag{3}$$

for $x = 0, 1, 2, \ldots$. Finally, the probability that the software will be operational for at least y time units, given that the last software failure occurred at some time $T = z$, is the reliability function, $R(y)$, or

$$R(y) = Pr(Y \geq y | T = z) = \exp(-f(y)\exp(-\lambda z)) \tag{4}$$

Three methods exist for estimating the parameters N and λ in Eq. (2) given individual times of error occurrences: method of moments, least squares, and maximum likelihood (MLE). Each method is discussed by Shooman,[2] but because MLEs have the best statistical properties, Goel and Okumoto[10] have derived these estimators for N and λ by letting z_i represent the failure time of the i-th error, $(i = 1, \ldots, b)$. The MLEs are then the solutions to the following system of equations:

$$\hat{N} = b/[1 - \exp(-\hat{\lambda} z_b)] \tag{5}$$

and

$$\hat{\lambda} = [q + (\hat{N}/b) z_b \exp(-\hat{\lambda} z_b)]^{-1} \tag{6}$$

where b is the total number of errors detected, and q is the average of the failure times, that is, $q = \Sigma(z_i/b)$, $i = 1, \ldots, b$.

Establishing Software Reliability Objective

Using the previous formulation, Goel and Okumoto have proposed a method for determining when to release a software product from testing.[10] Additionally, these equations allow management to assess the current status of a software project throughout the testing and operations phases of a project by trading off reliability with the costs associated with testing and operational failures. This trade can answer the two questions: 1) What reliability objective will minimize the simulator's life-cycle cost? and 2) How much testing is required to ensure this objective is met?

A reliability objective is normally established during the requirements definition or project planning phases of the system development effort. Unfortunately, unless a project manager has previously collected software reliability data on a similar project, it is difficult to initialize the model. This is a classic chicken-and-egg problem. Failure data is required to initialize the model and generate requirements; however, without requirements there is no justification to gather the data. The way out of this dilemma is to estimate values for the required parameters using data from other projects or accepted rules of thumb.

Table 2 contains estimates of N and λ for four projects, along with the size of the project (in source lines of code), and the number of errors recorded during system test. The data are from various sources,[11–14] and while only project C is a flight simulator, projects B and D are avionics systems. Project A is a military command and control system. Note that the data from project C was collected during operations and, hence, may not accurately reflect testing-phase parameter estimates, because much of the software was very mature.

An alternative to estimating values based on Table 2 is to use the rule of thumb suggested by Hecht.[15] That is, N equals 2% of the source lines of code and λ is initialized to 0.10. These seem to be conservative planning estimates based on the data in Table 2.

Attempts to predict software reliability prior to the start of integration testing are still considered preliminary.[16,17] Consequently, any costs incurred in the development phase prior to testing are considered fixed and are not changeable by altering the required reliability. Furthermore the software is considered released at the end of testing and the release data denotes the beginning of operations.

Given these considerations, let

C_1 = cost of fixing a fault during the test phase;

C_2 = cost of a failure during operations;

C_3 = cost of testing per unit time;

t = expected software operational lifetime;

T = software execution time; and

$f(T)$ = expected number of faults detected and corrected by time T.

With these variables, the expected cost of fixing faults during test is $C_1 f(T)$, and the expected cost of failures during

Table 2 NHPP parameter values for four data sets

Data set	N	λ	Size (KSLOC)	Failure sample	Test days
A	130	0.042	180	101	364
B	45	0.460	120	40	unknown
C	32	0.004	1100	24	58
D	1348	0.124	unknown	1138	100

Table 3 Parameter estimates used to establish a software reliability objective

Parameter	Estimate
Size (KSLOC)	500
N	250
λ	0.125
Cost of fixing a fault during test	\$2400/h
Cost of an operational failure	\$8410/h
Cost of testing	\$4495/h

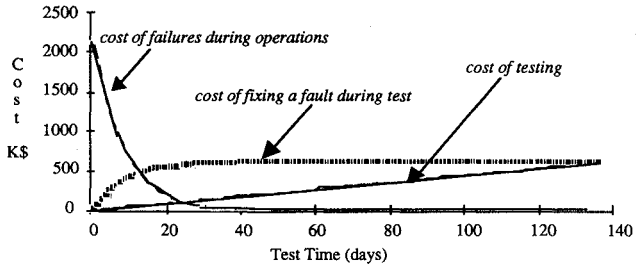

Fig. 2 Component costs of software failure by test days.

