SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2006; **36**:949–969
Published online 2 May 2006 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.737

Building high-quality software fault predictors**

Allen P. Nikora[1,*,†] and John C. Munson[2]

[1]*Jet Propulsion Laboratory, California Institute of Technology, 4800 Oak Grove Drive, Mail Stop 125-233, Pasadena, CA 91109-8099, U.S.A.*
[2]*Computer Science Department, University of Idaho, Moscow, ID 83844-1010, U.S.A.*

SUMMARY

Over the past several years, we have been developing software fault predictors based on a system's measured structural evolution. We have previously shown significant linear relationships 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 such investigations 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 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. 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. Copyright © 2006 John Wiley & Sons, Ltd.

Received 31 January 2005; Revised 30 September 2005; Accepted 14 November 2005

KEY WORDS: software measurement; software fault 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 [1–3]. Measuring the structural evolution of a software system has proven to be a well-defined

*Correspondence to: Allen P. Nikora, Jet Propulsion Laboratory, California Institute of Technology, 4800 Oak Grove Drive, Mail Stop 125-233, Pasadena, CA 91109-8099, U.S.A.
†E-mail: allen.p.nikora@jpl.nasa.gov
**A selected paper from COMPSAC 2004, edited by Eric Wong and Karama Kanoun.

Contract/grant sponsor: National Aeronautics and Space Administration's Office of Safety and Mission Assurance (OSMA)
Contract/grant sponsor: Software Assurance Research Program (SARP)

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 [4]. 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 [3]. The results obtained in collaboration with the Mission Data System (MDS), a space mission software technology development effort at the Jet Propulsion Laboratory (JPL) [5] 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 [6] and by Gokhale and Lyu [7], Schneidewind's work on Boolean Discriminant Functions [8], Khoshgoftaar's application of zero-inflated Poisson regression to predicting software fault content [9], and Schneidewind's investigation of logistic regression as a discriminator of software quality [10]. Each of these has provided useful insights into the problem of identifying fault-prone software components prior to testing. 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 by Khoshgoftaar and Allen is 'the number of faults discovered in a source file' [6]. Other researchers have used the number of Discrepancy Reports (DRs) written against modules [8,10], 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.

In addition current standards do not 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' [11,12]. 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 [2], 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 [13]. 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 20 October 2000, to 26 April 2002. The system contains over 15 000 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 [14] that was used to measure and manage metric data on the target software system. These data were obtained

Table I. 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 number of cycles in the module control flow graph

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 I. These measures were obtained for both the C and the C++ code modules in the MDS system. Munson has developed a precise definition of each of these measures and the standard used to measure them [15].

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

Previous work has shown that the metrics shown in Table I are highly correlated [16,17]. We can also see from Table II that some of the metrics used in this study are highly correlated with each other. For example, the metrics Nodes, Edges, MaxPath, AvePath and Cycles are highly correlated with each other. Models developed from highly correlated metrics may be subject to dramatic changes due to additions or deletions of variables or even discrete changes in metric values. To circumvent this problem, we used Principal Components Analysis (PCA) [18] to map the 12 raw metrics shown in Table I onto a set of uncorrelated metrics that represent essentially the same information. PCA is a statistical technique that reduces the dimensionality of the data by transforming the original feature space and extracting its eigenvectors. The eigenvectors define a linear transformation from the original feature space to a new uncorrelated space. We stopped extracting components when the eigenvalues associated with a component assumed values less than 1. The PCA results are shown in Table II. The eigenvalues are shown as the last row of Table II—the sum of all of the eigenvalues for the

Table II. The PCA.

	Domain		
Metric	1	2	3
Exec	0.60	0.49	0.47
NonExec	0.64	0.53	0.18
N_1	0.28	0.64	0.65
η_1	0.49	**0.70**	0.07
N_2	0.28	0.64	0.65
η_2	0.35	**0.90**	0.04
Nodes	**0.87**	0.31	0.27
Edges	**0.88**	0.31	0.27
Paths	0.17	−0.10	**0.89**
MaxPath	**0.87**	0.35	0.29
AvePath	**0.86**	0.34	0.33
Cycles	**0.67**	0.22	−0.02
Eigenvalues	4.79	3.13	2.24

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 12 original raw metrics that we have labeled as Domain 1, 2 and 3 in Table II. 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 the underlying orthogonal domain are shown in boldface type in Table II.

