Successful Application of Software Reliability: A Case Study

NORMAN F. SCHNEIDEWIND

Fellow of the IEEE
2822 Raccoon Trail
Pebble Beach, California 93953 USA

(Received on July 30, 2009, revised on May 3, 2010)

Abstract: The purpose of this case study is to help readers implement or improve a software reliability program in their organizations, using a step-by-step approach based on the Institute of Electrical and Electronic Engineers (IEEE) and the American Institute of Aeronautics and Astronautics Recommended (AIAA) Practice for Software Reliability, released in June 2008, supported by a case study from the NASA Space Shuttle.

This case study covers the major phases that the software engineering practitioner needs in planning and executing a software reliability-engineering program. These phases require a number of steps for their implementation. These steps provide a structured approach to the software reliability process. Each step will be discussed to provide a good understanding of the entire software reliability process. Major topics covered are: data collection, reliability risk assessment, reliability prediction, reliability prediction interpretation, testing, reliability decisions, and lessons learned from the NASA Space Shuttle software reliability engineering program.

Keywords: software reliability program, Institute of Electrical and Electronic Engineers and the American Institute of Aeronautics and Astronautics Recommended Practice for Software Reliability, NASA Space Shuttle application

1. Introduction

The IEEE/AIAA recommended practice provides a foundation on which practitioners and researchers can build consistent methods [1]. This case study will describe the SRE process and show that it is important for an organization to have a disciplined process if it is to produce high reliability software. To accomplish this purpose, an overview is presented of existing practice in software reliability, as represented by the recommended practice [1]. This will provide the reader with the foundation to understand the basic process of Software Reliability engineering (SRE). The Space Shuttle Primary Avionics Software Subsystem will be used to illustrate the SRE process.

The reliability prediction models that will be used are based on some key definitions and assumptions, as follows:

Definitions

Interval: an integer time unit t of constant or variable length defined by t-1 < t < t+1, where t>0; failures are counted in intervals.

Number of Intervals: the number of contiguous integer time units t of constant or variable length represented by a positive real number.
**Operational Increment (OI):** a software system comprised of modules and configured from a series of builds to meet Shuttle mission functional requirements.

**Time:** continuous CPU execution time over an interval range.

**Assumptions**

1. Faults that cause failures are removed.
2. As more failures occur and more faults are corrected, remaining failures will be reduced.
3. The remaining failures are "zero" for those OI's that were executed for extremely long times (years) with no additional failure reports; correspondingly, for these OI's, maximum failures equals total observed failures.

1.1 **Space Shuttle Flight Software Application**

The Shuttle software represents a successful integration of many of the computer industry's most advanced software engineering practices and approaches. Beginning in the late 1970's, this software development and maintenance project has evolved one of the world's most mature software processes applying the principles of the highest levels of the Software Engineering Institute's (SEI) Capability Maturity Model (the software is rated Level 5 on the SEI scale) and ISO 9001 Standards [2]. This software process includes state-of-the-practice software reliability engineering (SRE) methodologies.

The goals of the recommended practice are to: interpret software reliability predictions, support verification and validation of the software, assess the risk of deploying the software, predict the reliability of the software, develop test strategies to bring the software into conformance with reliability specifications, and make reliability decisions regarding deployment of the software.

Reliability predictions are used by the developer to add confidence to a formal software certification process comprised of requirements risk analysis, design and code inspections, testing, and independent verification and validation. This case study uses the experience obtained from the application of SRE on the Shuttle project, because this application is judged by NASA and the developer to be a successful application of SRE [6]. These SRE techniques and concepts should be of value for other software systems.

1.2 **Reliability Measurements and Predictions**

There are a number of measurements and predictions that can be made of reliability to verify and validate the software. Among these are *remaining failures, maximum failures, total test time* required to attain a given *fraction of remaining failures*, and *time to next failure*. These have been shown to be useful measurements and predictions for: 1) providing confidence that the software has achieved reliability goals; 2) rationalizing how long to test a software component (e.g., testing sufficiently long to verify that the measured reliability conforms to design specifications); and 3) analyzing the risk of not achieving *remaining failures* and *time to next failure* goals [6]. Having predictions of the extent to which the software is not fault free (*remaining failures*) and whether a failure it is likely to occur during a mission (*time to next failure*) provide criteria for assessing the risk of deploying the software. Furthermore, *fraction of remaining failures* can be used as both an
operational quality goal in predicting total test time requirements and, conversely, as an indicator of operational quality as a function of total test time expended [6].