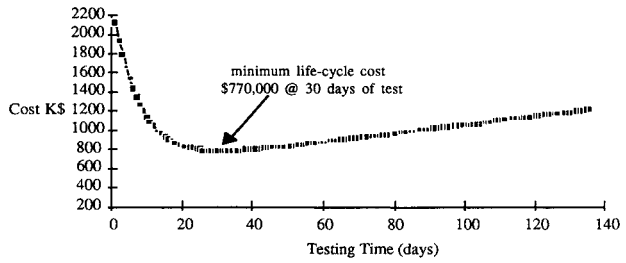

Fig. 3 Total life-cycle cost by testing time (days).

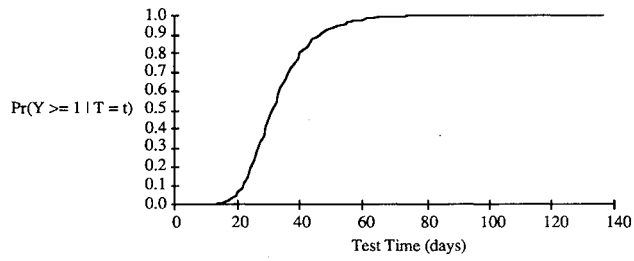

Fig. 4 Expected reliability vs test time (days).

operations is $C_2[f(t) - f(T)]$. The cost of testing is C_3T. So, the expected life-cycle cost (LCC) at any execution time is the sum of the three components, or

$$\mathrm{LCC}(T) = C_1 f(T) + C_2[f(t) - f(T)] + C_3 T \quad (7)$$

Substituting Eq. (2) for $f(T)$ and differentiating with respect to T, yields a cost minimum at

$$T = \ln[N2(C_2 - C_1)/C_3]/\lambda \quad (8)$$

This methodology is illustrated using the values shown in Table 3. Figures 2 through 4 result after substituting these values into Eqs. (7) and (8).

Figure 2 shows the curves associated with each component of the life-cycle cost equation. The cost of failures during operations decreases quickly as testing execution time increases. It becomes zero when all faults are removed from the system. This cost decrease in operations is offset, however, by an increase in the cost of debugging during test and the linear increase in the cost to test. The cost of fixing a failure during test reaches its maximum at N^*C_1, when all faults have been detected and corrected. The cost of testing continues to rise even if there are no more faults in the code, because testers, trying to break the simulator, are still using system resources at a constant rate.

From Fig. 3, if the software is released to operations with no testing at all, the total cost will be approximately \$2.1 million, and (from Fig. 4) the probability that the software operates without failure for 1 day is zero. By combining data illustrated in Figs. 3 and 4, the solution that minimizes life-cycle cost is $R(1 \text{ day}) = 0.50$, at a total cost of \$766,000. It is this reliability value that should be specified in the requirements if the customer is interested in the probability of successfully completing one training session of length 1 day [$y = 1$ in Eq. (4)]. Notice also from Fig. 3. that the cost penalty for overtesting is not as severe as for undertesting. For example, undertesting by 10 days costs \$43,150 more than the optimum; whereas overtesting by 10 days costs only \$19,600 more. This indicates that software managers should test longer, when possible, to minimize the risk of not meeting the objective due to the variance associated with parameter estimation. The software should continue to be debugged after release, resulting in a higher reliability product in the future.

Effects of Software Reliability Measurement on Testing Process

After an objective is established and agreed upon by the developer and the user, verification that the stimulator meets that level of reliability must occur. This verification takes place during the software testing process. Software testing is normally conducted in three stages:

1) *Unit testing* is executed by the development programmer on his or her particular subsystem. It is traditionally done in a development facility using software drivers to generate inputs to the subsystem rather than in an operational environment with the actual hardware or other software subsystems providing the inputs. Faults discovered during this stage are generally not recorded or tracked.

2) *Integration testing* is performed by a team of development programmers and test engineers. During this stage subsystems are combined through a step-by-step build up of the simulator. As it is integrated, each newly added subsystem is tested in conjunction with the simulator hardware and previously-combined subsystems. Faults are usually recorded and tracked during this stage, but the requirements for using current software reliability measurement models (i.e., tests are not executed in a manner similar to the operational environment using a complete system) are not met by the nature of the integration.

3) *System testing* is executed by the development test team (sometimes in conjunction with the customer) to verify that the simulator meets the functional requirements specification. Faults are recorded and tracked. The testing is generally done by flying the simulator in its intended operational environmenta. Thus, it is during this phase of testing that current software reliability models have the most applicability.

Using software reliability measurement requires two important modifications to the system testing effort. The first is a change in the collection of data during testing. The simulator execution time between failures, rather than the number of successful tests, is required to estimate the model parameters. These data can be measured directly with a hardware monitor attached to the processors running the simulation or by a software monitor embedded in the code. When neither of these options is available, the time can be estimated based on the number of interrupts handled during some known cpu-intensive time period or by using a discrete-event simulation of the system. Second, test cases should be selected at random from the input space in accordance with a specified training scenario. That is, instead of concentrating testing on one function at a time, the cases should be selected at random from all of the functions; the most used functions being assigned higher frequencies of execution during testing.