It is necessary to standardize all original or raw metrics so that they are on the same relative scale. For the ith module m_i^1 on the jth build of the system there will be a data vector $\mathbf{x}_i^j = \langle x_{i1}^j, x_{i2}^j, \ldots, x_{i12}^j \rangle$ 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 jth 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 kth raw metric for the ith module on the jth build.

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

Table III. The measurement baseline.

	Domain				
Metric	1	2	3	Mean	Stdev
Exec	0.041	0.030	0.152	1.51	4.33
NonExec	0.112	0.069	−0.067	3.99	5.30
N_1	−0.206	0.199	0.331	4.46	16.66
η_1	0.002	0.231	−0.134	1.36	2.12
N_2	−0.206	0.199	0.331	4.46	16.66
η_2	−0.131	0.393	−0.139	7.08	10.84
Nodes	0.282	−0.141	−0.029	5.01	7.13
Edges	0.285	−0.144	−0.030	4.74	9.26
Paths	−0.068	−0.215	0.608	24.52	865.59
MaxPath	0.263	−0.121	−0.017	3.66	5.60
AvePath	0.251	−0.123	0.012	3.31	4.65
Cycles	0.269	−0.094	−0.179	0.11	0.50

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; for example, the developer used a '+' instead of a '−'. On other occasions, two or three statements must be modified, added or deleted to remedy a single fault. Furthermore, 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 [19] 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 [4,15].

The following example shows this fault measurement process. Consider the following line of C code:

$$a = b + c \tag{1}$$

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 suppose the design, in fact, required that the difference between b and c be computed:

$$a = b - c \tag{2}$$

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, 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

$$a = c - b \tag{3}$$

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 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:

$$a = 1 + c - b \tag{4}$$

```
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.

This yields 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, 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.

Table IV. Failure report identifying changes. Files associated with the package (IAR-00967) for environment MDS_Env. End of harvest package details for IAR-00967.

File, revision	User	Date	Rev status
MDS_Rep/verification/TestMaster/defaults.dot, 20	rouquett	19/12/2001 16:04	Normal
MDS_Rep/verification/TestMaster/ghsolconfig.dot, 14	rouquett	19/12/2001 16:04	Normal
MDS_Rep/verification/TestMaster/tests/MakeHelper.pm, 7	rouquett	19/12/2001 09:45	Normal
MDS_Rep/verification/TestMaster/tests/UTestHelper.pm, 37	rouquett	19/12/2001 09:48	Normal
MDS_Rep/verification/TestMaster/tests/UTestHelper.pm, 38	rouquett	19/12/2001 16:03	Normal
MDS_Rep/verification/TestMaster/tests/UTestHelper.pm, 39	rouquett	19/12/2001 17:48	Normal
MDS_Rep/verification/TestMaster/tests/UTestHelper.pm, 40	rouquett	19/12/2001 10:02	Normal

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 five, which we then allocate to each of the three modules in this particular source file.

One potential issue is that using 'diff' to count faults in this way will count not only changes to executable code, but also changes to comments and white space. For the development effort we studied, a sample of the total number of changes made across all versions of all modules indicated that only a small fraction of the changes were associated with non-executable code. However, this may be an issue on other development efforts, as coding styles will differ from project to project.

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. In the section of an MDS problem report shown in Table IV 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.

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 [20]. 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 the problem report fragment of Table IV. 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.

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 lines of code (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 ith module m_i^1 on the jth build would be

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

where $\bar{\mathbf{x}}_i^B$ is a vector containing the means of the raw metrics for the baseline system and $\boldsymbol{\sigma}_i^B$ 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.

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 $\boldsymbol{\sigma}_i^B$ of standard deviations for this build. For the purposes of this study, the 1 July 2001 build was chosen as the baseline build. For this study, the actual baseline elements for the 1 July 2001 build are shown in Table III. Again, the columns of this table that are labeled Domain 1, 2 and 3 are, in fact, the transformation matrix for the baseline. The first column of this transformation matrix, for example, contains the coefficients that will send the z-scores of the raw metrics of each module onto the domain score for that module.

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: $m_1, m_2, m_3, \ldots, m_n$. 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. New modules will be added on some builds. Other builds will have some modules eliminated from them. A build list for each build will contain a precise listing of the collection of modules that were incorporated in that build. Recent work by Munson provides more intimate details of the build list management process as it relates to the software measurement process [15].

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: $\mathbf{v}^i = \langle v_1^i, v_2^i, v_3^i, \ldots, v_n^i \rangle$. This build index vector is, in fact, a succinct representation of the build list. It 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 ith module that went to nth 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 nth 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. Recent work by Munson provides details of the measurement process [15].

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. Second, 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 ith domain score of the ath 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

$$\mathrm{X}_{ia}^{L} = \sum_{j=0}^{L-1} \chi_{ia}^{j,j+1}$$

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

To initiate this investigation, we computed domain scores for all of the modules in all of the available builds of the MDS system. All of the module domain scores were baselined relative to the 7 July 2001 build of the system. This build was more or less an intermediate build in the long sequence of builds. Generally, the initial build of any system is incomplete. Therefore, it is a good idea to select as a baseline system a build that will contain the essential structure of the final system.

