Measuring Reliability of Software Products

Pankaj Jalote, Brendan Murphy, Mario Garzia, Ben Errez
Microsoft Corporation
One Redmond Way
Redmond, WA 98052
{pjalote, bmurphy, mariogar, bene}@microsoft.com

Abstract

Current methods to measure the reliability of software are usually focused on large server based products. In these approaches, the product reliability is traditionally measured in terms of catastrophic failures, as the failure data is generally collected manually through service organizations which filter out data on many types of operational failures. These method and metrics are not applicable for mass market products that run in multiple operational profiles, where other types of failures might be equally important, and where manual data collection is inadequate. For such products, unique issues arise in obtaining the failure and population data, and in analyzing this data to determine reliability. In this paper we first discuss some of the key issues in determining reliability of such software products, and then discuss two systems being used for measuring reliability of commercial software products.

1. Introduction

Knowing the desirable properties of a product in quantitative terms is an established part of the engineering activity. As reliability is one of the most desirable properties of most products in the modern world, its quantitative specification is clearly needed and desired. Though general reliability theory has been well developed for years, as the software process has some unique characteristics which do not exist for physical systems, a new set of models called the reliability growth models were proposed for estimating the reliability of software systems (for a survey of reliability models see [5,7].)

Most reliability growth models depend on one key assumption about evolution of software systems – faults are continually removed as failures are identified thereby increasing the reliability of the software. The data on failure and fixes for these models is typically obtained during the final stages of testing. The growth model is used to predict the reliability of the software system at any point in time during this failure-and-fix process. The key issue is to obtain a good model that can explain the past data and predict the future.

Once the product is released, however, we are no longer in a controlled test situation but instead are in an operational environment with many users (maybe even millions.) Consequently, faults are not necessarily removed as failures occur. Furthermore, as many installations of the software exist, it is possible to obtain sufficient failure data before any changes are made to the software to fix the faults. In other words, sufficient failure data about one particular software version can be available. Both these factors make it feasible to measure the reliability of a software system in production – something that is not practical with single-installation software and that goes beyond the test environment considered by growth models.

For measuring the reliability of a product, the main issue is that of collecting accurate and complete failure and population data that is needed for determining reliability. Often the failure data is obtained through a Product Service Organization (PSO) where users can report failures when they encounter them, and population data is obtained from the sales figures. (Examples of use of this approach are given in [1, 8, 19].) Measuring reliability this way implicitly assumes that reliability of a product is the same for all users. In addition, it also assumes that most failures are reported and that the user base is known.

This approach for measuring reliability can work for large server-type software products whose usage profile is similar, whose population data is well known, and where failures are likely to be reported due to the nature of their customer base. However, these assumptions do not hold for a mass-market product as it often has users with greatly varying operational profiles, the population data for different users groups is not easily known, and different types of users have different inclinations to failure reporting. Measuring reliability of such products raises many unique issues.

In this paper we first discuss the key issues associated with measuring the reliability of such widely used software products and then describe two measurement systems that are being used to measure reliability of commercial software products. But before we do that, let us define what we mean by reliability of a software product and how it can be computed from the failure data.

2. Product Reliability

The reliability of a system is a measure of its ability to provide a failure-free operation. For many practical situations, reliability of a system is represented as the failure rate. For measuring the failure rate of a software product, we can have N installations of the software under observation. If the total number of failures in all the N installations in a time period T is F, then the best estimate for the failure rate of the software is [18] $\lambda = F / (N * T)$. This approach for measuring failure rates has been widely used [1, 19].

Even this straightforward approach for quantifying reliability has some underlying assumptions. Some of the key assumptions in measuring reliability in this manner are:

- All failures have "equal" reliability impact, and that there is a single number that captures the reliability of the product under all usage scenarios.
- All the F failures can be recorded, and the population size N is known.
- By normalizing by T*N (and T is generally measured in days,) it is assumed that the system is in use for same amount of time each day (generally assumed to be 24 hours.)
- The operational profile is consistent across the user base.

For measuring reliability of a mass-market product, these assumptions do not hold. There are often multiple user groups who use the product in very different ways and therefore the impact of specific failures varies between the different user groups. The weight of different types of failures also changes from product to product – for example, for some products a user-interface failure is very important while for real-time applications performance failure are often far more important. The usage time of such software is generally not 24-hours a day, and users often do not report failures. The population size is also hard to obtain.

