

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
Chief Science Officer
Cylant, Inc.
Lexington, MA 02421
jmunson@cylant.com

Abstract

Over the past several years, we have been developing software fault predictors based on a system's measured structural evolution. We have previously 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 unambiguous, 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. 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.

We compared the new method of counting faults with two existing techniques to determine whether the fault counting technique has an effect on the quality of the fault models constructed from those counts. 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, software modeling.

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, Niko03]. 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 quantitative 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. Since code attributes are collected at the module level, we strive to collect information about faults at the same granularity.

We have recently developed 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 [Niko03]. The results obtained in collaboration with the Mission Data System (MDS), 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 three other fault-counting techniques and compare the models resulting from the application of two of those methods to the models obtained from the application of our proposed definition.

2. Related Work

Over the past several years, researchers have developed a number of fault predictors based on measurable characteristics of a software system. Examples include the classification methods proposed by Khoshgoftaar and Allen [Khos01a] and by Gokhale and Lyu [Gokh97], 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 discriminator of software quality [Schn01]. Each of these 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 reported in [Khos01a] is "the number of faults discovered in a source file". 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. This definition is repeatable for the system under study, and is related to a system's quality. However, it refers to the number of failures rather than the number of 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 may cause the system to fail, but does not help in determining how individual faults may be identified or measured.

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. 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, the definition of faults that was used was not quantitative. Although the rules provided a way of classifying the faults, and attempted to address faults at the module level, they were not sufficient to enable repeatable and consistent fault counts by different observers.

Four years ago, we started a collaborative effort with the MDS to address this limitation 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.

For our study, the structural evolution of the MDS was measured over a period from October 20, 2000, through April 26, 2002. 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.

3. Problem Statement

The overall objective of our work is to develop practical methods of predicting software fault content based on measurable characteristics that can be used by software development efforts to help them better manage the quality of their systems. We searched relationships between the rate at which faults are inserted into source code and the measured structural evolution of the source code. Such a relationship would allow us to estimate the system's fault content at any time during its development. This process, however, is predicated on our ability to define precisely software faults and to measure them with a high degree of precision.

The driving force behind modeling the relationship between software faults and software attributes such as size is that we can measure software attributes directly. It is possible to develop stringent standards for measuring source code. Measuring software faults is quite another matter. Fault measures can either be relatively fine grained or coarse grained. We will identify three distinct methods of measuring faults and then model the relationship of certain software attributes to these different fault measurement strategies.

Although other types of software artifacts could have been analyzed, source code has two advantages:

- Measuring its structural attributes is 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 objective of this paper, then, is to model the relationship of specific software attributes with the fault predictors created using our recently developed fault counting technique and compare this model with fault predictors developed using other proposed methods of counting faults.

4. Structural Metrics Used in this Study

The software attribute measurement data for this study were obtained from the Darwin system [Cyla03] that was used to measure and manage metric data on the target software system. These data were obtained by checking out each build of the system from the configuration control system and then applying the measurement tools incorporated in the Darwin Network Appliance. The Darwin system collected measurement data across the multiple builds of the target systems.

4.1. Software Attribute Measures

The software attributes that were measured for this study are shown in Table 1. These measures were obtained for both the C and the C++ code modules in the MDS system. The precise definition of each of these measures and the standard used to measure them can be found in [Muns02a].

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.

4.2. Derived Metrics

Our previous work has shown that the metrics shown in Table 1 are highly correlated [Muns90, Hall00]. Using principal components analysis (PCA) [Dil84], we identified the distinct orthogonal sources of variation and mapped these twelve raw metrics onto a set of uncorrelated metrics that represent essentially the same information. We stopped extracting components when the eigenvalues associated with a component assumed values less than 1. The PCA results are shown in Table 2. The eigenvalues are shown as the last row of Table 2 – the sum of all of the eigenvalues for the 12 original metrics is 12.0. The three domains together account for approximately 85% of the total variation observed in the original 12 metrics.

We found three distinct sources of variation in the twelve original raw metrics that we have labeled as Domain 1, 2, and 3 in Table 2. Domain 1 is most closely associated with the control flow attributes relating to the module's control flow graph structure

complexity, 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 most closely associated with underlying orthogonal domain are shown in boldface type in Table 2.

