Exposing Software Field Failures

Michael F. Siok and Clinton J. Whittaker
Lockheed Martin Aeronautics Co.

Dr. Jeff (Jianhui) Tian
Southern Methodist University

Software Reliability Engineering (SRE) provides a way to quantify the likelihood of software failures in software-intensive systems during test and in-field operations. One simple-to-use, yet practical technique uses software reliability growth models, field and laboratory usage models, and acceleration factors to estimate software field failure rates. A case study illustrates the technique for fighter aircraft avionics software and compares predicted to observed software field failures. Establishing a target software reliability objective during software development and working toward it provides assurance that the software is achieving an acceptable operational performance mark – a mark that can be predicted, observed, and measured both in the laboratory and in the field.

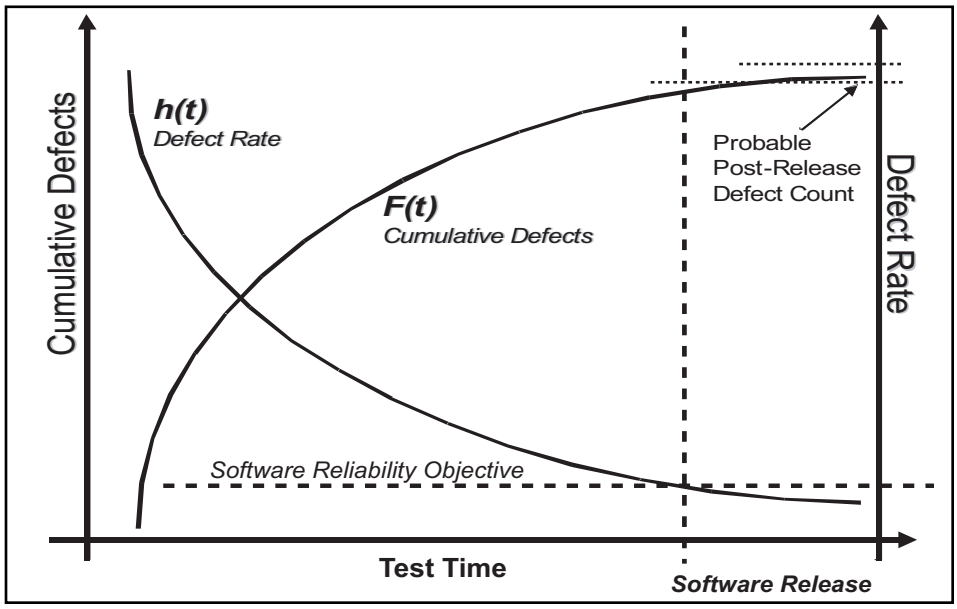
Growing the quality of software during a fighter aircraft avionics software development project is a natural outcome of conducting a disciplined software development effort. Through each step of the process, the software product evolves toward its end-product state, implementing more capabilities as it progresses toward its scheduled release date. Through a comprehensive and directed test regimen, the risk of releasing serious defects in the delivered software diminishes greatly as testing progresses, demonstrating software reliability *growth*. Once software development is complete and aircraft are being delivered, on-site operations teams manage the day-to-day operations of the deployed aircraft. Any malfunctioning avionic computers (e.g., computer will not boot, improper return to operation following a reboot, selected avionics operations in obvious violation of functional specification) may be managed through maintenance event actions by either replacing faulty equipment or reprogramming the computer and verifying a return to normal operation. These *software* maintenance events, however, have a cost – they take a fighter aircraft out of active service for the duration of the maintenance action. While software may not be the root cause of all such software maintenance events, if it were responsible for some, then perhaps models could be developed to predict their frequency. With these models, the software organization could then drive their software development efforts toward a specifiable and measurable operational quality goal and later observe the results of that effort in the field.

This article describes a practical and tested approach to using SRE time-based software reliability growth models to relate the release quality of fighter aircraft avionics software to the occurrence of software maintenance events in field operations. Using this approach, the frequency of expected software maintenance events can be estimated before the software is deployed to the field. This provides the software development organization with a key software quality metric, one with a customer-oriented view of the *operational performance* of the avionics software – a mean time to next *software* maintenance event.

