

# Understanding the Nature of Software Evolution

Allen P. Nikora  
Jet Propulsion Laboratory,  
California Institute of Technology  
Pasadena, CA 91109-8099  
[Allen.P.Nikora@jpl.nasa.gov](mailto:Allen.P.Nikora@jpl.nasa.gov)

John C. Munson  
Computer Science Department  
University of Idaho  
Moscow, ID 83844-1010  
[jmunson@cs.uidaho.edu](mailto:jmunson@cs.uidaho.edu)

## ABSTRACT

*Over the past several years, we have been developing methods of measuring the change characteristics of evolving software systems. Not all changes to software systems are equal. Some changes to these systems are very small and have low impact on the system as a whole. Other changes are substantial and have a very large impact of the fault proneness of the complete system. In this study we will identify the sources of variation in the set of software metrics used to measure the system. We will then study the change characteristics to the system over a large number of builds.*

*We have begun a new investigation in these areas in collaboration with a flight software technology development effort at the Jet Propulsion Laboratory (JPL) and have progressed in resolving the limitations of the earlier work in two distinct steps. First, we have developed a standard for the enumeration of faults. This new standard permits software faults to be measured precisely and accurately. Second, we have developed a practical framework for automating the measurement of these faults. This new standard and fault measurement process was then applied to a software system's structural evolution during its development. Every change to the software system was measured and every fault was identified and tracked to a specific code module. The measurement process was implemented in a network appliance, minimizing the impact of measurement activities on development efforts and enabling the comparison of measurements across multiple development efforts.*

*In this paper, we analyze the measurements of structural evolution and fault counts obtained from the JPL flight software technology development effort. Our results indicate that the measures of structural attributes of the evolving software system are suitable for forming predictors of the number of faults inserted into software modules during their development, and that some types of change are more likely to result in the insertion of faults than others. The new fault standard also insures that the model so developed has greater predictive validity.*

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

As software systems change over time, it is very difficult to understand and measure the effect of the changes. We would like to be able to describe, numerically, the way that each system increment, or build, is different from its successor and its predecessor. This is a very complex problem in that most modern software systems consist of thousands of program modules on each of which there may be as many as 20-30 distinct metrics collected. For any one build, there may be tens of thousands of metrics collected on a typical large system. Knowing what to measure, how to measure, and when to measure will be a key step in understanding the software evolution process.

As programs have increased in length several orders of magnitude in the last three decades, the problems associated with measuring these programs have also increased several orders of magnitude. Furthermore, these systems experience a very large number of changes during their development and

early deployment. Not all changes will have the same relative impact on the code in terms of its overall complexity. Some changes by their very nature will be innocuous, such as the introduction of comment statements. Other changes will make substantial changes to the basic architecture of a program module. Even the simplest measurements taken on each program module have a way of creating an enormous data management problem in the measurement of evolving systems. The key to the success of the measurement problem is to reduce the size of the problem with which we are working.

In this paper we will examine the measurement of the software from the standpoint of the measurement of software physical attributes and software quality attributes. We will then discuss the most critical issue of all in this measurement process. That is, how do we compare different versions of an evolving system in a meaningful way? We will also examine methods for reducing the huge volume of measurement data to information that can be used in the management of the software development and test activities.

## 2. Related Work

Over the past several years, a great deal of work has been done in the area of using measurements of software systems to identify fault-prone components and predict their fault content. Examples of this work include the classification methods proposed by Khoshgoftaar and Allen [Khos01a] and by Ghokale and Lyu [Ghok97], Schneidewind's work on Boolean Discriminant Functions [Schn97], Khoshgoftaar's application of zero-inflated Poisson regression to predicting software fault content [Khos01], and Schneidewind's investigation of logistic regression as a discriminant of software quality [Schn01]. Each of these efforts has provided useful insights into the problem of identifying fault-prone software components prior to test. The one thing that these efforts have in common is that each of them analyzed a snapshot of the subject system, rather than examining its evolution during development. This may limit the validity of those efforts' conclusions to the point in the development life cycle when the measurements were made. If, however, the entire evolution of a software system is analyzed, any conclusions that are reached should be applicable to any point in the development cycle of the artifact being studied. With this goal in mind, we conducted a small study on a JPL flight system several years ago [Niko98]. We found strong indications that measurements of a system's structural evolution could serve as predictors of the fault insertion rate. However, this study had two limitations:

- The study was relatively small – fewer than 50 observations were used in the regression analysis relating the number of faults inserted to the amount of structural change.
- The definition of faults that was used was not quantitative. The ad-hoc taxonomy, first described in [Niko97],

was an attempt to provide an unambiguous set of rules for identifying and counting faults. The rules were based on the types of changes made to source code in response to failures reported in the system. Although the rules provided a way of classifying the faults by type, and attempted to address faults at the level of individual modules, they were not sufficient to enable repeatable and consistent fault counts by different observers to be made. The rules in and of themselves were unreliable.

