The Effects of Fault Counting Methods on Fault Model Quality

Allen P. Nikora
Jet Propulsion Laboratory,
California Institute of Technology
Pasadena, CA 91109-8099
Allen.P.Nikora@jpl.nasa.gov

John C. Munson
Computer Science Department
University of Idaho
Moscow, ID 83844-1010
jmunson@cs.uidaho.edu

ABSTRACT

Over the past several years, we have been developing methods of predicting the fault content of software systems based on measured characteristics of their structural evolution. In previous work, we have shown there is a significant linear relationship between code churn, a set of synthesized metrics, and the rate at which faults are inserted into the system in terms of number of faults per unit change in code churn. A limiting factor in this and other investigations of a similar nature has been the absence of a quantitative, consistent, and repeatable definition of what constitutes a fault. The rules for fault definition were not sufficiently rigorous to provide completely unambiguous and repeatable fault counts.

Within the framework of a space mission software development effort at the Jet Propulsion Laboratory (JPL) we have developed a standard for the precise enumeration of faults. This new standard permits software faults to be measured directly from configuration control documents. To this end we have developed a set of tools to perform fault measurement on the JPL effort and have used those measurements in developing fault models. Our results indicate that reasonable predictors of the number of faults inserted into a software system can be developed from measures of the system's structural evolution.

To compare fault-counting methods on the quality of the fault models constructed from those counts, we analyzed the relationship between measurements of the JPL effort's structural evolution and three sets of fault counts. One set of fault counts was obtained using the definition we had previously developed; the other two counts were obtained by using other definitions of faults that have been proposed. The new fault definition provides higher quality fault models than those obtained using the other definitions of fault.

KEYWORDS: defect content estimation techniques, fault prediction, software measurement.

1. Introduction

Over the past several years, we have been investigating relationships between measurements of a software system's structural evolution and the rate at which faults are inserted into that system [Muns98, Niko98]. Measuring the structural evolution of a software system has proven to be a well-defined task that can easily be automated. Unfortunately, it has not been as easy to measure the number of faults inserted into the system - there has been no particular definition of just precisely what a software fault is. In the face of this difficulty it is hard to develop meaningful associative models between faults and code attributes. In developing a model, we would like to know how to count faults in an accurate and repeatable manner just we would expect to enumerate statements, lines of code, and so forth. In measuring the evolution of a system to talk about rates of fault introduction and removal, we measure in units proportional to the way that the system changes over time. Changes to the system are visible at the module level (by module we mean procedures, functions, and methods), so we measure at that level of granularity. Since the measurements of system structure are collected at the module level, we also strive to collect information about faults at the same granularity.

A fault, by definition, is a structural imperfection in a software system that may lead to the system's eventually failing. It is a physical characteristic of the system of which the type and extent may be measured using the same ideas used to measure the properties of more traditional physical systems. People making errors in their tasks introduce faults into a system. In order to count faults, there must be a well-defined method of identification that is repeatable, consistent, and identifies faults at the same level of granularity as our static source code measurements.

We have recently proposed a quantitative definition for software faults that allows automated identification and counting of those faults at the module level [Muns02]. Using this definition, we have identified strong relationships between measured structural change to a software system and the number of faults inserted into that system. The results obtained in collaboration with a space mission software technology development effort at JPL [Dvo99] indicate that our technique of counting faults can be used to develop fault models with good predictive power. However, there are other ways of counting software faults besides the technique we proposed. In this paper, we describe two other fault-counting techniques and compare the models resulting from the application of those methods to the models obtained from the application of our proposed definition.

2. Related Work

Over the past several years, a great deal of work has been done in the area of using measurements of software systems to identify fault-prone components and predict their fault content. Examples of this work include the classification methods proposed by Khoshgoftaar and Allen [Khos01a] and by Ghokale and Lyu [Ghok97], Schneidewind's work on Boolean Discriminant Functions [Schn97], Khoshgoftaar's application of zero-inflated Poisson regression to predicting software fault content [Khos01], and Schneidewind's investigation of logistic regression as a discriminant of software quality [Schn01]. Each of these efforts has provided useful insights into the problem of identifying fault-prone software components prior to test. However, these studies used different definitions at varying levels of precision of what constitutes a fault. For example, the definition of the "*Fault*" response variable used for the study reported in [Khos01a] is "the number of faults discovered in a source file". Here the definition used to identify and count faults is not made explicit. The definition used in [Schn97] and [Schn01] is the number of Discrepancy Reports (DRs) written against modules, where the DRs record (observed) deviations from requirements. Since DRs are readily visible artifacts of the problem reporting system, this definition is repeatable for the system that was studied, and is related to a system's quality. Other studies may have different definitions for faults.

Neither do current standards seem to provide quantitative definitions for faults. The following definition of what constitutes a fault is typical of that provided by current standards: "A manifestation of an error in software. A fault, if encountered, may cause a failure" [IEEE88, IEEE83]. This establishes a fault as a structural defect in a software system that underlies the failure of that system to operate as expected, but does not help in determining the type of failure that was observed, or establish how individual faults may be identified or measured. Some standards address the issue of the type of failure observed by describing schemes for classifying anomalies recorded during software development and operation. For instance, [IEEE93] provides details of an anomaly classification process, as well as criteria for classifying the type of anomaly observed, at what point in the development process the anomaly was observed, and the action taken in response to the anomaly. One particular table in this standard, Table 3c, allows classification of the type of behavior exhibited by the anomaly (e.g., "precision loss") or the type of defect that led to the anomaly (e.g., "referenced wrong data variable"). This type of scheme is helpful in determining the underlying causes of faults and failures, so that the development process may be modified to 1) identify the types of faults on which fault detection and removal resources should be focused for the current development effort, and 2) minimize the introduction of the most common types of faults in future development tasks. However, classification standards do not provide enough information to help count the number of faults in the system. Returning to Table 3c of [IEEE93], we see that some of the anomaly types can readily be traced to a single fault (e.g., "Operator in equation incorrect"). However, the response to an "I/O Timing" anomaly may involve changes to many lines of source code spread across multiple source code files. In this case, the standard does not provide enough information to allow counting the number of faults at the module level.