Software Reliability Growth Models

Software is nothing more than a set of instructions and data derived from a software design used to control the operation of a computer system. Once software is written, it is tested in stages (e.g., unit, component, system) to identify and remove defects [1]. All contracted capabilities in software are tested and verified prior to delivery. However, it is nearly impossible with large embedded fighter aircraft real-time computer systems to completely and economically test the software in all possible operational conditions under all possible workloads. A comprehensive and successful test program will, however, sufficiently exercise the system, providing assurance that the risk of a serious computer system failure due to software will be acceptably low [2]. Assuming that the software organization manages their software processes in keeping with or bettering historical performance and assuming that these processes are demonstrably in control, then software reliability growth can be assessed using software reliability growth models (SRGMs). An SRGM is a mathematical expression of software reliability growth based on, typically, the detection rate of software failures during test. A software failure in this case is the manifestation of a software fault (i.e., a latent defect in the code) during execution of the software, creating an anomalous behavior that would be evident to the system user [1, 2, 3]. Numerous SRGMs exist in the literature; standards and

Figure 1: *Example SRGM*



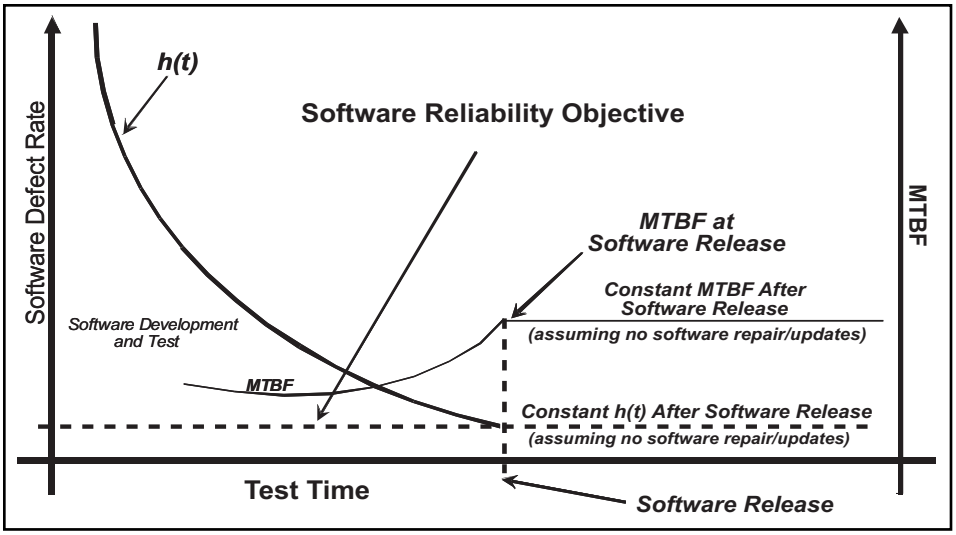


Figure 2: *Example Software Quality Goal Using SRE*

guidance on development and application of SRGMs, as well as other software reliability methods, are also widely available [2, 3, 4, 5, 6, 7]. While it is not necessarily a simple task, the software analyst chooses an SRGM that best models the defect discovery process on the software project while not creating an additional undue data collection and analysis burden on the software development staff [5, 7].

In its simplest form, an SRGM represents software reliability growth as a plot of defect arrivals per unit of measure (i.e., test time in this case) fitted to a mathematical model of the same. Defect counts may or may not be grouped. Software defects fall into the usual severity classes as their occurrences during test are visible by a system user and are recorded. Trivial, duplicate, and document-only defects are not counted.

Figure 1 (see page 15) provides an example SRGM; it illustrates the cumulative defects *F(t)* and Defect discovery rate *h(t)* curves. These curves show the expected arrival times and arrival rates of software faults discovered during test. Software reliability growth is also evident in this example – the rapid increase of defects discovered over time eventually starts to slow while testing continues at the same test intensity. By establishing a defect rate objective for software delivery and by planning the software development to achieve that objective, the software organization can estimate the total test time needed to reach that objective, estimate the probable release date of the software based on achieving that objective, and estimate the number of probable remaining defects in the software at release.