The various software reliability measurements and predictions can be divided into the following two categories to use in combination to assist in assuring the desired level of reliability of the software in mission critical systems like the Shuttle. The two categories are: 1) measurements and predictions that are associated with residual software faults and failures, and 2) measurements and predictions that are associated with the ability of the software to complete a mission without experiencing a failure of a specified severity. In the first category are: remaining failures, maximum failures, fraction of remaining failures, and total test time required to attain a given number of fraction of remaining failures. In the second category are: time to next failure and total test time required to attain a given time to next failure. In addition, there is the risk associated with not attaining the required remaining failures and time to next failure goals. Lastly, there is operational quality that is derived from fraction of remaining failures. With this type of information, a software manager can determine whether more testing is warranted or whether the software is sufficiently tested to allow its release or unrestricted use. These predictions provide a quantitative basis for achieving reliability goals [2].

1.3 Interpretations and Credibility

The two most critical factors in establishing credibility in software reliability predictions are the validation method and the way the predictions are interpreted. For example, a "conservative" prediction can be interpreted as providing an "additional margin of confidence" in the software reliability, if that predicted reliability already exceeds an established "acceptable level" or requirement. It may not be possible to validate predictions of the reliability of software precisely, but it is possible with "high confidence" to predict a lower bound on the reliability of that software within a specified environment.

If there historical failure data were available for a series of previous dates (and there is actual data for the failure history following those dates), it would be possible to compare the predictions to the actual reliability and evaluate the performance of the model. Taking this approach will significantly enhance the credibility of predictions among those who must make software deployment decisions based on the predictions [9].

1.4 Verification and Validation

Software reliability measurement and prediction are useful approaches to verify and validate software. Measurement refers to collecting and analyzing data about the observed reliability of software, for example the occurrence of failures during test. Prediction refers to using a model to forecast future software reliability, for example failure rate during operation. Measurement also provides the failure data that is used to estimate the parameters of reliability models (i.e., make the best fit of the model to the observed failure data). Once the parameters have been estimated, the model is used to predict the future reliability of the software. Verification ensures that the software product, as it exists in a given project phase, satisfies the conditions imposed in the preceding phase (e.g., reliability measurements of mission critical software components obtained during test conform to reliability specifications made during design) [5]. Validation ensures that the software product, as it exists in a given project phase, which could be the end of the project, satisfies requirements (e.g., software reliability predictions obtained during test correspond to the reliability specified in the requirements) [5].
Another way to interpret verification and validation is that it builds confidence that software is ready to be released for operational use. The release decision is crucial for systems in which software failures could endanger the safety of the mission and crew (i.e., mission critical software). To assist in making an informed decision, software risk analysis and reliability prediction are integrated and provide stopping rules for testing. This approach is applicable to all mission critical software. Improvements in the reliability of software, where the reliability measurements and predictions are directly related to mission and safety, contribute to system safety.

2. Implementing a Software Reliability Engineering Program

In broad terms, implementing a software reliability program is a two-phased process. It consists of (1) identifying the reliability goals and (2) testing the software to see whether it conforms to the goals. The reliability goals can be ideal (e.g., zero defects) but should have some basis in reality based on tradeoffs between reliability and cost. The testing phase is more complex because it involves collecting raw defect data and using it for assessment and prediction.

The following are major SRE steps in the recommended practice, keyed to the phases of the software development life cycle (not necessarily in chronological order):

2.1 State the Reliability Criteria (requirements analysis phase)

This might be stated, for example, as “no failure that would result in loss of life or mission”.

2.2 Collect Fault and Failure Data (testing and operations phase)

For each system, there should be a brief description of its purpose and functions and the fault and failure data, as shown below. Days # could be hours, minutes, as appropriate. Code the Problem Report Identification to indicate Software (S) failure, Hardware (H) failure, or People (P) failure.

- System Identification
- Purpose
- Functions
- Days # (since start of test)
- Problem Report Identification
- Problem Severity
- Failure Date
- Module with Fault
- Description of Problem

2.3 Establish Problem Severity Levels (requirements analysis phase)

Use a problem severity classification, such as the following:

3. Operator annoyance.
4. System ok, but documentation in error.
5. Error in classifying a problem (i.e., no problem existed in the first place).