During a small study on a JPL flight system several years ago [Niko98], we recognized the importance of developing a standard, quantitative definition for faults – since there were multiple researchers working on this task, we needed to have a common understanding of what constitutes a fault. The fault definitions available at that time were either ambiguous or not quantitative, or relied on development practices that did not necessarily apply to the effort being studied. In an attempt to define an unambiguous set of rules for identifying and counting faults, we developed an empirical taxonomy based on the types of editing changes we observed in response to reported failures in the system [Niko97]. We found strong indications that a system's measured structural evolution could predict the fault insertion rate. However, this study had two limitations:

- The study was relatively small – less than 50 observations were used in the regression analysis relating the number of faults inserted to the amount of structural change.
- More importantly, the definition of faults that was used was not quantitative. Although the rules provided a way of classifying the faults by type, and attempted to address faults at the level of individual modules, they were not sufficient to enable repeatable and consistent fault counts by different observers to be made. The rules in and of themselves were unreliable.

Three years ago, we started a collaboration with a space mission software technology development effort at JPL [Dvo99] to address the limitations of the earlier study. Our main concern was developing a quantitative definition of faults, so that we could automate what had been a time-consuming manual activity in the earlier study, the identification and counting of repaired faults at the module level. Our hope was that this would provide us with unambiguous, consistent, and repeatable fault counts, as well as a substantially larger number of observations than the earlier study.

To develop fault predictors for evolving systems, two types of measurements must be made:

- The structural evolution of a system as it changes over a series of builds.

- The number of faults discovered during the system's development.

Measuring a system's structural evolution for the collaborating software system was a straightforward activity – we developed and deployed the Darwin network appliance [Cyla03] for automatically make these measurements. Since we had access to the effort's source code repository, Darwin was able to take structural measurements of each version of each module (i.e., function or method) in the system and use those measurements to produce quantitative reports of the system's evolutionary history according to the techniques described in Sections 7 and 8. We were also able to develop a quantitative definition for software faults [Muns02]. Using this definition, we were able to unambiguously and repeatably identify and count faults, and develop models for predicting faults at the level of individual modules.

3. Problem Statement

The general objective of our current work is to develop practical methods of predicting fault content based on structural characteristics that can be used by production software development efforts to help them better manage the quality of the systems they create. We chose to search for relationships between the rate at which faults are inserted into source code and the measured structural evolution of the source code. If we are able to estimate the rate at which faults are inserted into a system and the structural evolution of that system is being monitored, then we should be able to estimate the system's fault content at any time during its development. This process, however, is predicated on our ability to define very precisely the definition of a software fault and to be able to these faults with a high degree of precision.

Although other types of artifacts in the software development process could have been analyzed, working with source code has two advantages:

- Measuring structural attributes of source code can be easily automated.
- Since the source code is controlled by a configuration management system, different versions of the system can be easily and unambiguously identified. In particular, a baseline against which all other versions are to be measured can be easily established.

The specific objective of this research paper is to compare the quality of the fault predictors created using the fault counting technique developed in an earlier phase of our work with fault predictors developed using other proposed methods of counting faults.

4. A Description of the Mission Data System

We worked in collaboration with the Mission Data System (MDS), a mission software technology development effort in progress at JPL, to collect and analyze the data reported herein. We were able to measure the structural evolution of the MDS during the development of a specific release. For every failure reported against the MDS, we were also able to identify the changes made to each module in response to that failure, and thereby count the number of faults that had been repaired. These measurements were inputs to regression analyses to identify relationships between the measured structural evolution and the number of faults discovered.

Until recently, planetary exploration missions were spaced years apart, with little attention to software reuse, given the rapid pace of computer technology and computer science. Also, since radiation-hardened flight computers remain years behind their commercial counterparts in speed and memory, flight software has typically been highly customized and tuned for each mission. In order to use software engineering resources more effectively and to sustain a quickened pace of missions, JPL initiated the MDS project in April 1998 to define and develop an advanced multi-mission architecture for an end-to-end information system for deep-space missions. MDS is aimed at several institutional objectives: earlier collaboration of mission, system and software design; simpler, lower cost design, test, and operation; customer-controlled complexity; and evolvability to in situ exploration and other autonomous applications.

Some important ways in which MDS differs from earlier systems are as follows:

- When appropriate, capabilities can be migrated from ground-based systems to flight systems to simplify operations.
- MDS is founded upon a state-based architecture, where state is a representation of the momentary condition of an evolving system.
- Domain knowledge is expressed explicitly in models rather than implicitly in program logic.
- Missions are to be operated via specifications of the desired state rather than sequences of actions[Dvo99].

For our study, the structural evolution of the MDS was measured over a period from October 20, 2000, through April 26, 2002. The first date corresponds to the date on which the first source files for the most recent increment were checked into the CM library. The system contains over 15000 distinct modules; over the time interval analyzed studied, there were over 1500 builds of the MDS. The total number of distinct versions of all modules was greater than 65,000. Over 1400 problem reports were included in the analysis; these problem reports provided the information from which the number of repaired faults was computed.

5. Structural Metrics Used in this Study

We would like very much to understand the distribution of faults in the code that we are building. To this end, it would be very useful to just measure them as we are developing the software. Nature, unfortunately, is both fickle and coy. She will not disclose these faults to us. We cannot measure then until we have fixed them. We have learned over time, however, that the distribution of faults in an evolving software systems is distinctly related to software attributes that we can measure. We can then use our historical data to build models that will permit us to understand 1) where faults are likely to be in the code that we have developed, 2) where the faults are located in the changes that we have just made, and 3) determine the rate at which faults are being introduced into changes that are being made to the underlying software system.

We have obtained measurement data from the Darwin system on the target software system. These data were obtained by checking out each build of the system from the configuration control system, then applying the measurement tools incorporated in the Darwin Network Appliance.

5.1. Static Metrics

The specific metrics used in this study are listed in Table 1. These metrics were obtained for both the C and the C++ code modules in the MDS system. The precise definition of each of these metrics and the standard used to measure them can be found in Munson [Muns02a].

Table 1 - Metrics Used in This Study

Metric	Definition
Exec	Number of executable statements
NonExec	Number of non-executable statements
N_1	Total operator count
η_1	Unique operator count
N_2	Total operand count
η_2	Unique operand count
Nodes	Number of nodes in the module control flow graph
Edges	Number of edges in the module control flow graph
Paths	Number of paths in the module control flow graph
MaxPath	The length of the path with the maximum edges
AvePath	The average length of the paths in the module control flow graph
Cycles	Total number of cycles in the module control flow graph

This metric set represents the essential characteristics of both the size of a program module and its control flow characteristics. All measurements were taken at the module level. For C program elements, a module is a function. For C++ a module is a function or an object.