Software reliability objectives can be set early in the software development project so that software reliability growth can be modeled and tracked throughout the development effort. An example objective may be *no known severity 1, 2, or 3 defects at delivery*. This objective is quantified, coordinated, and is then, by declaration, an acceptable software defect rate at release. Figure 2 illustrates a software development effort progressing toward a software reliability/quality objective. A software mean time between failure (MTBF) metric may be computed and used as the measure of software release quality. *Failure*, in this case, means the occurrence of the next severity, 1, 2, or 3 defect[1]. This software MTBF measure is compatible with other system reliability measures used on the project outside the software organization.

Once released and deployed, software exhibits a constant failure rate (i.e., no more software reliability growth due to code corrections and re-release. For this discussion, assume no field updates for software). Since software is not updated in the field like it was during laboratory test, the software organization can easily render a prediction of the expected software field failure rate based on testing experience in the laboratory and expected usage in the field. However, field usage rates may vary significantly from laboratory usage rates causing observed field failure occurrences to differ noticeably from predictions. To render a useful prediction, software field usage needs to be quantified, compared to laboratory usage, and an acceleration factor must be derived.

Relating Laboratory to Field Experience

Software defect discovery rates at release in the laboratory are expected to be different from field-reported software defect discovery rates due primarily to the differing usage rates of the software in the various operational environments. Reliability engineers are aware of this phenomenon when working with electronic equipment, and they use an acceleration factor to adjust observed laboratory reliability performance with expected field reliability performance [8, 9]. The same technique can be used for software. The acceleration factor is simply a ratio of the laboratory usage rate (LUR) to the customer field usage rate [8]. LUR is the average usage rate of the software while it undergoes testing. LUR is determined by summing the software test time for each test system used during the test phases of the project and dividing by the average number of test systems used concurrently over the entire test period. The resulting LUR provides a usage rate that is similar to running the software in its various operational environments in x number of systems, where x is the

Table 1: *Example Software Field Usage and Acceleration Factor*

Flight Month	(Flight Hours) FH	Number of Aircraft	FI	LUR	Acceleration Factor
1	227.6	13	17.50	90	5.14
2	314.3	13	24.17	90	3.72
3	521.4	16	32.59	90	2.76
4	550.3	16	34.39	90	2.62
5	591.4	16	36.96	90	2.43
6	652.0	16	40.75	90	2.21
7	590.7	19	31.09	90	2.89
.	.	.	.	.	.
.	.	.	.	.	.

number of test systems instead of fielded aircraft.

To determine customer field usage rates, consider that deployed software may be installed on many fighter aircraft systems that may be operated within the same time periods but at different rates. Variable customer software usage must be counted for each aircraft. Flight intensity (FI) is used to establish an average customer usage rate for a group of aircraft. FI is determined by taking total flight hours (FH) and dividing by the number of in-service aircraft for each time period. The acceleration factor is then simply, LUR/FI. Table 1 illustrates how flight intensity and the acceleration factor are computed.

Estimating Software Maintenance Events

Estimating the expected number of software maintenance events is now a matter of completing a few simple computations. Starting with the software release quality expressed as MTBF, multiply by the acceleration factor to get a field-adjusted MTBF. Then, divide the expected total flight hours for that time period by the field-adjusted MTBF to get the expected number of software maintenance events for that time period. (Round the answer to the nearest whole number.) Table 2 illustrates how to estimate software maintenance events using software release quality, acceleration factor, and FH.

Process Summary

Figure 3 provides an overview of the process to compute the expected software maintenance events just discussed. To compute the software maintenance event rate, enact the following steps:

1. Determine the release quality of the software by reviewing software development historical records and modeling/analyzing software reliability growth. Determine also the LUR of the software; use models in absence of recorded data.
2. Quantify the expected customer field usage of the software. Determine FI using the number of in-service aircraft and FH. Use models in absence of recorded data.
3. From the customer and LUR, compute an acceleration factor to relate the laboratory software reliability experience with the expected field usage software reliability.
4. Compute the number of software maintenance events using software release quality, the acceleration factor, and FH. Identify and count maintenance events observed in the field that exhibit the characteristics of a software failure. Compare the observed software maintenance event data to predictions.