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 [21]). 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 fault enumeration 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 most recent build of the MDS system. It is important to note that the final build of the MDS system will not contain all of the modules that have been in some of the builds. Modules come and go as the system evolves. Thus, it is important to remove from the fault enumeration all faults that are attributable to modules not in the final, or reference, build. Thus, the fault counts for modules not in the final build vanish together with the module domain churn values when the modules are removed from the evolving builds.

To model the relationship between the fault content of models and the domain metrics, we first 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 there are latent faults that have yet to be discovered. Second, approximately 90% of the modules in the final build have zero fault values. These modules would clearly dominate any regression model that was developed using them. Again, it is our stated objective to model the relationship between measures of software attributes and software faults.

From the subset of modules with a non-zero fault count, we developed three multiple linear regression models, one for each different method of fault enumeration. The dependent variable for each of these models was the module fault count. The module domain churn values were the dependent variables in each of the models. The regression Analysis of Variance (ANOVA) for each of the models is shown in Table V. For each type of fault count, there is a statistically significant relationship between module change activity, as measured by each of the three distinct fault counts, and the module domain churn metrics measures of code evolution.

The actual regression models corresponding to the different types of fault counts are shown in Table VI. The first of these models, the token-based fault counting model, shows that there is a

Table V. Regression ANOVA.

Source	Sum of squares	Df	Mean square	F	Sig.
Token-based fault counts					
Regression	10 091 546	3	3 363 848	293	$p < 0.05$
Residual	6 430 656	560	11 483		
Total	16 522 203	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 VI. Regression models.

	Coefficients	t	Sig.
Token-based fault counts			
(Constant)	18.24	3.5	$p < 0.05$
Domain 1 Churn	21.63	17.3	$p < 0.05$
Domain 2 Churn	−0.59	−0.3	$p > 0.05$
Domain 3 Churn	0.93	0.7	$p > 0.05$
'Sed' command counts			
(Constant)	2.484	14.555	$p < 0.05$
Domain 1 Churn	0.151	3.411	$p < 0.05$
Domain 2 Churn	0.529	6.489	$p < 0.05$
Domain 3 Churn	−0.087	−1.791	$p > 0.05$
Module change counts			
(Constant)	1.200	35.995	$p < 0.05$
Domain 1 Churn	0.009	1.041	$p > 0.05$
Domain 2 Churn	0.143	8.920	$p < 0.05$
Domain 3 Churn	−0.043	−4.483	$p < 0.05$

significant relationship with Domain 1, the control flow domain, only. This was the dominant term in the regression model. Metrics that related to program size, Domain 2, or path complexity, Domain 3, were not significant contributors to this model. This seems to dispel the notion that there is a constant fault density. That is, the bigger the program the more faults that it will contain.

For the two 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 fault counts produced by counting 'sed' commands, Domain 2 is also a significant term

Table VII. Model quality.

Model description	R^2	Adjusted R^2	Std. error of the estimate
Model 1—Token-based	0.61	0.61	107.16
Model 2—‘Sed’ commands	0.19	0.19	3.88
Model 3—Modules changed	0.14	0.14	0.76

in the regression model. Indeed, this term dominates the model insofar as the coefficient for this term in the model is much larger than that of the control flow domain. These numerical comparisons between the two regression coefficients are possible, it must be remembered, in that the domain scores used in the modeling process are, in fact, normalized with the mean of zero and a standard deviation of one. That the size domain figures more heavily in this model is not surprising. The fault-count-dependent variable in this model is derived from editor activity. Larger programs may be naturally expected to generate more editor activity in the fault change amelioration process.

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 is not a significant factor is this model. In the case of this model, the dependent measure is a macro-level measure of fault change activity. Changes in the control flow structure of the program will not be all that visible in this measurement strategy.

The three regression models are very different when we examine their predictive quality of the models as measured by the ratio of the sums of squares due to the regression to the total sums of squares. This statistic is the ratio of sums of squares due to the regression of the sum of total squares. As such, it is a measure of the proportion of total variance in the dependent variable that can be explained by the regression model. Thus, it is a measure of the relative quality of the regression model that we have developed. These data are shown in Table VII. 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 was used in this code analysis. 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. In this context, it is apparent there is a much clearer relationship between measurable software attributes and fault-change activity as measured by token-based change activity. Quite simply, the fault enumeration methods employed in Models 2 and 3 are not well related to measurable software attributes.