5.2. Derived Metrics

As has been clearly established from our previous work, these metrics are highly correlated [Muns90, Hall00]. There are twelve metrics. There are not twelve distinct sources of variation. We would like to be able to identify the distinct orthogonal sources of variation and map these twelve raw metrics onto a set of uncorrelated metrics that represent essentially the same information contained in the original twelve metrics.

We used principal components analysis (PCA) [Dil84] to identify the distinct sources of variation. We stopped extracting components when the eigenvalues associated with a component assumed values of less than 1. The results of this analysis are shown in Table 2.

We found three distinct sources of variation in the twelve original raw metrics. We have labeled these as Domain 1, 2, and 3 in this table. Domain 1 is most closely associated with the control flow attributes that relate to the complexity of the control flow graph structure of the measured program modules as is shown by the relatively high values (>0.85) of the Nodes and Edges metrics in this table. Domain 2 is most closely associated with the variety of data processed by a module and the operations performed; Domain 3 is associated with the number of distinct paths through the module. The raw metrics that are most closely associated with each the underlying orthogonal domains have been shown in boldface type in this table.

Table 2 - The Principal Components Analysis

Metric	Domain		
	1	2	3
Exec	.60	.49	.47
NonExec	.64	.53	.18
N_1	.28	.64	.65
η_1	.49	**.70**	.07
N_2	.28	.64	.65
η_2	.35	**.90**	.04
Nodes	**.87**	.31	.27
Edges	**.88**	.31	.27
Paths	.17	-.10	**.89**
MaxPath	**.87**	.35	.29
AvePath	**.86**	.34	.33
Cycles	**.67**	.22	-.02
Eigenvalues	4.79	3.13	2.24

The eigenvalues, in the last row of Table 2 show the relative proportion of variation accounted for by each of these new orthogonal domains. For this particular problem space, the sum of the eigenvalues for the twelve original metrics is 12.0. Thus, the relative proportion of variation accounted for by Domain 1 will be 4.79/12 = 0.40 or 40% of the variation in the original 12 metrics. All three domains together account for approximately 85% of the total variation observed in the original 12 metrics.

For measurement purposes, it is necessary to standardize all original or raw metrics so that they are on the same relative scale. For the i^{th} module m_i^j on the j^{th} build of the system there will be a data vector $\mathbf{x}_i^j = < x_{i1}^j, x_{i2}^j, \ldots, x_{i12}^j >$ of 12 raw complexity metrics for that module. We standardize each of the raw metrics by subtracting the mean $\overline{x}_1^j$ of the metric #1 over all modules in the j^{th} build and dividing by its standard deviation δ_1^j such that $z_{1i}^j = \left(x_{1i}^j - \bar{x}_1^j\right)/\delta_1^j$ represents the standardized value of the first raw metric for the i^{th} module on the j^{th} build.

A by-product of the original PCA of the 12 metric primitives is a transformation matrix, **T**, that will map the z-scores of the raw metrics into the reduced space represented by the three principal components. Let **Z** represent the matrix of z-scores shown in the table above for the original problem. We can obtain new domain metrics, **D**, using the transformation matrix **T** as follows: $\mathbf{D} = \mathbf{ZT}$ where **Z** is a n by 12 matrix of z-scores, **T** is a 12 by 3 matrix of transformation coefficients, and **D** is a n by 3 matrix of domain scores where n is the number of modules being measured in a particular build. The matrix, **T**, for this solution given in columns 2 through 4 of Table 3. The means and standard deviations that are used to compute the z-scores are also shown in columns 5 and 6 of this table.

For each module, there are now three new metrics, each representing one the three orthogonal principal components. For our subsequent investigations into modeling the relationship between code evolution and software faults, these domain scores have the very valuable property that they are uncorrelated. Each of the new metrics represents a distinct source of variation. This will completely eliminate the problem of multicollinearity from the linear regression models that we wish to develop.

6. Measuring Software Faults

Perhaps one of the most important considerations in the measurement of software faults is the ability to scale the fault. Not all faults are equal. Sometimes a simple operator is at fault. The developer used a "+" instead of a "-". Sometimes two or three statements must be modified, added, or deleted to remedy a single fault. We ought to be able identify and enumerate faults mechanically. That is, it should be possible to develop a tool that could count the faults for us. Further, some program changes to fix faults are substantially larger than are others. We would like our fault count to reflect that fact. If we have accidentally mistyped a relational operator like '<' instead of '>', this is very different from having messed up an entire predicate clause from an if statement. The actual changes made to a code module are tracked for us in configuration control systems such as RCS or CVS [Cede93] as code deltas. We must learn to classify the code deltas that we make as to the origin of the fix. In other words, each change to each module should reflect a specific code fault fix, a design problem, or a specification problem. If we manifestly change any code module, significantly change it, and fail to record each fault as we repaired it, we will pay the price in losing the ability to resolve faults for measurement purposes.

6.1. Token-Based Fault Counts

For the definition of fault developed in [Muns02], we based our recognition and enumeration of software faults on the grammar of the language of the software system. Specifically, faults are to be found in statements, executable and non-executable. In very simple terms, these structures will cause our executable statement count, Exec, to change. If any of the tokens change that comprise the statement then each of the change tokens will represent a contribution to a fault count. The granularity of measurement for faults will be in terms of tokens that have changed. Thus if one had typed the following statement in C:

```
a = b + c * d;
```

but had meant to type

```
a = b + c / d;
```

then there is but one incorrect token. In this example, there are eight tokens in each statement. There is one token that has changed. There is one fault. This circumstance is very different when wholesale changes are made to the statement. Suppose that this statement

```
a = b + c * d;
```

was changed to

```
a = b + (c * x) + sin(z);
```

We are going to assume, for the moment, that the second statement is a correct implementation of the design and that the first was not. This is clearly a not coding error. (Generally when changes of this magnitude occur they are design problems.) In this case there are 8 tokens in the first statement and 15 tokens in the second statement. This is a fairly substantial change in the code. Our fault recording methodology should reflect the degree of the change.

The important consideration with this fault measurement strategy is that there must be some indication as to the amount of code that has changed in resolving a problem in the code. We have regularly witnessed changes to tens or even hundreds of lines of code recorded as a single "bug" or fault. However, the number of tokens that have changed to ameliorate the original problem constitutes a measurable index of the degree of the change. To simplify and disambiguate further discussion, consider the following definitions.