Hence, for a mass-market product, the above approach for reliability measurement has to be extended to accurately represent the reliability experience of different user scenarios. For capturing the user perception of reliability, we need to have the ability to distinguish different types of failures in reliability measurement. We view reliability of a product as a vector comprising of failure rates for different failure types. That is, the reliability of a product is:

$$\text{Product Reliability} = [\lambda_1, \lambda_2, \lambda_3, \ldots\ldots, \lambda_n]$$

Note that from this reliability vector we can get a single reliability number for a product by taking a weighted sum of the failure rates for different types of failures. The weights will represent the relative importance for the product of the different failure types. If all failures are equal, then the overall failure rate is the sum of all the failure rates. Note also that varying reliability perceptions of various user groups can be reflected by assigning suitable weights to different types of failures. The weights, however, need to remain unchanged if the evolution of reliability with time is to be studied.

This view of product reliability also provides a practical framework for improving the product reliability experience of users. Measurement in this form, along with an idea of the users needs, can help better determine the product areas that need to be strengthened for improving the users' reliability experience.

3. Measuring Reliability

Let us now discuss some of the key issues faced when measuring the reliability of a software product, using the approach discussed above.

3.1. Failure Classification

As reliability is concerned with the frequency of different types of failures we need to have a clear and unambiguous classification of failures. The failure classification scheme should be general and comprehensive and should permit a unique classification of each failure. This failure classification will have to be from the users' perspective, as we are trying to capture the reliability experience of the user. Unfortunately, though many fault classifications have been proposed in the literature (for example, see [2] and the IEEE standard [9]), there are few classifications of failures available. One classification was proposed by Cristian, which classified failures as omission, timing, and response [4]. This classification partitions the failures at an abstract level and needs to be extended to capture the users view.

For a modern software product, we suggest that failures be partitioned at the top level as *unplanned events*, *planned events*, and *configuration failures*. Unplanned events are traditional failures like crash, hang, incorrect or no output, which are caused by software bugs. Planned events are those where the software is shutdown in a planned manner to perform some housekeeping tasks. Configuration failures occur due to problems in configuration setting. In many systems, configuration failures account for a large percentage of failures [3]. We also include in this category, failures due to human errors, which are very important in many data center operations. It can be argued that planned events and configuration failures are not software failures, as there is no software fault causing them, but as they affect the user's reliability experience, we believe they should be included. Some of the examples of the types of events that can be included under these categories are given in Figure 1.

- Unplanned Events
    - Crashes
    - Hangs
    - Functionally incorrect response
    - Untimely response – too fast or slow
- Planned Events
    - Updates requiring restart
    - Configuration changes requiring a restart
- Configuration failures
    - Application/System incompatibility error
    - Installation/setup failures

Figure 1: A failure classification

This failure classification provides a framework for counting failures. Different products may choose to focus on specific types of failures only, depending on what is of importance to their users and the overhead of measurement. However, if we want to compare reliabilities of different products, it is essential to use a standard framework and that failures are counted in the same manner.

3.2. The Population Size

A key data we need for determining reliability is the population size, that is, how many units of the product are in operation. In the past sales information has often been used [1, 8, 19]. Using the sales data for mass market product poses new problems. Many product manufacturers use multiple distribution channels to sell a product. Whereas the product manufacturer typically records a sale when the product is "sold" to the channel, when the product is actually installed onto a computer by a user (by the channel) is often not known. Additionally, large organizations may buy licenses for unlimited installations, with the actual number of users not reported to the product manufacturer. Hence, getting an accurate data about the actual population of units in use is not easy. Furthermore, using the entire user population base for reliability will require obtaining failure data from this base, which will be much harder for a widely-sold mass market product.

We propose that for determining reliability a (random) sample, called the *observed group*, of the population size be taken. With this identified observed group, failure data will be recorded only for this group of users. Regular statistical techniques can be used to determine the sample size such that the final result is accurate to the degree desired.