It is vital for us to be able to establish a relationship between measures of software attributes and measure of software faults for one simple reason. We can measure software attributes such as lines of code, program operators or total paths. We cannot measure software faults. We can only find software faults as a consequence of their disruptive activity. That is, we simply cannot measure the fault content of a program in advance. This value is known only to nature. Nature will reveal these faults to us only by the school of hard knocks. If we can establish a viable predictive model for software faults based on software attributes that we can measure, then we will have a very good idea which program modules are most fault prone. Of the three different fault-measuring strategies, we can see that the token-based model begins to give us the resolution on fault measurement that we seek.

Within the framework of this investigation, it is evident, then, 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. Again, this is important to us in that we can easily measure software attributes such as program control flow complexity and size complexity. Also, we can easily measure change activity attributable to faults in modules in the token-based fault-measuring scheme. Furthermore, this measurement process can be easily automated for essentially every different configuration of control system. In the token-based module change model shown in Table VI, the dominant factor was measurable changes in the control structure of a module. This factor alone accounted for over 60% of the variance in the criterion measure of software faults. 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 II. This represents a significant step in the understanding of the fault content of program modules.

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.

In addition to predicting the absolute number of faults inserted into a module during its development, the measurements identified in Table I can be used to estimate a module's proportional fault burden. During the earlier stages of implementation, there may be insufficient fault data (e.g. from code checking or testing) to allow the construction of the models predicting the absolute number of faults, such as that shown in the 'Token-based fault counts' section of Table VI. However, the domain scores from the PCA may be combined into a Fault Index (FI) [20]. The number of faults inserted into a module is proportional to the cumulative magnitude of changes to its FI value (e.g., a module having a cumulative change of 600 to its FI will have had twice as many faults inserted into it as a module having cumulative FI change of 300). This means that even before fault data is available to construct regression models, modules can be ordered by the cumulative change to their FI values to: (1) identify the most fault-prone modules; and (2) compare modules in terms of their estimated proportional fault content.

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. Furthermore, 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 practical 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 the 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 the 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.

The question also arises of whether the good correlation for the token-based fault model is simply an artifact of measuring the same thing from two different perspectives, i.e. do the experimental results just indicate that larger modules (with higher domain churns) tend to contain: (1) more faults (and hence more fix reports); or (2) faults that need greater effort to fix (and hence more token counts)? In answer to the first part of this question, our experience with this particular development effort indicates that final module size is not as important an indicator of fault content as is the total amount of measured structural change. For this particular development effort, a smaller module with a greater amount of measured structural change, whether that change was additions or deletions, was likely to have more cumulative faults inserted throughout its development history than a larger module that undergoes comparatively less change throughout its development. In other words, the size of the module is not a significant factor in determining the number of faults inserted into it, but rather the amount of change made to that module during its development. This was also true of a previous but smaller-scale study with the CASSINI on-board control software [2]. However, future work on additional software development efforts will be required before we can say whether this conclusion can be generalized. As far as the second part of the question is concerned, this is also a subject for future work—we did not perform any analyses of the relative ‘size’ of fault repairs versus the size of or cumulative amount of measured change to a module. We hope to perform such analyses on this and other development efforts as part of our future work.