Definition: A fault is an invalid token or bag of tokens in the source code that may cause a failure when the compiled code that implements the source code token is executed.

Definition: A failure is the departure of a program from its specified functionalities.

Definition: A defect is an apparent anomaly in the program source code.

Each line of text in each version of the program can be seen as a bag of tokens. That is, there may be multiple tokens of the same kind on each line of the text. When a software developer changes a line of code in response to the detection of a fault, either through normal inspection, code review processes, or as a result of a failure event in a program module, the tokens on that line will change. New tokens may be added. Invalid tokens may be removed. The sequence of tokens may be changed. Enumeration of faults under this definition is simple, straightforward. Most important of all, this process can be automated. Measurement of faults can be performed very precisely, which will eliminate the errors of observation introduced by existing ad hoc fault reporting schemes [Muns02, Muns02a].

An example would be useful to show this fault measurement process. Consider the following line of C code.

(1) a = b + c;

There are five tokens on this line of code. They are B1 = {<a>, <=>, <b>, <+>, <c>} where B1 is the bag representing this token sequence. Now let us suppose that the design, in fact, required that the difference between b and c be computed:

(2) a = b - c;

There will again be five tokens in the new line of code. This will be the bag B2 = {<a>, <=>, <b>, <->, <c>}. The bag difference is B1 - B2 = {<+>, <-> }. The cardinality of B1 and B2 is the same. There are two tokens in the difference. Clearly, one token has changed from one version of the module to another. There is one fault.

Now let us suppose that the new problem introduced by the code in statement (2) is that the order of the operations is incorrect. It should read:

(3) a = c - b;

The bag for this new line of code will be B3 = {<a>, <=>, <c>, <->, <b>}. The bag difference between (2) and (3) is B2 - B3 = {}. The cardinality of B2 and B3 is the same. This is a clear indication that the tokens are the same but the sequence has been changed. There is one fault representing the incorrect sequencing of tokens in the source code.

Continuing the example above, let us suppose that we are converging on the correct solution however our calculations are off by 1. The new line of code will look like this.

(4) a = 1 + c - b;

This will yield a new bag B4 = {<a>, <=>, <1>, <+>, <c>, <->, <b>}. The bag difference between (3) and (4) is B3 - B4 = {<1>, <+>}. The cardinality of B3 is five and the cardinality of B4 is seven. Clearly there are two new tokens. By definition, there are two new faults.

A change may span multiple lines of code. All of the tokens in all of the changed lines so spanned will be included in one bag. This will allow us to determine just how many tokens have changed in the one sequence.

The source code control system should be used as a vehicle for managing and monitoring the changes to code attributable to faults, and to design modifications and enhancements. Changes to code modules should be discrete. That is, multiple failures should not be fixed by one version of the code module. Each version of a module should represent exactly one enhancement or one failure repair.

6.2. Number of Editor Commands

Another way of counting the number of faults is to simply count the number of "sed" commands required to implement the changes made in response to a reported failure. This has the advantage of being simpler than the technique described above, yet still provides an unambiguous and repeatable count that is related to the number of faults repaired.

```
20c20,32
<
---
>
>     template <>
>     int ILD< Mds::Fw::Car::Loki::NullType >::addDependencyToConnector(const
Mds::Fw::Init::InitFunctorBase& /* connector */)
>     {
>        return 0;
>     }
>
>     template <class U>
>     int ILD<U>::addDependencyToConnector(const Mds::Fw::Init::InitFunctorBase&
connector)
>     {
>        return InterfaceListDependency<U>::addDependencyToConnector(connector);
>     }
>
22c34
<     void InterfaceListDependency<TList>::addDependencyToConnector(const
Mds::Fw::Init::InitFunctorBase& connector)
---
>     int InterfaceListDependency<TList>::addDependencyToConnector(const
Mds::Fw::Init::InitFunctorBase& connector)
25a38
>        return
29,31c42,43
<
<          connector);
<
---
>          connector)
>         +
35,39d46
<     template <class U>
<     void ILD<U>::addDependencyToConnector(const Mds::Fw::Init::InitFunctorBase&
connector)
<     {
<        InterfaceListDependency<U>::addDependencyToConnector(connector);
<     }
```

Figure 1 – Differential Comparison of Faulty, Repaired Module

To count faults in this manner, it is first necessary to identify each version of each source file to which changes have been made in response to a given reported failure. If a development effort's problem reporting system tracks the source file revisions associated with each failure report, this becomes a straightforward task. A differential comparison ("diff") is then performed between the version known to be faulty and the version implementing the repairs – an example is shown in Figure 1 (the embedded "sed" commands are indicated in larger boldface type). The number of embedded "sed" commands is then counted and recorded as the number of repaired faults. If we know the starting line of each module within the source files being compared, we are able to assign the correct fault count to individual modules. In our study, the Darwin appliance provides the starting line of each module within a source file along with the structural measurements for that module. For the example shown in Figure 1, the number of faults repaired within the source file is counted as 5, which we then allocate to each of the three modules in this particular source file.

6.3. Number of Modules Changed

An even simpler way of counting faults is to count the number of modules that have changed in response to a reported failure. Consider the problem report shown in Figure 2 below. At the bottom of the problem report is a list of the files that were changed in response to the problem report – for each source file that was changed, the filename and version number of the modified file are given (e.g., the first source file implementing repairs is version 20 of "MDS_Rep/verification/TestMaster/defaults.dot"). By analyzing the modules within each file, we can identify those modules that have changed. One fault is counted for each module that has changed. If the differential comparison shown in Figure 1 were for a source file containing only one module, then only one fault would be counted, even though multiple changes have been made.