Flight Month	Software Release Quality ($MTBF_{SW}$)	Acceleration Factor	Field-Adjusted $MTBF_{SW}$	FH	Number of Expected ME (FH/Field Adj. $MTBF_{SW}$)
1	100	5.14	514.2	227.6	0
2	100	3.72	372.4	314.2	1
3	100	2.76	276.2	521.4	2
4	100	2.62	261.7	550.3	2
.	.	.	.	.	.
.	.	.	.	.	.

Table 2: *Computing Software Maintenance Event (ME) Estimates*

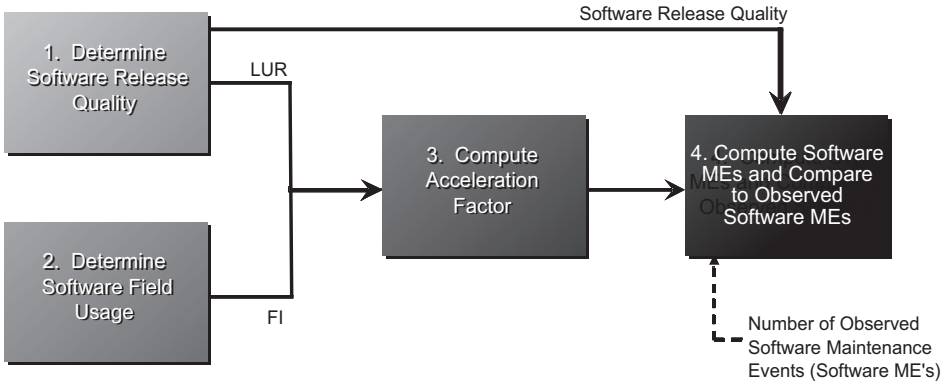
The following case study provides some practical experience using this technique.

Case Study

As part of an engineering study, the Lockheed Martin Aeronautics Company software engineering organization teamed with the system reliability engineering organization to determine if some aircraft field maintenance events reported with a particular aircraft avionics configuration could, in fact, be software or software-related maintenance events. Further, if they were, could their occurrence frequency be predicted? Recall that a maintenance event is a request for maintenance action on an aircraft due to an observed operational anomaly. To service the maintenance request, the aircraft is temporarily removed from active service. Once serviced, a description of the observed anomaly as well as the corrective action performed is noted on a maintenance event report and the aircraft is returned to active service. For this study, one embedded computer of a specific fighter aircraft avionics system configuration was identified for investigation.

Maintenance events are coded to identify an affected part needing maintenance action. *Software* maintenance events had no assigned maintenance codes. Suspected software maintenance events, however, were usually assigned to the *system* category against the embedded computer system serviced. Unknown maintenance event causes as well as a few other maintenance event *actions* were also assigned to this category. Software maintenance events and their subsequent actions had to be identified by reviewing every *system-coded* maintenance event record charged against the specific embedded computer. Selected other *non-system-coded* maintenance event reports were also reviewed looking for likely software maintenance events that might have been coded differently. Probable software maintenance requests and actions (e.g., software will not boot, improper avionics software operation after reboot, selected operations in violation of specification, computer reprogrammed and checked out OK) were identified and counted. From this work, the number of software maintenance actions per month was collected along with the number of aircraft and flight hours logged during the first 18 months of field operations of this specific