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.

Finally, we are working to extend these types of fault models to earlier phases in the software development lifecycle. In particular, we are interested in the effects of changes in requirements specifications to the number of faults inserted into the implemented system. We have known for a number of years that the cost of identifying and repairing faults during earlier lifecycle activities, such as requirements specification, can be orders of magnitude lower than the cost of doing so during test and operations [22]. The earlier we are able to model software fault content, then, the better we will be able to control overall development cost. In attempting to extend our modeling techniques to the requirements specification phase, the following issues must be dealt with.

Table VIII. Parts of speech tagged by TnT.

TnT symbol	Part of speech	TnT symbol	Part of speech	TnT symbol	Part of speech
.	End of sentence	NNP	Proper noun, singular	UH	Interjection
CC	Coordination conjunction	NNPS	Proper noun, plural	VB	Verb, base form
CD	Cardinal number	NNS	Noun, plural	VBD	Verb, past tense
DT	Determiner	PDT	Predeterminer	VBG	Verb, gerund or present participle
EX	Existential ‘there’	POS	Possessive ending		
FW	Foreign word	PRP	Personal pronoun	VBN	Verb, past participle
IN	Preposition of subordinating conjunction	PRPS	Possessive pronoun	VBP	Verb, non-3rd person singular present
JJ	Adjective	RB	Adverb	VBZ	Verb, 3rd persona singular present
JJR	Adjective, comparative	RBR	Adverb, comparative	WDT	‘Wh’-determiner
JJS	Adjective, superlative	RBS	Adverb, superlative	WP	‘Wh’-pronoun
LS	List item marker	RP	Participle	WPS	Possessive ‘Wh’-pronoun
MD	Modal	SYM	Symbol	WRB	‘Wh’-adverb
NN	Noun, singular or mass	TO	‘To’		

- Many requirements are written in natural language, making it more difficult to measure their structural attributes than is the case with source code.
- In many cases, a baseline against which changes can be measured is not readily identifiable. While source code is almost always well controlled with a version control system such as Revision Control System (RCS) or Concurrent Versions System (CVS), it is our experience that the rigor of requirements change control varies widely from project to project.

Natural language processing techniques developed over the past several years provide a way of dealing with the first issue. When applied to requirements text, natural language parsers, noun- and verb-phrase chunkers, and parts-of-speech taggers produce results that can be measured in ways analogous to the outputs of source code analyzers. As an experiment, we applied the Trigrams’n’Tags (TnT) parts-of-speech tagger [23] to a set of 523 requirements for a JPL space mission software system. For each word in a text set, TnT identifies it as one of the parts of speech shown in Table VIII. The Penn Treebank Project’s part-of-speech tagging guidelines describe the parts-of-speech tags [24]. For each requirement, the TnT analyzer allowed us to count the number of occurrences of

Table IX. Sample requirements text and parts-of-speech sequences.

Requirements text sequence	Parts of speech sequence
The spacecraft shall be capable of issuing a safing command to the payload and disable the payload upon entering safe mode	DT NN MD VB JJ IN VBG DT JJ NN TO DT JJ CC JJ DT NN IN VBG JJ NN
Except during moi the spacecraft shall exit safe mode by ground command only	IN IN FW DT NN MD VB JJ NN IN NN NN RB
Within four hours of being commanded out of safe mode, the orbiter shall be capable of returning to an operational state that allows the orbiter to resume planned activities	IN CD NNS IN VBG VBN RP IN JJ NN DT NN MD VB JJ IN VBG TO DT JJ NN WDT VBZ DT NN TO VB JJ NNS
After achieving the primary science orbit the spacecraft shall be capable of providing the electra with an average of 25 Whrs of energy per contact, which reduces the 189 W allocated power to the instruments whenever electra is operating for data relay	IN VBG DT JJ NN NN DT NN MD VB JJ IN VBG DT NN IN DT NN IN CD NNS IN NN IN NN WDT VBZ DT CD NN VBN NN TO DT NNS WRB NN VBZ VBG IN NNS NN
While in cruise to Mars the orbiter shall be capable of providing 151 W including all margins for the payload elements. The requisite 11 W of power for providing the uso signal to the spacecraft shall be deducted from the payloads 151 W allocation	IN IN NN TO VB DT NN MD VB JJ IN VBG CD NN VBG DT NNS IN DT JJ NNS. DT JJ CD NN IN NN IN VBG DT JJ NN TO DT NN MD VB VBN IN DT NNS CD NN NN