Note: Not all problems result in failures.
2.4 Develop Reliability Assurance Criteria (requirements analysis phase)

Two criteria for software reliability levels will be defined. Then these criteria will be applied to the risk analysis of mission critical software. In the case of the Shuttle example, the "risk" represents the degree to which the occurrence of failures does not meet required reliability levels, regardless of how insignificant the failures may be. Although it may be counterintuitive to include minor failures in reliability assessments, in reality, doing so provides a conservative lower bound on assessment. That is, the actual reliability is highly unlikely to be lower than the assessment.

Next, a variety of equations that are used in reliability prediction and risk analysis will be defined and derived, including the relationship between time to next failure and reduction in remaining failures. Then it is shown how the prediction equations can be used to integrate testing with reliability and quality. An example is shown of how the prediction equations can be used to make decisions about whether the software is ready to deploy. Note that these equation are based on the model in [9] because this model is used on the Shuttle and is one of the models recommended in the recommended practice [1]. Other models could be used, such as those in [9].

If the reliability goal is the reduction of failures of a specified severity to an acceptable level of risk [7], then for software to be ready to deploy, after having been tested for time \( t \), it must satisfy the following criteria:

1) Predicted mean number of remaining failures \( r(t) < r_c \),
   where \( r_c \) is a specified critical value, and
2) predicted mean time to next failure \( T_{F}(t) > t_m \),
   where \( t_m \) is mission duration.

For systems that are tested and operated continuously like the Shuttle, \( t, T_{F}(t), \) and \( t_m \) are measured in execution time. Note that, as with any methodology for assuring software reliability, there is no guarantee that the expected level will be achieved. Rather, with these criteria, the objective is to reduce the risk of deploying the software to a "desired" level.

2.5 Apply the Remaining Failures Criterion (testing phase)

Criterion (1) sets the threshold on remaining failures that must be satisfied in order to deploy the software (i.e., no more than a specified number of failures).

If it is predicted that \( r(t) \geq r_c \), then the process is to continue to test for a time \( t' > t \) that is predicted to achieve \( r(t') < r_c \), using the assumptions 1 and 2 that more failures will be experienced and more faults will be corrected so that the remaining failures will be reduced by the quantity \( r(t) - r(t') \). If the developer does not have the resources to satisfy the criterion or is unable to satisfy the criterion through additional testing, the risk of deploying the software prematurely should be assessed. It is known that it is impossible to demonstrate the absence of faults [3]; however, the risk of failures occurring can be reduced to an acceptable level, as represented by \( r_c \). This scenario is shown in Figure 1. In case A, \( r(t) < r_c \) is predicted and the mission begins at \( t \). In case B, \( r(t) \geq r_c \) is predicted and the mission would be postponed until the software is tested for time \( t' \) when \( r(t') < r_c \) is predicted. In both cases criterion 2) would also be required for the mission to begin.
2.6 Apply the Time to Next Failure Criterion (testing phase)

Criterion 2 specifies that the software must survive for a time greater than the duration of the mission. If $T_F(t) \leq t_m$, is predicted, the software is tested for a time $t'$ that is predicted to achieve $T_F(t') > t_m$, using assumptions 1 and 2 that more failures will be experienced and faults corrected, so that the mean time to next failure will be increased by the quantity $T_F(t') - T_F(t)$. Again, if it is infeasible for the developer to satisfy the criterion for lack of resources or failure to achieve test objectives, the risk of deploying the software prematurely should be assessed. This scenario is shown in Figure 2.
In case A, $T_F(t) > t_m$ is predicted and the mission begins at $t$. In case B, $T_F(t) \leq t_m$ is predicted, and in this case the mission would be postponed until the software is tested for time $t'$ when $T_F(t') > t_m$ is predicted. In both cases criterion 1) would also be required for the mission to begin. If neither criterion is satisfied, the software is subjected to additional inspection and testing, to remove more faults, until the desired level of risk is achieved.

2.7 Make a Risk Assessment (pre deployment or launch phase)

Reliability Risk pertains to executing the software of a mission critical system where there is the chance of injury (e.g., astronaut injury or fatality), damage (e.g., destruction of the Shuttle), or loss (e.g., loss of the mission) if a serious software failure occurs during a mission. In the case of the Shuttle, where the occurrence of even trivial failures is rare, the fraction of those failures that pose any reliability risk is too small to be statistically significant. As a result, in order to have an adequate sample size for analysis, all failures (of any severity) over the entire 20-year life of the project have been included in the failure history database for this analysis. Therefore, the risk criterion metrics to be discussed for the Shuttle quantify the degree of risk associated with the occurrence of any software failure, no matter how insignificant it may be. As mentioned previously, this approach provides a conservative lower bound to reliability predictions.