Figure 3: *Process to Compute Software MEs*



Flight Month	Software Release Quality (MTBFsw)	LUR	FI	Acceleration Factor (LUR/FI)	Field-Adjusted MTBFsw	FH	Number of Predicted Software ME	Number of Observed Software ME
1	100	90	17.50	5.14	514.2	227.6	0	0
2	100	90	24.17	3.72	372.4	314.2	1	0
3	100	90	32.59	2.76	276.2	521.4	2	1
4	100	90	34.39	2.62	261.7	550.3	2	1
5	100	90	36.96	2.43	243.5	591.4	2	2
6	100	90	40.75	2.21	220.9	652.0	3	2
7	100	90	31.09	2.89	289.5	590.7	2	2
8	100	90	33.24	2.71	270.7	638.2	2	2
9	100	90	33.73	2.67	266.8	683.7	3	3
10	100	90	26.75	3.36	336.5	699.0	2	2
11	100	90	14.87	6.05	605.0	491.9	1	1
12	100	90	15.17	5.93	593.3	703.9	1	2
13	100	90	14.68	6.13	612.9	689.2	1	2
14	100	90	18.47	4.87	487.2	867.1	2	3
15	100	90	17.49	5.15	514.7	830.0	2	2
16	100	90	12.27	7.33	733.2	582.6	1	1
17	100	90	12.85	7.00	700.4	616.8	1	1
18	100	90	15.12	5.95	595.3	725.6	1	2

Table 3: *Project Data*

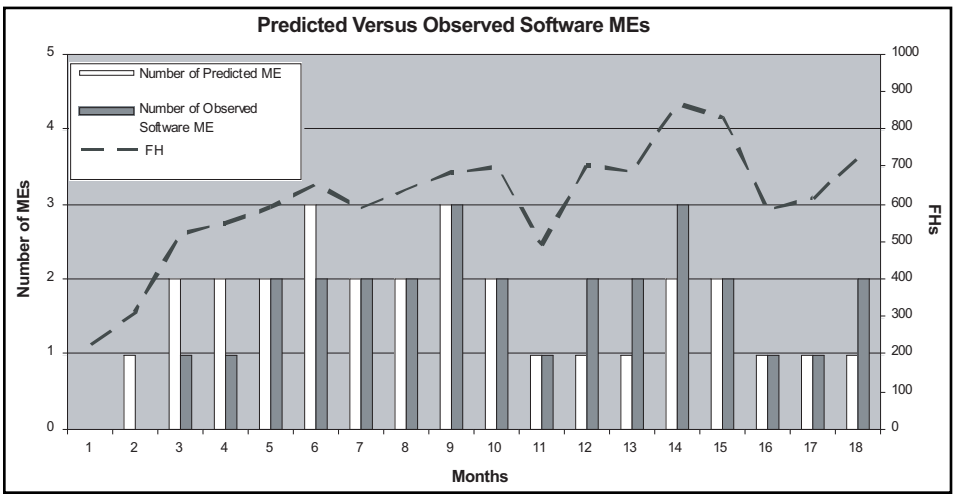
avionics configuration.

To establish the release quality of the avionics software, the management team on the project that initially developed and delivered the software was consulted on details of the software development activities. Software project metrics used during the development effort were reviewed to gain an understanding of the software process performance as well as the project dynamics at the time. From interviews, archived project data, and results of similar discussions on other similar in-house avionics software projects, an SRGM was selected, a LUR was estimated, and the software quality at release was computed and expressed as an MTBF.

The data for this engineering study from both the system reliability and software engineering groups was collected and organized into a spreadsheet. The software release quality, LUR, FI, acceleration factor, field-adjusted software MTBF, FH, and the number of predicted software maintenance events, rounded to the nearest whole number, were put into this spreadsheet. A last column identified the number of probable software maintenance actions observed in the data set. This data is presented in Table 3; the data is derived from the actual data taken from the study but has been altered to disguise proprietary information.

The number of predicted and observed software maintenance events for each month is plotted in Figure 4. FH is also plotted on the same graph. It was expected that the number of software maintenance events would follow the flight hours; that is, as flight hours increased or decreased for each month, so too did the number of software maintenance events. The observed software maintenance events exhibited this relationship with some small variation. In comparing the number of predicted to observed software maintenance events, the prediction tended to be a little pessimistic early in the field program, while later the prediction turned perhaps a little optimistic. This method models an effect generally observed in software reliability, where early in the field program when new software is released, field failure reports are initially high. As more systems are deployed with the same software, the number of failure reports per accumulated system time reduces to a near steady rate [3]. While not a perfect predictor of software maintenance events, this method does provide a useful approximation of the expected software field performance.

Figure 4: *Predicted Versus Observed Software ME Counts*



Some variations were expected in predicted and observed software maintenance event counts in this study. This variation could be attributable to the estimates of software MTBF, LUR, and FI used, possible errors in identifying appropriate maintenance event records for the study, errors in maintenance event record documentation, or record-keeping itself.