If we fix the population size early, it allows the reliability growth with age to be tracked. It has been observed that in many cases failure rate of units decrease in the initial stages as users stabilize their configuration and learn to avoid failure causing situations. By fixing the sample relatively early, we avoid the problem of mixing failure rates of old and new units, and can easily determine the steady state reliability. Fixing a sample early also allows understanding of the impact of patches and service packs released by the product manufacturer.

3.3. Obtaining Failure Data

For reliability computation, we need a mechanism to collect failure data, where the failures are occurring on system used by users distributed around the world.

In the past, failures reported by the users to the PSO have been used [1, 8, 14, 19]. But it is well known that customers do not report all the problems they encounter as they sometimes solve it themselves. This non-reporting is far more pronounced in mass-market products. Furthermore, for a mass-market product, there may be multiple levels of PSOs – a retailer or a distribution channel may be providing a PSO or a large user organization may have its own PSO. A failure will typically be escalated to the PSO of the product manufacturer only if it cannot be addressed by other PSOs. Hence, this method of data collection, though useful for trend analysis and getting some general sense of reliability, will not lead to an accurate determination of reliability.

If data is to be reported by the user, we suggest the use of polling. In this approach, users in the observed group are periodically asked to fill a form to report failures they have experienced in the last 24 hours. If we assume that the probability of multiple failures of a type in 24 hours is minimal (a fair assumption for the widely distributed products that we are considering,) this form can be a simple, with check boxes for each failure type, and its submission can be easily automated.

The most accurate data collection for the observed group will occur if the data is collected and reported automatically through proper instrumentation and triggers in the product. An event logging mechanism provides the ability for products to record special events. Products using event logging mechanism have to be programmed to record their specific events in the log. These events will typically be based on user interactions (to capture the usage time,) and the program state and exit status (to capture failure data.) From these event logs, a product manufacturer can filter out system events of interest that are recorded by the OS itself (e.g. reboots, crashes, etc.), and the specific events their products have recorded. This subset of the event log can be used to determine the reliability and availability of the products. The level of detail possible in the reliability analysis depends on what events are being recognized and logged by the product.

Event logging has been used in operating systems to assist in the management and repair of systems and determine availability and reliability of such systems [6, 10, 15, 16]. The focus of these systems is often on system shutdowns and recovery. However, event logs have not been used much for measuring reliability of mass-market products. In such products, the type of failures that may be of interest is broader and identifying a definitive set of events to record is harder.

3.4. Usage Time

For an accurate computation of reliability, the actual usage time of the product by the user needs to be determined to be able to calculate the failure rates. As a convenience, it is often implicitly assumed that the product is used, on an average, for the same amount of time every day by every user. With this assumption, the day count can be used for determining reliability.

However, the usage duration for different users may vary considerably for mass-market products. As the failures encountered by a user clearly depend on the amount of usage of the product – the longer the usage duration the more the chances of encountering failures – to get an accurate idea of the reliability of the product in use, we need to capture the usage time. Employing usage time instead of number of days of ownership for reliability computation is similar to the calendar time vs. CPU time discussion in reliability growth models [17]. For reliability growth models, it is widely believed that using CPU time gives better reliability estimates. Note that usage time collection throws up new issues for mass market products as the use of such products is generally spread over many sessions.

3.5. Hardware/Software Configuration

For server-type software, its underlying hardware configuration is often well defined and understood. This is not so for mass market product – a product may run as a client or as a server, may run on a machine with lots of memory or a machine with little memory, a machine with network connection or without, etc. Besides the hardware configuration, the product may also co-exist with many other types of software resident on the computer including games, entertainment software, various programs downloaded from the Internet, etc. It is known that the failure rate of software often depends on the load on the hardware or the capacity of the hardware [11, 12] – for example a software is more likely to fail in a system with small memory as compared to a system with large memory. Though our reliability definition does not require information about hardware/software configuration, for improving the reliability of products, the dependency of reliability on the configuration needs to be understood and characterized. For this reason, it is highly desirable if the configuration information for users in the observed group is also collected.

4. Example Measurement Systems

Microsoft has many mass-market products. Efforts are underway at Microsoft for measuring reliability of some of these products. Here we briefly describe two such efforts, and how they address the issues mentioned above.