As an example, the Schneidewind Software Reliability Model (other software reliability models could be used as well) is used to compute a parameter: fraction of remaining failures as a function of the archived failure history during test and operation [6]. The prediction methodology uses this parameter and other reliability quantities to provide bounds on total test time, remaining failures, operational quality, and time to next failure that are necessary to meet defined Shuttle software reliability levels.

The test time $t$ can be considered a measure of the degree to which software reliability goals have been achieved. This is particularly the case for systems like the Shuttle where the software is subjected to continuous and rigorous testing for several years in multiple facilities, using a variety of operational and training scenarios (e.g., by the contractor in Houston, by NASA in Houston for astronaut training, and by NASA at Cape Canaveral). In Figure 3, $t$ is interpreted as an input to a risk reduction process, and $r(t)$ and $T_F(t)$ as the outputs, with $r_c$ and $t_m$ as risk thresholds of reliability that control the process.

**Risk Criteria Levels**

![Risk Reduction Process Diagram](image)

*Figure 3: Risk Reduction Process*
While it must be recognized that test time is not the only consideration in developing test strategies and that there are other important factors, such as the consequences for reliability and cost in selecting test cases [11], nevertheless, for the foregoing reasons, test time has been found to be strongly positively correlated with reliability growth for the Shuttle [9].

2.8 Evaluate Remaining Failures Risk (pre deployment or launch phase)

To obtain the mean value of the risk criterion metric (RCM) in equation (4), first, the mean remaining failures must be predicted in equation (3).

\[ r(t) = \alpha \left( \frac{\exp(-\beta(t-(s-1)))}{\beta} \right) \]  

Equation (3) is plotted in Figure 4 as a function of \( t \) for \( r_c = 1 \), for the Shuttle software release OID, a software system comprised of modules and configured from a series of builds to meet Shuttle mission functional requirements, where positive, zero, and negative values correspond to \( r(t) > r_c \), \( r(t) = r_c \), and \( r(t) < r_c \), respectively.

In Figure 4, these values correspond to the following regions: above the X-axis predicted remaining failures are greater than the specified value; on the X-axis predicted remaining failures are equal to the specified value; and below the X-axis predicted remaining failures are less than the specified value, which could represent a "safe" threshold or in the Shuttle example, an "error-free" condition boundary. In the example it can be seen that at \( t = 80 \) the risk transitions from the high risk region to the low risk region.

**Figure 4: RCM for Remaining Failures, \( r_c = 1 \), OID**
2.9 Evaluate Time to Next Failure Risk (pre deployment or launch phase)

The mean value of the risk criterion metric (RCM) for criterion 2 is formulated as follows:

$$\text{RCM}_{TF}(t) = \frac{(t_m - T_F(t))}{t_m} = 1 - \frac{T_F(t)}{t_m}$$ (4)

Equation (4) is plotted in Figure 5 as a function of test time $t$ for $t_m = 8$ thirty day intervals, for OID, where there is high risk for $T_F(t) < t_m$. Once $T_F(t) > t_m$, the risk is low.

![Figure 5: RCM for Time to Next Failure ($t_m = 8$ days) OIC](image)

3. Make Reliability Predictions (test and operations phases)

In order to support the reliability goal and to assess the risk of deploying the software, various reliability and quality predictions are made during the test phase to validate that the software meets requirements. For example, suppose the software reliability requirements state the following: 1) ideally, after testing the software for time $t$, the mean predicted remaining failures shall be less than one; 2) if the ideal of 1) cannot be achieved due to cost and schedule constraints, mean time to next failure, predicted after testing for time $t$, shall exceed the mission duration; and 3) the risk of not meeting 1) and 2) shall be assessed.

3.1 Additional Risk Evaluation (test and operations phases)

In addition to remaining failures and time to failure risk, which have already been discussed, various other predictions are made in order to provide a comprehensive assessment of risk. These predictions are based on the Schneidewind Software Reliability Model [1, 8, 9, 10]. Again, other models recommended in the Recommended Practice for Software Reliability [1] could be used. The Statistical Modeling and Estimation of Reliability Functions for Software (SMERFS) [4] tool is used to support predictions.