The study results indicated that, on average, the longer that the software was being run in this group of aircraft, the more likely it was that a software maintenance event would occur. It would seem to follow then that if one could predict the occurrence of these software maintenance events, one could also fix the software so that these events would not recur. Discovery of software maintenance event causes in the field is not usually easy, however. All the easy and nearly all of the *hard-to-find* software faults were found and fixed during the laboratory and flight-test program. Further, since maintenance event

reports are often short on problem descriptions and aircraft are not usually instrumented to collect data with the maintenance action request, the software maintenance events identified may not all *undeniably* be *software* maintenance events. Recovery and study of fault logs provide some help with fault cause identification, but require expert analyses. Without instrumentation data and/or complete descriptive information leading up to the point when the observed failure occurred, any likely future fix in software or the software development process is remote unless this same anomalous behavior can be repeated reliably in the laboratory or can be discovered in the laboratory on future versions of that same software.

To determine if these study results were coincidental, this same method was used on two other embedded avionic computer systems on two different fielded fighter aircraft programs. Not surprisingly, the results were similar. While results of three studies hardly prove method validity, results associated with these applications indicated that further use of the technique in-house was warranted and encouraged. Since this study was completed, SRE methods have been and continue to be applied on new start software development efforts and on additional completed and near-completed projects in a number of software application domains within the company. These SRE measures are being used in concert with other more traditional software metrics to estimate and predict the overall quality of the company-developed and fielded software.

A Note on Software Failure

When software fails to deliver its required service, the service is said to have failed. Software itself does not fail. Failures attributable to software generally occur because of requirements, design, or programming errors or they occur as a result of a computer equipment malfunctions. The software maintenance events recorded in this study could be attributed to any of these error types or computer equipment malfunctions. They likely resulted from any one or more of the following causes:

1. A specific rare event occurring in the operational environment uncovered a latent defect in the software.
2. A specific but rare use of the software not completely tested in the laboratory uncovered a latent defect in the software.
3. A specific unforeseen new use of the software not specifically provided for or tested caused an unexpected response from the software.
4. An unexpected rare event occurring in the operational environment caused an unexpected response from the software.
5. An equipment anomaly that the software was not designed to handle was encountered causing an unexpected response from the software.
6. Corrupted data caused anomalous operation.

Wrap-Up

With software size for fighter aircraft embedded real-time systems growing into the millions of source lines of code, implementing thousands of capabilities required to be operational under all workload scenarios, it is becoming a daunting challenge to economically assemble, debug, and verify this software within the complete range of system operations using current methods. Further, since software plays such a major role in providing the fighter aircraft's total system capabilities, the fighter's overall reliability is now being viewed as a function of the reliability of the hardware *and* the software [10]. Using the SRE methods described in this article to model the reliability growth of software provides a fairly simple-to-use and useful quality measure that can be used during software development as a predictor and estimator of software operational quality.

The SRE techniques described in this article provide a practical use of time-based SRE measures and metrics in identifying software product operational quality and in confirming that quality in field operations. It is a low-cost metric; it uses software fault and field failure data readily available to the engineering staff. In introducing this technique to our software engineering staff, no additional costs were incurred for data collection (i.e., required information was already being collected) and only a small cost was incurred for staff training and metrics deployment (i.e., about one person-month to develop training material and deliver instruction to a target audience of about 20 engineers). Our study results indicated that use of these few simple but tested SRE techniques within our current family of software measures had value as software operational performance predictors. Since this study, SRGMs have been studied and applied in-house on a number of fighter aircraft software programs. In the longer term, these measures are proving their worth and earning their way into our company's standard software practice.◆