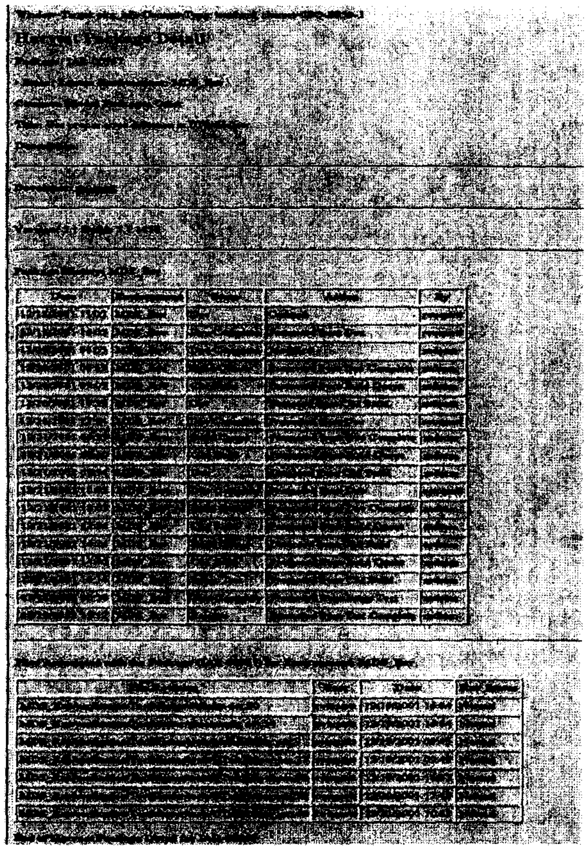
[illegible]

Figure 2 – Failure Report Identifying Changed Source File Versions

7. The Measurement Baseline

Software systems grow and mature just as do biological organisms. We would not think to measure a child at birth and think that we know all there is to know about that child. Measurement is an on-going process. We must, therefore, come to understand that our software systems change rapidly over time. Whenever they are changed, them must be re-measured. To understand what a software system is today, we must have current measurement data on the system together with data on its evolution. We know that faults are removed over time. Modules that have not changed very much are likely to have had most of their faults removed. Modules that have changed a lot are very likely to have had new faults introduced into them. Hence, understanding change activity is vital to our understanding where the problems in the system might be.

The first step in the measuring the evolutionary development of a software system is to establish a baseline reference point in the build process. When a number of successive system builds are to be measured, we choose one of the systems as a baseline system. All others will be measured in relation to the chosen system. Sometimes it will be useful to select the initial system build for this baseline. If we select this system, then the measurements on all other systems will be taken in relation to the initial system configuration.

From the first build of each such system to the last build the differences may be so great as to obscure the fact that it is still the same system. We would like to be able to quantify the differences in the system from its first build, through all builds to the current one. Then and only then will it be possible to know how these systems have changed.

A complete software system generally consists of a large number of program modules. Each of these modules is a potential candidate for modification as the system evolves during development and maintenance. As each program module is changed, the total system must be reconfigured to incorporate the changed module. We will refer to this reconfiguration as a build. For the effect of any change to be felt it must physically be incorporated in a build.

As program modules change from one build to another, the attributes of the modified program modules change. This means that there are measurable changes in modules from one build to the next. Each build is numerically and measurably different from its predecessor with respect to a particular set of metrics. Thus, there is no such thing as measuring a software system but once. Whenever changes are made to a system, those system elements that have changed must be re-measured.

We must be careful to standardize the metric scores in a way that will not erase the effect of trends in the data. For example, let us assume that we were taking measurements on *LOC* and that the system we were measuring grew in this measure over successive builds. If we were to standardize each build of the system by its own mean *LOC* and its own standard deviation, the mean of this system would always be zero. Thus, we will standardize the raw metrics using a baseline system such that the standardized metric vector for the i^{th} module m_i^j on the j^{th} build would be

$$\mathbf{z}_i^j = \frac{\mathbf{x}_i^j - \bar{\mathbf{x}}_i^B}{\boldsymbol{\delta}_i^B}$$

where $\bar{\mathbf{x}}_i^B$ is a vector containing the means of the raw metrics for the baseline system and δ_i^B is a vector of standard deviations of these raw metrics. Thus, for each system, we may build an $m \times k$ data matrix, $\mathbf{Z}^j$, that contains the standardized metric values relative to the baseline system on build B.

Table 3 - The Measurement Baseline

Metric	Domain			Mean	Stdev
	1	2	3		
Exec	.041	.030	.152	1.51	4.33
NonExec	.112	.069	-.067	3.99	5.30
N_1	-.206	.199	.331	4.46	16.66
η_1	.002	.231	-.134	1.36	2.12
N_2	-.206	.199	.331	4.46	16.66
η_2	-.131	.393	-.139	7.08	10.84
Nodes	.282	-.141	-.029	5.01	7.13
Edges	.285	-.144	-.030	4.74	9.26
Paths	-.068	-.215	.608	24.52	865.59
MaxPath	.263	-.121	-.017	3.66	5.60
AvePath	.251	-.123	.012	3.31	4.65
Cycles	.269	-.094	-.179	0.11	0.50

When we have identified a target build, B, to be the baseline build we will then compute the three constituent elements of the baseline. These elements are $\mathbf{T}^B$ the transformation matrix for the baseline build, the vector of metrics means for the baseline build $\bar{\mathbf{x}}_i^B$, and a vector $\boldsymbol{\delta}_i^B$ of standard deviations for this build. For the purposes of this study, the July 1, 2001 build was chosen as the baseline build. Table 3 shows the actual baseline that will be used to compute the derived metrics used in this study.

8. Measuring Change Activity

In order to describe the complexity of a system at each build, it will be necessary to know the version of each of the modules was in the program that failed. Each of the program modules is a separate entity. It will evolve at its own rate. Consider a software system composed of n modules as follows: $m_1, m_2, m_3, \cdots, m_n$. Each build of the system will unify a set of these modules. Not all of the builds will contain precisely the same modules. Clearly there will be different versions of some of the modules in successive system builds. This process is described in detail in [Muns02a].

We can represent the build configuration in a nomenclature that will permit us to describe the measurement process more precisely by recording module version numbers as vector elements in the following manner: $\mathbf{v}^i = < v_1^i, v_2^i, v_3^i, \cdots v_m^i >$. This build index vector will allow us to preserve the precise structure of each for posterity. Thus, v_i^n in the vector v^n would represent the version number of the i^{th} module that went to n^{th} build of the system. The cardinality of the set of elements in the vector v^n is determined by the number of program modules that have been created up to and including the n^{th} build. In this case the cardinality of the complete set of modules is represented by the index value m. This is also the number of modules in the set of all modules that have ever entered any build.

The prime objective of this discussion is to demonstrate the measurement process for measuring successive stages of an evolving software system. Thus, we will be able to assess the precise effect of the change from the build represented by v^i to v^{i+1}. These data will serve to structure the regression test activity between builds. Those modules that have the greatest change in complexity from one build to the next should receive the majority of test effort in the regression test activity.