Table 1 - Software Attributes 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 #/cycles in the module control flow graph

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

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 $x_i^j = \langle x_{i1}^j, x_{i2}^j, \dots, x_{i12}^j \rangle$ of 12 raw metrics for

that module. We standardize each of the k raw metrics by subtracting the mean \bar{x}_k^j of the metric k over all modules in the j^{th} build and dividing by its standard deviation σ_k^j such that $z_{ki}^j = (x_{ki}^j - \bar{x}_k^j) / \sigma_k^j$ represents the standardized value of the k^{th} 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 maps 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 Table 2 above for the original problem. We obtain new domain metrics, D , using the transformation matrix T as follows: $D = ZT$ where Z is an n by 12 matrix of z-scores, T is a 12 x 3 matrix of transformation coefficients, and D is a n x 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 used to compute the z-scores are also shown in columns 5 and 6 of Table 3.

For each module, there are now three new metrics, each representing one of the three orthogonal principal components. These domain scores are uncorrelated, thereby eliminating the problem of multicollinearity from the linear regression models that we wish to develop.

5. Measuring Software Faults

One of the major problems in the analysis of software faults and their relationship to measurable software attributes is the lack of a standard way of counting faults. Because of this, previous attempts to develop models of software quality are of questionable value. We define below three different approaches to fault measurement from a very low level token based measure to a high-level failure report level measure.

5.1. Token-Based Fault Counts

One of the most important considerations in the measurement of software faults is the ability to scale the fault. Sometimes a simple operator is at fault; the developer used a "+" instead of a "-". Other times two or three statements must be modified, added, or deleted to remedy a single fault. Further, some program changes to fix faults are substantially larger than are others. We would like our fault count to reflect that fact. The actual changes made to a code module are tracked 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 repair action for each module should reflect a specific code fault fix, a design problem, or a specification problem. If we change any code module and fail to record each fault as we repair it, we lose the ability to resolve faults for measurement purposes.

The important consideration with any 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. 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 implementing the tokens 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.

By taking as the fault count the number of tokens that have changed, we take into account the size and extent of the fault.

Each line of text in each version of the program can be seen as a bag of tokens. When a developer changes a line of code in response to the detection of a fault, the tokens on that line will change. New tokens may be added, invalid tokens may be removed, or the sequence of tokens may be changed. Enumeration of faults under this definition is unambiguous and consistent, and can be automated. This definition of fault eliminates the errors introduced by existing ad hoc fault reporting schemes [Muns02, Muns02a].

The following example shows 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 $B1 = \{ \langle a \rangle, \langle = \rangle, \langle b \rangle, \langle + \rangle, \langle c \rangle \}$ where $B1$ is the bag representing this token sequence. Now suppose 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 $B2 = \{ \langle a \rangle, \langle = \rangle, \langle b \rangle, \langle - \rangle, \langle c \rangle \}$. The bag difference is $B1 - B2 = \{ \langle + \rangle, \langle - \rangle \}$. 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, indicating one fault.

Now suppose that the 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 $B3 = \{<a>, <=>, <c>, <->, \}$. 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 this example, suppose that we are converging on the correct solution but the calculations are off by 1. The new line of code will look like this.

(4) $a = 1 + c - b;$

This yields a new bag $B4 = \{<a>, <=>, <1>, <+>, <c>, <->, \}$. 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, indicating two new faults.

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

5.2. Number of Editor Commands

Another way of determining the number of faults is to count the number of "sed" commands required to implement the changes made in response to a reported failure. This is simpler than the technique described above, yet still provides an unambiguous and repeatable count that is related to the number of repair actions performed.

To count faults in this manner, each version of each source file to which changes have been made in response to a given reported failure must be identified. 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 stream editor, "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. 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.

5.3. 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. At the bottom of the problem report shown in Figure 2 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 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.

```

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

```

Figure 1 – Differential Comparison of Faulty, Repaired Module

5.4. Number of Failure Reports

A popular method of approximating the number of faults is to simply count the number of failure reports. At the system level, this technique can work quite well – in fact, we have shown that there is a high correlation (> 0.9) between counts of the number of observed failures and measurements of the amount of structural

change experienced by the system as a whole [Niko03a]. However, we chose not to include failure counts in this study because of the problem of scaling them to the level of individual modules. An individual failure may result in changes to more than one module, as shown in Figure 2. To use failure counts at a module level, it would be necessary to count a failure report multiple times; specifically, for each module repaired in response to that failure report, the number of failures for that module would need to be increased by 1. This assumes that if one of the modules were not changed, the failure would occur. We were not comfortable making this assumption without a detailed analysis of the repair actions, which was beyond the scope of this study.



Figure 2 – Failure Report Identifying Changes

6. The Measurement Baseline

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.

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.

We must 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. 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

$$z_i^j = \frac{x_i^j - \bar{x}_i^B}{\sigma_i^B}$$

where \bar{x}_i^B is a vector containing the means of the raw metrics for the baseline system and σ_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, 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 T^B the transformation matrix for the baseline build, the vector of metrics means for the baseline build \bar{x}_i^B , and a vector σ_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 presents the baseline we used to compute the derived metrics.