4.1. Office Customer Experience Improvement Program (CEIP)

One of the first systems to attempt to apply the proposed methodology was Microsoft Office Systems 2003 through its Customer Experience Improvement Program (CEIP) technology. CEIP is an elaborate, programmable, event recording system for products, which can be used to record both failure data and usage data.

To record failures of a software product through CEIP, the product has to be programmed to record events using CEIP provided APIs. Generally, for capturing failure information, three types of events are captured.

- Application termination – when an application terminates, an event is recorded. This records normal exits, crash exits, hangs, and user forced exits. Events on some exits are recorded by a handler that is executed before exiting. Some exits cannot be recorded at the time they occur; instead, they are identified and recorded when the application is restarted using various status tracking mechanisms.
- Assert failures – An application may be shipped to the customers with assert statements in its code (called *ship asserts.*) If a ship assert fails, it is treated as an event and information about the failing assert and some related state information is recorded.

- Alerts – most applications give alerts to the users when some special situations arise (e.g. file does not exist, network not available, file writing fails, etc.) Many of these alerts represent failure situations, and recording of alerts provides additional failure information.

Application termination events are used mostly to identify crashes and hangs. Assert and alert events (as well as some application termination events) are used to identify functionality failures. Some alerts identify configuration failures. Failures of each assert and alert is recorded as a separate event, and their grouping into categories is left for post-processing of the event log. How completely the different types of failures are recorded through these means depends on how well the application can identify the failures through asserts and alerts. Our experience is that most failures that users are concerned about can be identified through these means.

When the event data is collected, configuration information like the version of the underlying Operating System, patches uploaded, the version of the application, language being used, amount of memory in the system, information about the hardware, etc. is also collected. Through this data, loading of patches and other updates can be identified, providing information about "planned events". The configuration data also provides the ability to study correlations of failures with different configuration parameters.

In addition to failure data, CEIP also collects usage data using the same event logging mechanism as used for failure data. Essentially, the start and end of a session are recorded, giving the duration of each session as well as the total number of sessions for the user. Many other user actions (like giving keyboard input, moving the mouse) are also recorded. Furthermore, by polling every minute to see if the user is active, the actual time the user is interacting with the application is also recorded. The mechanism is now also being used to record the performance of applications. Rules will be built later to classify some levels of performance as constituting "untimely response" failures.

CEIP is available to a user by subscription only. That is, it is enabled only for users who subscribe to it, and data is collected only for these users. Each subscriber is assigned a unique number. The subscribers therefore form the observed group. Through this subscription process, the total population size of the observed group is known. It should be mentioned that this method of selecting the observed group is not strictly random. However, there are thousands of users who have subscribed to CEIP providing a large and wide user base for measurements.

Hence, the information from CEIP provides data on different types of failures encountered by users for the application being investigated, the usage time for the application and the number of sessions, and the total population. (CEIP provides a lot more data for understanding the user behavior as well.) From this data, failure rate for different types of failures being recorded can be determined, giving the reliability of the application.

Let's look at an example. Though data about different types of failures is recorded and their failure rates can be determined, quite frequently managers focus on crash-failure and hang-failures, as these are most disruptive for the users, and the users attach the highest weight to these failures. Given in Table 1 is part of a sample report that is generated by CEIP tools. The report gives the total number of sessions, the total session length, and the count of crash and hangs failures for some observation period. Failure rate for these two failure types is then computed from the data – it can be computed as number of failures

Product	**No of Sessions**	**No of Crash Failures**	**No of Hang Failures**	**Session Length (mts)**	**Crash Failure Rate (per hr)**	**Hang Failure Rate (per hr)**
A	33,000	300	1,000	3,140,000	0.0057	0.0191
B	422,000	1200	8,700	46,450,000	0.0015	0.0112
C	20,000	100	700	2,540,000	0.0023	0.0165
D	24,000	100	1,000	5,940,000	0.0010	0.0101
E	153,000	600	3,300	12,920,000	0.0027	0.0153
F	12,000	100	200	900,000	0.0066	0.0133
G	648,000	2600	29,900	183,530,000	0.0008	0.0097

Table 1: An example CEIP report

per session or number of failures per hour of usage. Note that the data in this table is for illustration purposes only – it has been obtained from an actual report but has been scaled to protect the confidentiality of the data.