While in aerobraking after reaching an apoapsis of 1000 km the spacecraft shall be capable of providing 91 W including all margins for the payload elements. The requisite 11 W of power for providing the uso signal to the spacecraft shall be deducted from the payloads 91 W allocation	IN IN VBG IN VBG DT NN IN CD NN DT NN MD VB JJ IN VBG CD NN VBG DT NNS IN DT JJ NNS. DT JJ CD NN IN NN IN VBG DT JJ NN TO DT NN MD VB VBN IN DT NNS CD NN NN
While in aerobraking prior to reaching an apoapsis of 1000 km the spacecraft shall be capable of providing 101 W including all margins for the payload elements. The requisite 11 W of power for providing the uso signal to the spacecraft shall be deducted from the payloads 101 W allocation	IN IN VBG RB TO VBG DT NN IN CD NN DT NN MD VB JJ IN VBG CD NN VBG DT NNS IN DT JJ NNS. DT JJ CD NN IN NN IN VBG DT JJ NN TO DT NN MD VB VBN IN DT NNS CD NN NN

each part of speech. Examples of requirements text and the corresponding sequences of parts-of-speech tags are shown in Table IX. We then standardized each parts-of-speech count and performed a principal components analysis, the results of which are shown in Table X. The first domain is associated with the most commonly-used parts of speech, including nouns, verbs, determiners, (e.g., a, the, every, another, etc.) and adjectives. Progressively less-frequently used parts of speech, such as 'Wh-adverbs (e.g., how, where, why, etc.)', personal pronouns and symbols, are associated with the remaining domains. We see immediately that there are substantially more orthogonal domains for measurements of requirements

Table X. Requirements text PCA rotated loading matrix (Varimax, Gamma = 1.0).

	Domain								
Metric	1	2	3	4	5	6	7	8	9
NN	**0.878**	0.058	0.038	−0.038	0.172	−0.117	0.099	0.096	0.006
IN	**0.867**	0.126	0.012	−0.057	0.141	−0.121	0.01	0.084	0.095
VB	**0.846**	−0.14	−0.031	−0.006	−0.026	0.005	0.029	−0.015	0.145
DT	**0.837**	0.092	0.011	0.001	0.171	−0.046	0.117	0.132	0.076
JJ	**0.809**	−0.002	0.042	0.066	0.126	0.038	0.054	0.113	0.022
MD	**0.804**	−0.209	0.023	−0.045	−0.002	−0.029	0.091	−0.123	0.202
NNS	**0.779**	0.204	0.136	−0.017	0.03	0.03	−0.032	0.048	0.079
TO	**0.686**	−0.05	−0.121	0.147	−0.037	0.069	0.049	0.296	0.086
VBN	**0.671**	0.334	0.019	0.013	0.01	0.096	0.043	0.052	0.08
CC	**0.62**	0.266	0.049	−0.093	−0.026	−0.155	−0.146	0.051	−0.119
PRPS	**0.579**	−0.047	0.01	0.091	−0.204	0.102	0.067	−0.129	0.503
CD	**0.564**	0.055	0.166	−0.01	0.259	−0.253	0.066	0.029	−0.193
EX	**0.509**	−0.115	−0.14	0.028	−0.363	−0.069	−0.032	−0.144	0.255
WRB	0.022	**0.724**	−0.038	−0.065	0.142	−0.057	0.008	−0.137	0.146
VBZ	0.34	**0.573**	−0.066	0.096	0.096	−0.239	0.283	0.318	0.015
PRP	0.215	**0.523**	−0.063	−0.031	−0.131	0.272	−0.148	0.061	−0.449
RBR	−0.008	0.004	**0.747**	0.18	−0.122	0.056	−0.058	0.002	−0.048
PDT	0.065	−0.093	**0.747**	−0.119	0.059	0.038	0.107	−0.09	0.02
SYM	0.032	−0.066	−0.006	**0.782**	0.24	0.04	−0.018	−0.191	−0.14
FW	−0.018	0.024	0.065	**0.753**	−0.06	−0.076	−0.028	0.224	0.134
WDT	0.099	0.088	−0.155	0.201	**0.624**	0.094	0.057	−0.066	0.077
JJR	0.08	0.095	−0.08	−0.087	0.098	**−0.78**	−0.023	−0.053	0.046
WPS	0.023	−0.036	−0.015	0.091	−0.137	**−0.571**	−0.032	0.004	−0.097
RBS	0.03	0.144	−0.022	0.051	−0.139	0.015	**0.8**	−0.013	−0.021
NNP	0.146	−0.122	0.074	−0.11	0.201	0.046	**0.684**	0.065	−0.077
RP	0.093	−0.071	−0.109	0.007	0.083	0.093	0.038	**0.802**	−0.063
JJS	0.186	0.121	−0.062	−0.009	−0.01	0.042	−0.097	−0.032	**0.698**
VBG	0.37	0.066	0.115	−0.01	0.262	0.13	−0.107	0.28	**0.482**
RB	0.411	0.071	0.233	0.069	−0.134	−0.337	0.037	0.458	0.119
VBD	0.296	−0.268	−0.059	−0.186	0.321	0.009	0.004	0.098	−0.087
VBP	0.213	0.139	0.359	0.012	0.46	−0.075	−0.053	0.239	0.032
‘Variance’ explained	7.755	1.650	1.471	1.402	1.351	1.389	1.328	1.424	1.500
Percent variance explained	25.015	5.321	4.746	4.521	4.357	0.481	4.285	4.594	4.837
Cumulative percent variance explained	25.015	30.336	35.082	39.603	43.96	44.441	48.726	53.32	58.157

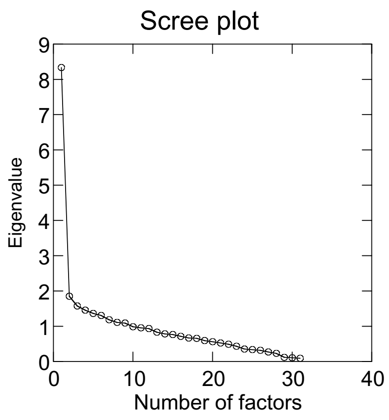

Figure 2. Scree plot for requirements text measures PCA.

structure than there were for source code; using the same PCA stopping rule as was used for the analysis of source code structure (stop when eigenvalues < 1), nine components can be extracted before stopping the analysis. Furthermore, with the exception of the first component, which accounts for about 25% of the variance in the rotated loading matrix, the other components account for substantially smaller, and much more similar, amounts of variance—this is seen in the scree plot shown in Figure 2 as well as the last two rows of Table X. If we are able to develop fault models from these types of measurements, these results indicate that they will be substantially more complicated than those we have developed for source code.

In developing fault models using measures of requirements' structural attributes, we also have to resolve the second issue mentioned above, namely that change management and version control of requirements is not always as rigorous as that for source code. It is not always feasible to identify a baseline set of requirements against which changes can be measured, and it may also be the case that all changes to requirements during their specification are captured. We are currently searching for one or more software development efforts that have instituted rigorous change management and version control measures for specifications, and hope to report further developments along these lines within the next year.