When evaluating the precise nature of any changes that occur to the system between any two builds i, and j, we are interested in three sets of modules. The first set, $M_c^{i,j}$, is the set of modules present in both builds of the system. These modules may have changed since the earlier version but were not removed. The second set, $M_a^{i,j}$, is the set of modules that were in the early build, i, and were removed prior to the later build, j. The final set, $M_b^{i,j}$, is the set of modules that have been added to the system since the earlier build.

As an example, let build i consist of the following set of modules.

$$M^i = \{m_1, m_2, m_3, m_4, m_5\}$$

Between build i and j module m_3 was removed giving. Thus,

$$\begin{aligned} M^j &= M^i \cup M_b^{i,j} - M_a^{i,j} \\ &= \{m_1, m_2, m_3, m_4, m_5\} \cup \{\ \} - \{m_3\} \\ &= \{m_1, m_2, m_4, m_5\} \end{aligned}$$

Then between builds j and k two new modules, m_7 and m_8 are added and module m_2 is deleted giving

$$\begin{aligned} M^k &= M^j \cup M_b^{j,k} - M_a^{j,k} \\ &= \{m_1, m_2, m_4, m_5\} \cup \{m_7, m_8\} - \{m_2\} \\ &= \{m_1, m_4, m_5, m_7, m_8\} \end{aligned}$$

With a suitable baseline in place, it is possible to measure software evolution across a full spectrum of software metrics. We can do this first by comparing average metric values for the different builds. Secondly, we can measure the increase or decrease in system complexity as measured by the changes in the domain metrics, or we can measure the total amount of change the system has undergone across all of the builds to date.

The change in domain score in a single module between two builds may be measured as the absolute value of the difference in domain scores on these two builds. We will call this code churn measure ***domain churn***. In the case of code churn, what is important is the absolute measure of the nature that code has been modified. From the standpoint of fault introduction, removing substantial amounts of code is probably as catastrophic as adding a large amount.

Let $d_{ia}^{B,j}$ represent the i^{th} domain score of the a^{th} module on build j baselined by build B. The new measure of domain churn, χ, for module m_a is simply $\chi_{ia}^{j,k} = \left| d_{ia}^{B,j} - d_{ia}^{B,k} \right|$. That is, the domain churn may be established by computing the baselined domain scores for any two builds and then find the absolute difference between these values. This represents the relative amount of change activity that there has been on each of the three domains between any two builds.

Now we wish to characterize, or measure, the complete change to the system over all of the builds from build *0* to build *L*. Many modules, however, may have come and gone over the course of the evolution of the system. We are only interested in the history of the survivors; those modules that are now in the final build *L*.

It is now possible to compute the total domain change activity for the aggregate system across all of the system builds. The total domain change activity of the system for module m_a on domain *i* is the sum of the domain churn for this module from the point of its first introduction to the final build *L* is given by

$$X_{ia}^{L} = \sum_{j=0}^{L-1} \chi_{ia}^{j,j+1}.$$

The value of the domain churn X_{ia}^{L} for each module is, of course, dependent on the referent baseline build B.

Let us also observe that if module m_a were not present on builds *j* and *j+1*, then $\chi_{ia}^{j,j+1} = 0$. Also, if module m_a had been introduced on build *j+1* then $\chi_{ia}^{j,j+1} = \left| d_{ia}^{B,j+1} \right|$.

9. Relationships Between the Different Software Fault Counts and Change Activity

As a software system evolves through a number of sequential builds, faults will be identified and the code will be changed in an attempt to eliminate the identified faults. The introduction of new code however, is a fault prone process just as the initial code generation was. Faults are introduced during this evolutionary process.

Source code may change for two distinct reasons. First, some changes to code during its evolution represent enhancements, design modifications, or changes in the code in response to evolving requirements. Second, the code may be changed as part of the fault repair process. Both of these types of incremental code enhancements may also result in the introduction of faults. Thus, as a system progresses through a series of builds, the domain scores of each program module that has been altered must also change. The conjecture we have been exploring is that the rate of change in these domains should serve as a good index of the rate of fault introduction.

To this end, we computed domain scores all of the builds of the MDS system. These domain scores were baselined relative to the July 7, 2001 build of the system, a build more or less intermediate in the sequence of builds. In general, it is not a good practice to use an initial build as a baseline build, since the initial build is generally quite incomplete.

The next step in this investigation was to compute the fault count for each program module. The driving force behind this measurement process was the Internal Anomaly Report (IAR). All changes to the software were tracked under the CCC Harvest version control system (now incorporated into Computer Associates' CM systems – see [CA02]). Each change to a program module was made either as an enhancement or in response to a particular IAR. If a module code delta was attributed to an IAR, then the faults attributed to that change were calculated using the three different techniques described in Section 6.

Once the three different fault counts had been established for each incremental module version, each type of fault count was accumulated so that by the final build a cumulative fault count of that type was available for each module in the final build. The fault counts for modules not in the final build, of course, vanished with the module domain churn values when the modules disappeared from the evolving builds.

We now have, for each module in the final version of the system, three different measures of the number of faults that have been found in that module to date. We also have cumulative domain churn values for each of the three orthogonal domains. To investigate the relationship between the fault content of models and the domain metrics, we now eliminated those modules whose fault count was zero. There are two very good reasons for eliminating these modules. First, a zero fault count for a module on the last build does not imply that there are no faults in this module. It could very well mean that the faults have yet to be discovered. Second, approximately 90% of the modules in the final build have zero fault values. They would clearly dominate any regression model that was developed using them.

With the data from the remaining modules, we developed three multiple linear regression models, one for each type of fault count, with the cumulative fault count as the dependent variable and the domain churn values as independent variables. The regression ANOVAs for these analyses are shown in Table 4, Table 5, and Table 6. It is clear that for each type of fault count, there is a significant relationship between domain churn and a module's fault burden. That is, there is a distinct association with module change activity as measured by each of the three distinct criterion measures and the module domain churn metrics as a measure of code evolution.