From this data measures such as mean time to crash, or mean time to hang, could easily be calculated. Similarly, the average session length and the mean number of sessions before a crash/hang can also be determined.

How the mean time to some type of failure (or the failure rate) evolves with time can easily be determined from past data. A sample chart showing the mean time to crash (MTTC) failures for a few different products is shown below in Figure 2. The names of the products and the actual values of time has been omitted to protect the confidentiality of the data, though the trends are as obtained from the actual data.

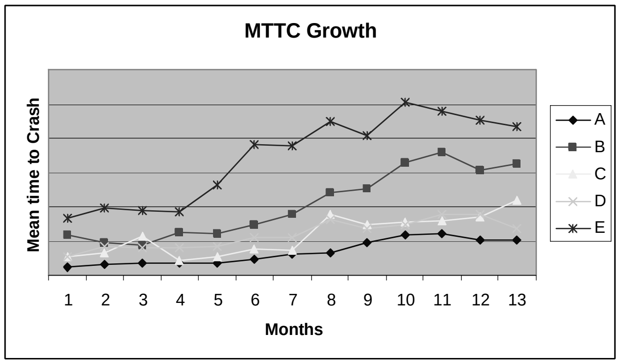

Figure 2: Mean time to crash failures

The CEIP has two parts – a client that resides in the user's machine, and a centralized server. The data recorded in the client machine is sent to the server, along with configuration data.

The event logging mechanism requires the event to specify the parameters it wants to record. These parameters include things like program counter (or offset), application name, tag of the assert or alert, unique user tag, etc. These parameters are used to define different buckets in which the events are grouped. For efficiency (millions of records are sent every day), for one bucket, data is captured only for the first few events. After that only the event count is incremented. For each new combination of parameters, a new bucket is created.

This method of bucketing allows the application developers to identify the major causes of a type of failure event. And, as is often the case, for many products the 80-20 rule was found to hold true. That is, most of the failures encountered by users are caused by a few problems. These problems then become the high priority issues to be resolved by product developers in order to give maximum reliability benefit to the users.

CEIP is being deployed by thousands of customers, providing millions of records each day. Internally at Microsoft, reports from CEIP are used to report various reliability and quality metrics. Its use has been quite effective in identifying and removing various reliability problems, including defects. Its use continues to grow. Though the current focus is on overall rates of different types of failures, in the future correlations between configurations and different failure rates may be studied, information for which is already available.

4.2. Microsoft Reliability Analysis Service (MRAS)

While CEIP is a measurement platform for products running in MS Office context, the Microsoft Reliability Analysis Service (MRAS) focuses on reliability (and availability) tracking of Windows servers, and products running on servers like MS SQL database, MS IIS web server, MS mail Exchange, and the Windows Active Directory. Another difference is that while CEIP data is reported internally at Microsoft, MRAS also provides the customer with custom reliability and availability reports for their servers. MRAS is discussed in detail in [6].

MRAS is made up of two main components – the MRAS client and the MRAS reporting site. The MRAS Client is installed on a particular server and is supplied with the set of servers it should track. It is responsible for collecting the server log data, and subsequently uploading it to the MRAS reporting site. In each collection, the client collects only new data, making frequent collections efficient. While there are hundreds of events that may be recorded in a server's log, the client only collects a relatively small number of events (approximately 100 different events).

The reporting site serves as the data warehouse and analysis component. The data from clients is loaded to the data warehouse, analyzed and stored in tables for later reporting. The reliability reports are accessed and managed through a web interface.

The server group to be monitored is specified by the user. Hence, a client knows the precise number of servers it is monitoring. It also can obtain the data about the configuration of the server and which applications are running on it. From data from clients, the reporting site knows the total number of servers being monitored, as well as the total number of the monitored applications running. In other words, it has the observed group size for the servers as well as the applications it is monitoring. (Note again that this approach of observed group selection is not completely random.)

MRAS focuses primarily on planned and unplanned events that result in loss of availability as that is of primary interest for server and server applications. Therefore, though it can be used to track any event, it primarily focuses on crashes and events leading to shutdown of the system or applications. Part of a sample report of MRAS is shown in Table 2. In this report, OS crash failures are reported separately. All other failures leading to OS shutdowns are combined together (further details on these could, however, be provided, if needed.) Note that in this failure category, "planned events" leading to shutdowns are also included. Similarly, in this report, events of different types of the applications it is monitoring is combined together into one number.