7. Measuring Change Activity

In order to describe the complexity of a system at each build, it is necessary to know which version of each module was in the program at any point in time. Consider a software system composed of n modules as follows: $m_1, m_2, m_3, \dots, m_n$. Not all of the builds will contain precisely the same modules; there will be different versions of some of the modules in successive system builds. Details are given in [Muns02a].

We represent the build configuration in a nomenclature that permits us to describe the measurement process more precisely by recording module version numbers as vector elements in the following manner: $v^i = \langle v_1^i, v_2^i, v_3^i, \dots, v_m^i \rangle$. 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.

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. Details of the measurement process are given in [Muns02a].

With a suitable baseline in place, software evolution across a full spectrum of software metrics can be measured. We do this first by comparing average metric values for the different builds. Secondly, we can measure the changes in the domain metrics, or we can measure the total amount of change to the system 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 – faults can be inserted by removing code as well as by adding code.

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 $\chi_{ia}^{j,k} = |d_{ia}^{B,j} - d_{ia}^{B,k}|$. 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 . 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 . Note 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 $\chi_{ia}^{j,j+1} = |d_{ia}^{B,j+1}|$.

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

In this investigation, we computed domain scores all of the available 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. Since the initial build is generally quite incomplete, it is a good idea to change the baseline as development progresses.

The next step in this investigation was to compute the fault count for each program module, using information available from the Internal Anomaly Report (IAR) system. 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 Sections 5.1, 5.2, and 5.3.

After establishing each type of fault count for each incremental module version, the fault counts were 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 vanished with the module domain churn values when the modules disappeared from the evolving builds.

To investigate the relationship between the fault content of models and the domain metrics, we 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. For each type of fault count, there is a distinct association with module change activity as measured by each of the three distinct fault counts and the module domain churn metrics as a measure of code evolution.

Table 4 - Regression ANOVA

Source	Sum of Squares	Df	Mean Square	F	Sig.
Token-Based Fault Counts					
Regression	10091546	3	3363848	293	p<0.05
Residual	6430656	560	11483		
Total	16522203	563			
"Sed" Command Counts					
Regression	2419	3	806.547	53	p<0.05
Residual	10073	668	15.080		
Total	12493	671			
Module Change Counts					
Regression	65	3	21.835	38	p<0.05
Residual	390	674	0.579		
Total	455	677			

Table 5 - Regression Models

Model	Coefficients	t	Sig.
Token-Based Fault Counts			
(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
"Sed" Command Counts			
(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
Module Change Counts			
(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 regression models corresponding to the different types of fault counts are shown in Table 5. 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 differences, 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 contributes to our understanding of the fault insertion mechanism, and indeed dominates the model. Finally, for the model produced with counts of the number of modules changed, Domains 2 and 3 are the important factors in this model. Domain 1 plays no significant role.

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 6. 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. Clearly, Models 2 and 3 are not useful predictors.

Table 6 - Model Quality

Model Summary	R Square	Adjusted R Square	Std. Error of the Estimate
Mdl 1 - Token-Based	0.61	.061	107.16
Mdl 2 - "Sed" Commands	0.19	0.19	3.88
Mdl 3 - Modules Changed	0.14	0.14	0.76

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 5. In the token-based module change model shown in

Table 5, 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.

9. 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 token-based 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.

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 measurements are taken, meaningful models relating the number of faults inserted into a software module to the amount of structural change made to that module can be developed. This measurement process makes it much more practi-

cal to analyze large software systems such as those developed to support NASA flight missions.

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.

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 was sponsored by the National Aeronautics and Space Administration's Office of Safety and Mission Assurance (OSMA) Software Assurance Research Program (SARP), which is administered by NASA's IV&V Facility. The authors also wish to thank the MDS project for the cooperation that made this study possible. Finally, we thank our colleagues who reviewed earlier versions of this paper and made many helpful suggestions.

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, available at: <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, Sep. 28-30, 1999, Albuquerque, NM.
- [Gokh97] 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, 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.
- [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, Ch. 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.
- [Niko03] A. Nikora, J. Munson, "Developing Fault Predictors for Evolving Software Systems", proceedings of the 9th International Software Metrics Symposium, Sep 3-5, Sydney, Australia
- [Niko03a] A. Nikora, J. Munson, "Understanding the Nature of Software Evolution", proceedings of the International Conference on Software Maintenance, Sep 22-26, Amsterdam, The Netherlands.
- [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.