Summary

This paper has provided an overview of the major steps required to establish a software reliability program for training flight simulation software. The process starts with the definition of software failure. Given the definition, the paper describes a method for determining the direct cost of software failures. Actual failure cost data from the Shuttle Mission Training Facility were presented as an example. The next step in the process is to choose and initialize a model to specify the reliability objective of the software. The NHPP model was described and two methods for initializing it were presented. Given the specified objective, the direct cost of operating the simulation software and the amount of system testing required to deliver the simulator can be estimated. The final step is to verify that the objective has been met. This was described in the paper along with an overview of the measurement requirements and their impact upon the testing philosophy.

The cost of software failures, both direct and indirect, is the largest cost of operating a training flight simulator today. This makes specification and verification of software reliability a primary concern of simulation software users. Contracts for flight simulators should specify a reliability objective that minimizes the life-cycle cost of the software, unless safety concerns drive the user to a more expensive, higher reliability requirement.

Acknowledgments

This paper was patterned after a talk presented by A. F. Ackerman at the AIAA Space Based Observation Systems: Committee On Standards in November 1989. Further thanks are due to A. Hajare, T. Featherston, and the other MITRE employees for their insightful comments on earlier drafts of this paper.

References

[1]Federal Aviation Administration, "Airplane Simulator and Visual System Evaluation," Advisory Circular 120-40A, July 1986.

[2]Shooman, M. L., *Software Engineering: Design/Reliability/Management*, McGraw-Hill, New York, 1983.

[3]American National Standards Institute/Institute of Electrical and Electronics Engineers Std 729-1983, "Glossary of Software Engineering Terminology," *Software Engineering Standards*, 2nd ed. Institute of Electrical and Electronics Engineers, 1989, pp. 1–38.

[4]Beaty, W. K., personal communication, MITRE Corp., April 1990.

[5]Stark, G. E., "Dependability Evaluation of Integrated Hardware/Software Systems," *Institute of Electrical and Electronics Engineers Transactions on Reliability*, Vol. R-36, Oct. 1987, pp. 440–444.

[6]Farmer, W. M., Johnson, D. M., and Thayer, F. J, "Towards a Discipline for Developing Verified Software," MITRE Corp., Bedford, MA, MTP-261, Aug. 1986.

[7]Farr, W. F., "A Survey of Software Reliability Modeling and Estimation," Naval Surface Weapons Center, Dahlgren, VA, NSWC-TR 82-171, Sept. 1983.

[8]Stark, G. E., "Software Reliability and Its Application to Future NASA Projects: A Review of the Literature," NASA, JSC 22194, June 1986.

[9]Vienneau, R., "User's Guide to a Computerized Implementation of the Goel-Okumoto Non-homogeneous Poisson Process Software Reliability Model," Illinois Institute of Technology Research Institute, Data Analysis Center for Software (DACS), Rome, NY, 90-0027, Nov. 1987.

[10]Goel, A. L., and Okumoto, K., "Time-Dependent Error-Detection Rate Model for Software Reliability and Other Performance Measures," *Institute of Electrical and Electronic Engineers Transactions on Reliability*, Vol. R-28, No. 3, Aug. 1979, pp. 206–211.

[11]Musa, J. D., "Software Reliability Data," Bell Telephone Laboratories, Data Analysis Center for Software (DACS), Rome, NY, Jan. 1980.

[12]Schafer, R. E., Alter, J. F., Angus, J. E., and Emoto, S. E., "Validation of Software Reliability Models," Rome Air Development Corp., RADC-TR-79-147, June 1979.

[13]Stark, G. E., and Shooman, M. L., "A Comparison of Software Reliability Models Based on Field Data," Operations Research Society of America/Technical Information Management Society Joint National Conf., Miami, FL. 1986.

[14]Moek, G., "Comparison of Some Software Reliability Models for Simulated and Real Failure Data," *International Journal of Modelling and Simulation*, Vol. 4, No. 1, 1984, pp. 29–41.

[15]Hecht, H., and Hecht, M., "Software Reliability in the System Context," *Institute of Electrical and Electronics Engineers Transactions on Software Engineering*, Vol. SE-12, No. 1, 1986, pp. 51–58.

[16]Henry, S., Kafura, D., Mayo, K., Yerneni, A., and Wake, S., "A Reliability Model Incorporating Software Quality Factors," *Proceedings of the Annual National Joint Conference on Software Quality and Reliability*, March 1–3, 1988, pp. 340–352.

[17]Soistman, E. C., and Ragsdale, K. B., "Impact of Hardware/Software Faults on System Reliability—Study Results," Rome Air Development Center, RADC-TR-85-228, Dec. 1985.