In the following equations, parameter $\alpha$ is the failure rate at the beginning of interval $s$; parameter $\beta$ is the negative of the derivative of failure rate divided by failure rate (i.e., relative failure rate); $t$ is test time or the last interval of observed failure data; $s$ is the starting interval for using observed failure data in parameter estimation that provides
the best estimates of $\alpha$ and $\beta$ and the most accurate predictions [8]; $X_{s-1}$ is the observed failure count in the range $[1,s-1]$; $X_{s,t}$ is the observed failure count in the range $[s,1]$; and $X_t = X_{s-1} + X_{s,t}$. Failures are counted against operational increments (OIs).

**Cumulative Failures:** When estimates are obtained for the parameters $\alpha$ and $\beta$, with $s$ as the starting interval for using observed failure data, the predicted failure count in the range $[1,t]$ is obtained (i.e., cumulative failures) [6]:

$$F(t) = \frac{\alpha}{\beta} [1 - \exp(-\beta ((t-s+1)))] + X_{s-1} \quad (6)$$

Figure 6 provides risk reduction in the sense that the predicted cumulative failures provide an upper bound on the actual failures (i.e., there is assurance that the actual failures will not exceed the predicted values). In addition, risk is mitigated by the fact that the predictions increase at an increasing rate. Also shown in this figure is the mean relative error (MRE) between actual and predicted values. The MRE is high due to the fact that predictions are consistently higher than actual values.

**Maximum Failures:** Let $t \to \infty$ in equation (6) and obtain the predicted failure count in the range $[1, \infty]$ (i.e., maximum failures over the life of the software):

$$F(\infty) = \frac{\alpha}{\beta} + X_{s-1} \quad (7)$$

Applying equation (7), the predicted maximum failures = 18.4706. Thus, we would have low risk that the actual cumulative failures will not exceed the value.

**Fraction of Remaining Failures:** If equation (3) is divided by equation (7), *fraction of remaining failures*, predicted at time $t$ is obtained:

$$p(t) = \frac{r(t)}{r(\infty)} \quad (8)$$

According to the manager of Shuttle software development, equation (8) is an excellent management tool for providing confidence that the software is ready to deploy,
as the fraction remaining failures becomes miniscule, with increasing testing, as Figure 7 attests [5].

![Figure 7: Operational Quality (Fraction Fault Removal) vs. Total Test Time, OIA](image)

*Operational Quality:* The operational quality of software is the complement of $p(t)$. It is the degree to which software is free of remaining faults (failures), using the assumption 1 that the faults that cause failures are removed. It is predicted at time $t$ as follows:

$$Q(t) = 1 - p(t) \quad (9)$$

This risk metric is useful because some software engineers and managers would prefer to see things in a positive light -- quality growth. Figure 7 demonstrates that after $t = 100$ the improvement in quality becomes miniscule, and the cost to remove additional faults would be significant. Thus this figure metrics for risk assessment and a stopping rule for when to terminate testing.

*Total Test Time to Achieve Specified Remaining Failures.* The predicted test time required to achieve a specified number of remaining failures at $t$, $r(t)$, is obtained from equation (3) by solving for $t$:

$$t = \frac{1}{\beta}(\beta(s-1)-\log(\frac{r(t)\beta}{\alpha})) \quad (10)$$

Equation (10) is another risk reduction metric based on the concept that the predicted test time to achieve a specified number of remaining failures reveals how much test time and effort would be required to achieve various levels of risk, as represented by specified remaining failures, as shown in Figure 8, where, naturally, the test time and cost becomes significantly high in order to achieve significant reductions in risk.

### 3.2 Interpret Software Reliability Predictions (pre deployment or launch phase)
Successful use of statistical modeling in predicting the reliability of a software system requires a thorough understanding of precisely how the resulting predictions are to be interpreted and applied [9]. The Shuttle software (430 KLOC) is frequently modified, at the request of NASA, to add or change capabilities using a constantly improving process.

Each of these successive versions constitutes an upgrade to the preceding software version. Each new version of the software (designated as an Operational Increment, OI) contains software code that has been carried forward from each of the previous versions ("previous-version subset") as well as new code generated for that new version ("new-version subset"). We have found that by applying a reliability model independently to the code subsets we can obtain satisfactory composite predictions for the total version [9].

It is essential to recognize that this approach requires a very accurate code change history so that every failure can be uniquely attributed to the version in which the defective line(s) of code were first introduced. In this way, it is possible to build a separate failure history for the new code in each release. To apply SRE to a software system, it should be broken your down into smaller elements to which a reliability model can be more accurately applied. This approach has been successfully applied to predict the reliability of the Shuttle software for NASA [9].