Table 4 - Regression ANOVA – Changed Token Counts

Source	Sum of Squares	df	Mean Square	F	Sig.
Regression	10091546	3	3363848	293	$p<0.05$
Residual	6430656	560	11483		
Total	16522203	563			

Table 5 - Regression ANOVA – "Sed" Command Counts

Source	Sum of Squares	df	Mean Square	F	Sig.
Regression	2419	3	806.547	53	$p<0.05$
Residual	10073	668	15.080		
Total	12493	671			

Table 6 - Regression ANOVA – Module Change Counts

Source	Sum of Squares	df	Mean Square	F	Sig.
Regression	65	3	21.835	38	p<0.05
Residual	390	674	0.579		
Total	455	677			

The regression models corresponding to the different types of fault counts are shown in Table 7, Table 8, and Table 9. For the models corresponding to the fault counts produced by computing token differences or counting the number of "sed" commands, Domain 1 is significant. For the model produced with token difference fault counts, Domain 1 dominates, and Domains 2 and 3 do not contribute to our understanding of the fault introduction process. The regression coefficients for these terms are not significant ($p>0.05$). For the model produced with fault counts produced by counting "sed" commands, Domain 2 does contribute to our understanding of the fault insertion mechanism, and indeed dominates the model. Finally, for the model produced with fault counts computed from the number of modules changed, Domains 2 and 3 are the important factors in this model. Domain 1 does not play a significant role.

Table 7 - Regression Model 1 – Token-Based Fault Counts

Model	Coefficients	t	Sig.
(Constant)	18.24	3.5	p<.05
Domain 1 Churn	21.63	17.3	p<.05
Domain 2 Churn	-.59	-0.3	p>.05
Domain 3 Churn	.93	0.7	p>.05

Table 8 - Regression Model 2 – "Sed" Command Counts

Model	Coefficients	T	Sig.
(Constant)	2.484	14.555	p<.05
Domain 1 Churn	.151	3.411	p<.05
Domain 2 Churn	.529	6.489	P<.05
Domain 3 Churn	-0.087	-1.791	p>.05

Table 9 - Regression Model 3 – Module Change Counts

Model	Coefficients	t	Sig.
(Constant)	1.200	35.995	p<.05
Domain 1 Churn	0.009	1.041	p>.05
Domain 2 Churn	0.143	8.920	p<.05
Domain 3 Churn	-0.043	-4.483	p<.05

The domains scores for each of the modules is, in fact, a factor score for that module relative to the measurement baseline. The principal components of the original 13 raw metrics are shown in Table 2. The metrics most closely associated with first principal component, Domain 1, were Nodes, Edges, MaxPath, AvePath, and Cycles. These metrics are attributes of the control flow graph representation of a program module. From this we can infer that for the model developed using fault counts based on token differences, the fault burden attributed to change activity is most closely associated with the change activity in those modules that had the greatest change made to their control structure.

For the model developed using fault counts based on a count of the number of "sed" commands implementing the required changes, Domains 1 and 2 were significant. The metrics most closely associated with Domain 2 are the counts of the unique operators and unique operands; we can infer that for this model, the fault burden is most closely associated with change activity in those modules having the greatest change made to 1) the data items they process, and 2) their control structure.

For the model developed using fault counts based on a count of the number of modules that changed in response to a reported failure, Domains 2 and 3 were significant. According to Table 2, the metric most closely associated with Domain 3 is the number of paths through the module. For this model, then, the fault burden is most closely associated with change activity in those modules having the greatest change made to 1) the data items they process, and 2) the number of paths through the module.

Table 10 – Model Quality

Model Summary	R Square	Adjusted R Square	Std. Error of the Estimate
Model 1	0.61	.061	107.16
Model 2	0.19	0.19	3.88
Model 3	0.14	0.14	0.76

The three regression models, however, are not at all similar when we examine their predictive quality as measured by the R^2 statistic. This statistic is the ratio of sums of squares due to regression to the sums of squares total. Finally, we want to know something about the relative quality of the regression model that we have developed. These data are shown in Table 10. We can see from this table that for the model obtained from fault counts based on token differences (Model 1), the adjusted R^2 is approximately 0.61. This means, roughly, that we can account for approximately 60% of the variation in the cumulative fault count with the cumulative domain churn for Domain 1. This is a very respectable value for the limited metric set that the Darwin tool currently uses. For Models 2 and 3, we see that we can only account for a considerably smaller (less than 20%) percentage of the variation in the cumulative fault count with the cumulative churn in Domains 1, 2, and 3.

Within the framework of this investigation, it is evident that we can develop higher quality fault predictors using fault counts based on token differences than either of the other types of fault counts described in Section 6. Furthermore, the highest quality fault predictor seems to be the simplest. In the module change model shown in Table 9, the dominant factor was measurable changes in the control structure of a module. Among the set of 12 metrics used in this investigation, those metrics most closely associated with the observed variation in software faults were the control metrics shown in the first principal component (Domain 1) of Table 2.

10. Discussion and Future Work

We have seen that the method by which faults are counted can have a significant effect on the fault predictors developed using those counts. Of the predictors developed as part of this study, the one having the highest quality was based on the fault counting technique we developed in an earlier phase of this work. We have also seen that by using an appropriate fault counting technique, predictors with a relatively high degree of accuracy can be developed. For the predictor developed from fault counts based on token differences, about 60% of the variation in the cumulative fault count was explained by our set of measurements, although the number of measurements used in the study was rather limited. This is a sufficiently large value for development efforts to start using these measurements as a management tool. Software managers should be able to use these measurements to:

- Identify modules having the highest fault burden.
- Determine how many more faults a given module has had inserted into it than another module.

A driving force behind our research has been the quest for a scientific means of defining the notion of a fault and quantifying the measurement of these faults. Faults come in different sizes. Some are really small. Some represent really egregious hacks to the code. These differences in size will clearly have an impact on the fault repair process.

As a measure of the success of our endeavors, it is now possible to describe to other researchers exactly what we considered to be a fault. It is now possible to communicate to other researchers in software reliability exactly how we got the measurements that we did. This is the essence of science.

We have, then, developed a functional definition of software faults that can be applied to source code revision management systems for the automatic measurement of software faults. Further, this definition allows faults to be unambiguously measured at the level of individual modules. Since faults are measured at the same level at which structural measurement are taken, it becomes more feasible to construct meaningful models relating the number of faults inserted into a software module to the amount of structural change made to that module. This measurement process makes it much more practical to analyze large software systems such as those developed to support NASA flight missions. In other words, faults may be quantified by a software tool that can analyze the deltas in code modules maintained by the configuration control system and measure those changes specifically attributable to failure reports.

Future work will involve investigation of these relationships for additional software development efforts at JPL and other NASA centers. Although fault counts based on token differences resulted in the highest quality fault predictor for this study, there is insufficient data at this point to generalize this conclusion. Detailed analysis of additional software development efforts is required before more general conclusions can be reached. We have started collaborative efforts with additional projects at JPL to perform this investigation; we have also started working with the Software Assurance Technology Center at the Goddard Space Flight Center to investigate development efforts at other NASA centers.