ACKNOWLEDGEMENTS

The work described in this paper was carried out at the JPL, 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

1. Munson J, Nikora A. Estimating rates of fault insertion and test effectiveness in software systems. *Proceedings of the 4th ISSAT International Conference on Reliability and Quality in Design*, August 1998. International Society of Science and Applied Technologies: New Brunswick, NJ, 1998; 263–269.
2. Nikora A, Munson J. Determining fault insertion rates for evolving software systems. *Proceedings of the 1998 IEEE International Symposium of Software Reliability Engineering*, Paderborn, November 1998. IEEE Press: Piscataway, NJ, 1988; 306–315.
3. Nikora A, Munson J. Developing fault predictors for evolving software systems. *Proceedings of the 9th International Software Metrics Symposium*, Sydney, September 2003. IEEE Press: Piscataway, NJ, 2003; 338–350.
4. Munson J, Nikora A. Toward a quantifiable definition of software faults. *Proceedings of the 13th IEEE International Symposium on Software Reliability Engineering*, Annapolis, MD, November 2002. IEEE Press: Piscataway, NJ, 2002; 388–395.
5. Dvorak D, Rasmussen R, Reeves G, Sacks A. Software architecture themes in JPL's mission data system. *AIAA Space Technology Conference and Exposition*, Albuquerque, NM, September 1999. American Institute of Aeronautics and Astronautics: Reston, VA, 1999.
6. Khoshgoftaar T, Allen E. Modeling software quality with classification trees. *Recent Advances in Reliability and Quality Engineering* (*Lecture Notes in Computer Science*, vol. 1845), ch. 15, Pham H (ed.). World Scientific: Singapore, 2001; 247–270.
7. Gokhale S, Lyu M. Regression tree modeling for the prediction of software quality. *Proceedings of the 3rd ISSAT International Conference on Reliability and Quality in Design*, Anaheim, CA, March 1997. International Society of Science and Applied Technologies: New Brunswick, NJ, 1997; 31–36.
8. Schneidewind N. Software metrics model for integrating quality control and prediction. *Proceedings of the 8th International Symposium on Software Reliability Engineering*, Albuquerque, NM, November 1997. IEEE Press: Piscataway, NJ, 1997; 402–415.
9. Khoshgoftaar T. An application of zero-inflated poisson regression for software fault prediction. *Proceedings of the 12th International Symposium on Software Reliability Engineering*, Hong Kong, November 2001. IEEE Press: Piscataway, NJ, 2001; 66–73.
10. Schneidewind N. Investigation of logistic regression as a discriminant of software quality. *Proceedings of the 7th International Software Metrics Symposium*, London, April 2001. IEEE Press: Piscataway, NJ, 2001; 328–337.
11. *IEEE Standard Glossary of Software Engineering Terminology*. IEEE Std 729-1983, Institute of Electrical and Electronics Engineers, 1983.
12. *IEEE Standard Dictionary of Measures to Produce Reliable Software*. IEEE Std 982.1-1988, Institute of Electrical and Electronics Engineers, 1989.
13. Nikora A, Munson J. Finding fault with faults: A case study, with J. Munson. *Proceedings of the Annual Oregon Workshop on Software Metrics*, Coeur d'Alene, ID, May 1997. OCATE: Portland State University, Portland, OR, 1997.
14. The Darwin software engineering measurement appliance, Cylant Technology, 2000.
15. Munson J. *Software Engineering Measurement*. CRC Press: Boca Raton, FL, 2002.
16. Munson J, Khoshgoftaar T. Regression modeling of software quality. *Information and Software Technology* 1990; **32**(2):105–114.
17. Hall G, Munson J. Software evolution: Code delta and code churn. *Journal of Systems and Software* 2000; **54**(2):111–118.
18. Dillon W, Goldstein M. *Multivariate Analysis: Methods and Applications*. Wiley Interscience: New York, 1984.
19. Ximbiot. CVS-Concurrent Versions System v.1.11.21.
http://ximbiot.com/cvs/manual/cvs-1.11.21/cvs.html [10 March 2006].
20. Nikora A, Munson J. Understanding the nature of software evolution. *Proceedings of the International Conference on Software Maintenance*, Amsterdam, September 2003. IEEE Press: Piscataway, NJ, 2003; 83–93.
21. Computer Associates. AllFusion Harvest Change Manager Features, Descriptions & Benefits.
http://www3.ca.com/Files/FactSheet/af_harvest_cm_fdb.pdf [11 February 2002].
22. Boehm B. *Software Engineering Economics*. Prentice-Hall: Englewood Cliffs, NJ, 1981.
23. Brants T. TnT—a statistical part-of-speech tagger. *Proceedings of the Sixth Applied Natural Language Processing Conference ANLP-2000*, Seattle, WA, April 2000. Association for Computational Linguistics: East Stroudsburg, PA, 2000; 224–231.
24. Santorini B. Part-of-speech tagging guidelines for the penn treebank project (3rd revision, 2nd printing).
http://www.cis.upenn.edu/~treebank/home.html [February 1995].