In addition to the above information, as availability is of interest to users of servers, MRAS also records time to restore, from which availability is computed and reported. MRAS also provides a breakdown of known shutdown reasons, which is useful for understanding and addressing the main causes of downtime leveraging the Shutdown Event Tracker (SET) feature provided in Windows Server 2003 [16]. In addition to shutdown information, the report also provides information on the crash stop codes and crashing application modules, and a host of other information that can be used for analysis as well as reliability improvement. Further details about the MRS can be found in [6].

Beta versions of MRAS have been deployed to over 200 corporate customers and being used extensively within Microsoft to collect on thousands of servers. MRAS has been instrumental in measuring the reliability of beta versions of Windows Server 2003 at both Microsoft and customer sites.

5. Interpreting Reliability Measurements

The previous Section provided a couple of examples of measurements systems developed within Microsoft to quantify the reliability of the products. Each of these measurement systems have been applied to multiple products. Reliability measurements from these systems clearly provide a method of measuring the reliability trends of individual products and the relative reliability of different versions of the same products. Impact of reliability initiatives can also be measured.

However, care has to be exercised when using these measurements for comparing reliability across products. One method of comparing reliability is to convert the product reliability to a single number using some weights for the different types of failures. Though one can say that for comparison same weights should be used for different products, this may not be the correct approach as reliability of every product depends on a number of factors often unique to the product, and users of a product may attach different "unreliability values" to failures as compared to another product. If we select the weights that capture the users view, then comparison of the final number is possible, but validating the weights is not easy.

Of course reliability of different products with respect to some failure type can be compared. However, even with this comparison one has to be careful as even though we have a framework for identifying failures and system to record them, recording of events is still done by products. If we want to compare reliability of different products, uniform policies for recording events will be imposed. Only when one is sure that different products are recording all the same events can comparisons be meaningfully done.

The reliability measurements provided by the systems can have other uses as well. As many studies have shown, technical reliability is only one of many factors that impact the end users perception of reliability. Examples of other factors that impact the users perception of the product reliability are installation issues (configuration problems, driver incompatibility), product learning (failure rate decrease following product installation), patch distribution, ease of management, interoperability etc. Reliability measurements can be used to understand these factors.

For instance, the ease of product installation can be measured based on the time it takes for a product reliability to reach steady state following its installation.

If configuration failures are being recorded and measured, then the failure rate trend for those failures will also provide insight in this aspect. Similarly, product learning and patch distribution can be measured through comparing the reliability of systems that install the product shortly after its release, against product that are installed a year after release. Ease of management may be measured as the number of actions/reboots it takes to perform any management action.

Using the understanding between different failure rates and reliability attributes, it is possible to compare reliability attributes across products. For instance, ease of installation can be compared by using failure data collected from the different products for a period immediately after installation (e.g. 2 weeks after application installation). Obviously the comparisons are more relevant when applied to products within the same class (e.g. application to application would be valid but application to operating system would require much greater interpretation).

Total Running Time	**No of Shutdowns**	**No. of OS Crashes**	**No of App Exceptions**	**OS Crash Rate**	**OS Shutdown Rate**	**App Exception Rate**
622 years	11243	142	281	0.23 (per year)	18.07 (per year)	0.45 (per year)

Table 2: Parts of an MRAS report

6. Summary

Software reliability measurement efforts in past have focused on large server based products, where users report failures, and the user population base is well known. For a mass market product that runs in different operational profiles with different user groups attaching different levels of importance to different types of failures, this approach is not suitable. Reliability measurement for such a product throws up new problems.

For mass market products, we define reliability as a vector of failure rates for different types of failures. This definition allows different reliability experience of different groups to be quantified in a manner consistent with their environment. In this paper, we discussed some of the key issues that arise when measuring reliability of mass market products, and suggested a failure classification framework that can be used for capturing data on failures.