3.3 Use Software Reliability Tools (test and operations phases)

It is infeasible to do large-scale reliability prediction by hand. Therefore, there are software reliability tools available to make the model predictions easier to achieve. The Statistical Modeling and Estimation of Reliability Functions for Software (SMERFS) is a software package available for this purpose [4]. However, it is important for the user to understand the capabilities, applicability, and limitations of such tools.

Figure 8: Launch Decision: Remaining Failures vs. Total Test Time, OIA
4. Lessons Learned

Several important lessons have been learned from the experience of twenty years in developing and maintaining the Shuttle software, which you could consider for adoption in your SRE process:

1) No one SRE process method is the "silver bullet" for achieving high reliability. Various methods, including formal inspections, failure modes analysis, verification and validation, testing, statistical process control, risk analysis, and reliability modeling and prediction must be integrated and applied.

2) The process must be continually improved and upgraded. For example, recent experiments with software metrics have demonstrated the potential of using metrics as early indicators of future reliability problems. This approach, combined with inspections, allows many reliability problems to be identified and resolved before testing.

3) The process must have feedback loops so that information about reliability problems discovered during inspection and testing is fed back not only to requirements analysis and design for the purpose of improving the reliability of future products but also to the requirements analysis, design, inspection and testing processes themselves. In other words, the feedback is designed to improve not only the product but also the processes that produce the product.

4) Given the current state-of-the-practice in software reliability modeling and prediction, practitioners should not view reliability models as having the ability to make highly accurate predictions of future software reliability. Rather, software managers should interpret these predictions in two significant ways: a) providing increased confidence, when used as part of an integrated SRE process, that the software is safe to deploy; and b) providing bounds on the reliability of the deployed software (e.g., high confidence that in operation the time to next failure will exceed the predicted value and the predicted value will safely exceed the mission duration).

5. Conclusions

We showed how software reliability predictions can increase confidence in the reliability of mission critical software such as the NASA Space Shuttle Primary Avionics Software System. These results are applicable to other mission critical software. Remaining failures, maximum failures, total test time required to attain a given fraction of remaining failures, and time to next failure were shown to be useful reliability measurements and predictions for: 1) providing confidence that the software has achieved reliability goals; 2) rationalizing how long to test a piece of software; and 3) analyzing the risk of not achieving remaining failure and time to next failure goals. Having predictions of the extent that the software is not fault free (remaining failures) and whether it is likely to survive a mission (time to next failure) provide criteria for assessing the risk of deploying the software. Furthermore, fraction of remaining failures can be used as both an operational quality goal in predicting total test time requirements and, conversely, as an indicator of operational quality as a function of total test time expended.

Software reliability engineering is a tool that software managers can use to provide confidence that the software meets reliability goals.
References


Bibliography

Successful Application of Software Reliability: Case Study


**Norman F. Schneidewind**, Ph.D., is Professor Emeritus of Information Sciences in the Department of Information Sciences and the Software Engineering Group at the Naval Postgraduate School. He is now doing research and publishing in software reliability and metrics with his consulting company Computer Research. Dr. Schneidewind is a Fellow of the IEEE, elected in 1992 “for contributions to software measurement models in reliability and metrics, and for leadership in advancing the field of software maintenance”. In 2001, he received the IEEE Reliability Engineer of the Year award from the IEEE Reliability Society. In 1993 and 1999, he received awards for Outstanding Research Achievement by the Naval Postgraduate School.

Dr. Schneidewind was selected for an IEEE USA Congressional Fellowship for 2005 and worked with the Committee on Homeland Security and Government Affairs, United States Senate, focusing on homeland security, cyber security, and privacy. In March, 2006, he received the IEEE Computer Society Outstanding Contribution Award.
for "outstanding technical and leadership contributions as the Chair of the Working Group revising IEEE Standard 982.1".

He is the developer of the Schneidewind software reliability model that was used by NASA to assist in the prediction of software reliability of the Space Shuttle, by the Naval Surface Warfare Center for Tomahawk cruise missile launch and Trident software reliability prediction, and by the Marine Corps Tactical Systems Support Activity for distributed system software reliability assessment and prediction. This model is recommended by the IEEE and the American Institute of Aeronautics and Astronautics Recommended Practice for Software Reliability. In addition, the model is implemented in the Statistical Modeling and Estimation of Reliability Functions for Software (SMERFS), software reliability-modeling tool.