We are also interested involve enlarging the set of measurements taken by Darwin and determining the effect of the enlarged set on the accuracy of the fault predictors. For instance, Darwin does not currently take any measurements specifically related to objects (e.g., number of methods, depth of an object in the class hierarchy). Future versions of Darwin might well implement the object-oriented measures proposed by Chidamber and Kemerer [Chid94]. Our criterion for including these new metrics into the Darwin system, of course, is that they are able to 1) identify new sources of variation in the metric space and 2) that the explain additional variation in our fault criterion measure.

The Darwin network appliance is still in its period of infancy. It presently incorporates a relatively simple metric analysis tool that is capable of explaining at least 60% of the variation in our software fault measure. The main issues that had to be solved first in the measurement process were infrastructure problems. We are now able, however, to track all aspects of software source evolution. Mechanisms are in place to measure software faults very precisely as described in [Muns02]. Mechanisms are also in place to automate the complete measurement of a rapidly evolving software system. As a preliminary report and investigation, the Darwin measurement system has clearly established itself as a viable tool for the understanding of the etiology of software faults and their relationship to software attribute that can be measured.

There may be uncontrolled sources of noise, which we intend to address in future work. For example, developers might be making enhancements to the system at the same time they are responding to a reported failure. In this case, the enhancements would be counted as repairs made in response to the failure. Addressing this issue will involve selecting an appropriate subset of the reported failures and interviewing developers about the changes made in response to those failures. We will be careful to select representative failures from all system components to control for the noise inserted by each development team. We will also select reported failures from different times during the development effort, to determine whether the number of enhancements reported as fault repair changes over time.

Acknowledgments

The work described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology. This work is sponsored by the National Aeronautics and Space Administration's IV&V Facility. The authors wish to thank the MDS project for the cooperation that made this study possible.

References

[CA02] Computer Associates, "AllFusion Harvest Change Manager Features, Descriptions & Benefits", Feb. 11, 2002, available at: http://www3.ca.com/Files/FactSheet/af_harvest_cm_fdb.pdf

[Cede93] Per Cederqvist, "Version Management with CVS for CVS 1.11.1p1", available at: http://www.cvshome.org/docs/manual/.

[Chid94] S. Chidamber, C. Kemerer, "A Metrics Suite for Object Oriented Design", IEEE Transactions on Software Engineering, vol. 20, no. 6, June, 1994, pp. 476-493.

[Cyla03] "The Darwin Software Engineering Measurement Appliance", Cylant, http://www.cylant.com/

[Dil84] W. Dillon, M. Goldstein, Multivariate Analysis: Methods and Applications, Wiley-Interscience, 1984, ISBN 0471083178

[Dvo99] D. Dvorak, R. Rasmussen, G. Reeves, A. Sacks, "Software Architecture Themes In JPL's Mission Data System", AIAA Space Technology Conference and Exposition, September 28-30, 1999, Albuquerque, NM.

[Ghok97] S. S. Gokhale, M. R. Lyu, "Regression Tree Modeling for the Prediction of Software Quality", proceedings of the Third ISSAT International Conference on Reliability and Quality in Design, pp 31-36, Anaheim, CA, March 12-14, 1997

[Hall00] G. A. Hall and J. C. Munson, "Software evolution: code delta and code churn", Journal of Systems and Software 54 (2) (2000) pp. 111-118

[IEEE83] "IEEE Standard Glossary of Software Engineering Terminology", IEEE Std 729-1983, Institute of Electrical and Electronics Engineers, 1983.

[IEEE88] "IEEE Standard Dictionary of Measures to Produce Reliable Software", IEEE Std 982.1-1988, Institute of Electrical and Electronics Engineers, 1989.

[IEEE93] "IEEE Standard Classification for Software Anomalies", IEEE Std 1044-1993, Institute of Electrical and Electronics Engineers, 1994.

[Khos01] T. Khoshgoftaar, "An Application of Zero-Inflated Poisson Regression for Software Fault Prediction", proceedings of the 12th International Symposium on Software Reliability Engineering, pp 66-73, Hong Kong, Nov, 2001.

[Khos01a] T. M. Khoshgoftaar, E. B. Allen, "Modeling Software Quality with Classification Trees", in H. Pham (ed), Recent Advances in Reliability and Quality Engineering, Chapter 15, pp 247-270, World Scientific Publishing, Singapore, 2001.

[Muns90] J. C. Munson and T. M. Khoshgoftaar, "Regression Modeling of Software Quality," Information and Software Technology, Vol. 32 No. 2 March 1990, pp. 105-114.

[Muns98] J. Munson and A. Nikora, "Estimating Rates Of Fault Insertion And Test Effectiveness In Software Systems" Proceedings of the Fourth ISSAT International Conference on Reliability and Quality in Design, August 12-14, 1998 pp. 263-269.

[Muns02] J. Munson, A. Nikora, "Toward a Quantifiable Definition of Software Faults", Proceedings of the 13th IEEE International Symposium on Software Reliability Engineering, IEEE Press.

[Muns02a] J. Munson, Software Engineering Measurement, CRC Press, 2002, ISBN 0849315034.

[Niko97] A. Nikora, J. Munson, "Finding Fault with Faults: A Case Study", with J. Munson, proceedings of the Annual Oregon Workshop on Software Metrics, Coeur d'Alene, ID, May 11-13, 1997.

[Niko98] A. P. Nikora, J. C. Munson, "Determining Fault Insertion Rates For Evolving Software Systems", proceedings of the 1998 IEEE International Symposium of Software Reliability Engineering, Paderborn, Germany, November 1998, IEEE Press.

[Niko01] A. Nikora, J. Munson, "A Practical Software Fault Measurement and Estimation Framework", Industrial Presentations proceedings of the 12th International Symposium on Software Reliability Engineering, Hong Kong, Nov 27-30, 2001.

[Schn97] N. F. Schneidewind, "Software Metrics Model for Integrating Quality Control and Prediction", proceedings of the 8th International Symposium on Software Reliability Engineering, pp 402-415, Albuquerque, NM, Nov, 1997.

[Schn01] N. F. Schneidewind, "Investigation of Logistic Regression as a Discriminant of Software Quality", proceedings of the 7th International Software Metrics Symposium, pp 328-337, London, April, 2001.