We have described two example systems for measuring reliability of applications. One is the Office Customer Experience Improvement Program (CEIP), which uses the subscription mechanism to specify the observed group. Detailed failure and usage data is collected through logging of different events. The second example is that of Microsoft Reliability Analysis Service (MRAS), which uses the logging mechanism of Windows server. Products running on the server can record their own events. The event log is sent to a central place, where it is analyzed and report given. As only specified servers are monitored, the size of the observed group is known.

This paper focuses primarily on reliability measurement. Of course, once reliability is measured, a product manufacturer also wants to know how to improve the reliability. It will therefore be best if systems and mechanisms put in place for measurement to have the ability to provide information that can help reliability improvement. Both CEIP and MRAS provide detailed information to aid reliability improvement efforts.

We believe that efforts like these can lead towards well established product reliability measurement norms and platforms, and development of the capability of comparing reliability attributes of different software products, as is the case in many engineering disciplines.

7. References

[1] R. Chillarege, S. Biyani, J. Rosenthal, “Measurement of failure rate in widely distributed software”, Proc. 25th Fault Tolerant Computing Symposium, FTCS-25, 1995, pp. 424-433.

[2] R. Chillarege, et. al. “Orthogonal defect classification – A concept for in-process measurements”, *IEEE Trans. On Software Engineering,* Vol 18(11), Nov 1992, pp. 943-956.

[3] R. Chillarege, What is software failure, *IEEE Transactions on Reliability,* Vol 45(3), Sept 1996, pp. 354-355.

[4] F. Cristian, “Understanding fault-tolerant distributed systems”, *Communications of the ACM,* Vol 34(2), Feb 1991, pp. 56-78.

[5] W. Farr, “Software reliability modeling survey” in *Software Reliability Engineering,* Editor: M. R. Lyu, McGraw Hill and IEEE Computer Society Press, 1996, pp. 71-117.

[6] M. R. Garzia, “Assessing the reliability of windows servers”, Proc. Conference on Dependable Systems and Networks (DSN), San Francisco, 2003.

[7] A. L. Goel, “Software reliability models: Assumptions, limitations, and applicability”, *IEEE Transactions on Software Engineering,* Vol 11:12, 1985, pp. 1411-1423.

[8] J. Gray, “A census of Tandem system availability between 1985 and 1990”, *IEEE Transactions on Reliability,* Vol 39:4, Oct 1990, pp. 409-418.

[9] IEEE, *IEEE Guide to Classification for Software Anomalies,* IEEE Standard 1044.1, 1995.

[10] R. K. Iyer and I. Lee, Measurement-based analysis of software reliability, in *Software Reliability Engineering,* Editor: M. R. Lyu, McGraw Hill and IEEE Computer Society Press, 1996, pp. 303-358.

[11] R. K. Iyer, S. E. Butner, and E.J. McCluskey, A statistical failure/load relationship: results of a multi-computer study, *IEEE Trans. On Computers,* Vol C-31, July 1982, pp. 697-706.

[12] R.K. Iyer and P. Velardi, Hardware-related software errors: measurement and analysis, *IEEE Tran. On Software Engg.,* Vol. SE-11 (2), Feb 1985, pp.223-231.

[13] W. D. Jones, M. A. Vouk, “Field data analysis” in *Software Reliability Engineering,* Editor: M. R. Lyu, McGraw Hill and IEEE Computer Society Press, 1996, pp 439-489.

[14] S. Kan, D. Manlove, B. Gintowt, “Measuring system availability – field performance and in-process metrics”, ISSRE 2003, Supplementary Proceedings, pp. 189-199.

[15] B. Murphy, T. Gent, “Measuring system and software reliability using an automated data collection process”, *Quality and Reliability Engineering International,* 1995.

[16] B. Murphy and B. Levidow, Windows 2000 dependability, *Proc. IEEE DSN,* June 2000.

[17] J. D. Musa, A. Iannino, and K. Okumoto, *Software Reliability – Measurement, Prediction, Application,* McGraw-Hill, 1987.

[18] K. S. Trivedi, *Probability and Statistics with Reliability, Queuing and Computer Science Applications,* Second Edition, John Wiley and Sons, 2002.

[19] A. P. Wood, “Software Reliability from the customer view”, IEEE Computer, August 2003, pp. 37-42.