References

1. Tian, Jeff. Software Quality Engineering: Testing, Quality Assurance, and Quantifiable Improvement. Hoboken, NJ: John Wiley & Sons, Inc., 2005.
2. Musa, John. "Software Reliability Engineering: More Reliable Software, Faster Development and Testing." Software Reliability Engineering and Testing Courses. McGraw-Hill, 1998.
3. Lyu, Michael R. Handbook of Software Reliability Engineering New York, NY: McGraw-Hill Companies, 1996.
4. American National Standards Institute (ANSI). "Recommended Practice for Software Reliability." ANSI R-013-1992. Feb. 1993.
5. Carnes, Patrick. Software Reliability in Weapon Systems. Proc. of the Eighth International Symposium on Software Reliability Engineering – Case Studies, 2-5 Nov. 1997.
6. Schniedewind, Norman F. "A Recommended Practice for Software Reliability." CROSSTALK Aug. 2004 <www.stsc.hill.af.mil/crosstalk/2004/8>.
7. Wallace, Delores R. Practical Software Reliability Modeling. Proc. of the 26th Annual NASA Goddard Software Engineering Workshop, Nov. 2001.
8. O'Conner, Patrick D.T. Practical Reliability Engineering. 3rd ed. West Sussex, UK: John Wiley & Sons Ltd., 1991.
9. Ireson, Grant W., Clyde F. Coombs, and Richard Y. Moss. Handbook of Reliability Engineering and Management. 2nd ed. New York, NY: McGraw-Hill Companies, 1996.
10. Hamner, Robert S., "Software Reliability?" Transactions of the ASME Journal of Electronic Packaging 122. 4 (Dec. 2000): 357-60.

Note

1. The defect severity code identifies the consequence of the defect occurring in the released software. Severity 1 – Mission cannot be accomplished. Severity 2 – Task within the mission cannot be accomplished. Severity 3 – Task can be accomplished using a work-around.

About the Authors

Michael F. Siok is a software engineer for Lockheed Martin Aeronautics Company in Fort Worth, Texas. He has been with the company for 20 years, most recently performing software project planning and applying software reliability engineering techniques. Siok is a member of the Institute of Electrical and Electronics Engineers, the Association for Computing Machinery, and the International Council of Systems Engineering, and is a registered professional engineer in Texas. He has a Bachelor of Engineering Technology in electronics from Southwest State University in Marshall, MN; a master's degree in engineering management from Southern Methodist University (SMU) in Dallas, Texas; and is currently pursuing a doctorate in engineering management at SMU.

Lockheed Martin Aeronautics Co.
P.O. Box 748, MZ 8604
Fort Worth, TX 76101
Phone: (817) 935-4514
Fax: (817) 762-9428
E-mail: mike.f.siok@lmco.com

Clinton J. Whittaker has 18 years of professional experience as a reliability/avionics engineer. Currently, he is the reliability engineering lead at Lockheed Martin Aeronautics in Fort Worth, Texas. Since joining Lockheed Martin in 2002, Whittaker has provided reliability oversight of selected aircraft programs primarily focusing on avionics hardware and software development and test. His background includes reliability and avionics experience with Alcatel in telecommunication networks; Beal Aerospace, as the avionics lead over the hardware and software development for a commercial rocket venture; and Texas Instruments/Raytheon, as a production and field reliability engineer on two missile programs. Whittaker has a bachelor's degree in electrical engineering from Texas A&M University.

Lockheed Martin Aeronautics Co.
P.O. Box 748, MZ 2462
Fort Worth, TX 76101
Phone: (817) 762-3034
Fax: (817) 655-7181
E-mail: clinton.j.whittaker
@lmco.com

Jeff (Jianhui) Tian, Ph.D., is currently with the computer science and engineering department at Southern Methodist University in Dallas, Texas, and worked for IBM Software Solutions Toronto Laboratory from 1992-1995 as a software quality and process analyst. His current research interests include software testing, measurement, reliability, safety, and complexity and application to various systems. Tian is a member of the Institute of Electrical and Electronics Engineers and the Association for Computing Machinery, and is a registered professional engineer. He has a bachelor's degree in electrical engineering from Xi'an Jiaotong University, a master's degree in engineering science from Harvard University, and a doctorate in computer science from the University of Maryland.

Southern Methodist University
Dept. of Computer Science
and Engineering
PO Box 750122
Dallas, TX 75275
Phone: (214) 768-2861
Fax: (214) 768-3085
E-mail: tian@engr.smu.edu