Before recommending the use of measurements of structural evolution as a fault predictor, we needed to address the limitations of the earlier study. Our main concern was developing a quantitative definition of faults, so that we could automate what had been a time-consuming manual activity in the earlier study, the identification and counting of repaired faults at the module level. Our hope was that this would provide us with unambiguous, consistent, and repeatable fault counts, as well as a substantially larger number of observations than the earlier study.

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

- The structural evolution of a system as it changes over a series of builds
- The number of faults discovered during the system's development.

Measuring a system's structural evolution is a straightforward activity – the DARWIN™ network appliance can automatically make these measurements if it has access to a software development effort's source code repository. DARWIN™ will then take structural measurements of each version of each module (i.e., function or method) in the system and use those measurements to produce quantitative reports of the system's evolutionary history according to the techniques described in Sections 6 and 7.

Measuring faults is not quite as straightforward an activity. The structure of a software component can easily be made because there are standard, quantitative definitions of structural attributes (e.g., number of physical lines of code, number of operators) that can be used to develop measurement tools. The following definition of what constitutes a fault is typical of that provided by current standards: "A manifestation of an error in software. A fault, if encountered, may cause a failure" [IEEE88, IEEE83]. This establishes a fault as a structural defect in a software system that underlies the failure of that system to operate as expected, but does not help in determining the type of failure that was observed, or establish how individual faults may identified or measured. Some standards address the issue of the type of failure observed by describing schemes for classifying anomalies recorded during software development and operation. For instance, [IEEE93] provides details of an anomaly classification process, as well as criteria for classifying the type of anomaly observed, at what point in the development process the anomaly was observed, and the action taken in response to the anomaly. For example, Table 3c in this standard al-

lows classification of the type of behavior exhibited by the anomaly (e.g., “precision loss”) or the type of defect that led to the anomaly (e.g., “referenced wrong data variable”). This type of scheme is helpful in determining the underlying causes of faults and failures, so that the development process may be modified to 1) identify the types of faults on which fault detection and removal resources should be focused for the current development effort, and 2) minimize the introduction of the most common types of faults in future development tasks. However, classification standards do not provide enough information to help count the number of faults in the system. Returning to Table 3c of [IEEE93], we see that some of the anomaly types can readily be traced to a single fault (e.g., “Operator in equation incorrect”). However, the response an “I/O Timing” anomaly may involve changes to many lines of source code spread across multiple source code files. In this case, the standard does not provide enough information to allow counting the number of faults at the module level.

Orthogonal Defect Classification (ODC), initially reported in 1992 [Chi92], provides a framework for 1) identifying defect types and the sources of error in a software development effort, 2) determining the effectiveness of the different defect detection techniques and strategies used by the organization, and 3) using the feedback provided by analysis of the defects to help the organization reduce the number of defects it inserts into its systems. Like [IEEE93], ODC provides a scheme for classifying defects, which is useful in identifying sources of error at different points in the development process. However, it does not seem to be possible to use the classification scheme to consistently count faults at the module level. The recognition process for defects is not sufficiently well defined to permit the automatic recognition of these defects.

### 3. Problem Statement

The objective of our current work is to develop practical methods of measuring software evolution. This will involve the establishment of a baseline reference artifact or system against which all system builds will be calibrated. Although other types of artifacts could have been analyzed, working with source code has two advantages:

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

Through the analysis of the structural evolution of a software system, we overcome the limitations of the related work identified in Section 2 – that is, any predictors of fault content we develop should have predictive validity at any point during the development of the artifact being studied. This is in contrast to models developed from single and iso-

lated system builds. In a more general sense, we wish to understand the complete system from its first build until the most recent build.

We worked in collaboration with the Mission Data System (MDS), a mission software technology development effort in progress at JPL. We were able to measure the structural evolution of the MDS during the development of a specific release. We also were able to measure the fault discovery process by our new fault measurement methodology [Muns02]. For every failure reported against the MDS, we were also able to identify the changes made to each module in response to that failure, and thereby count the number of faults that had been repaired.

### 4. A Description of the Mission Data System

The brief description of the MDS provided here is summarized from Dvorak, et al. [Dvo99]. Until recently, planetary exploration missions were spaced years apart, with little attention to software reuse, given the rapid pace of computer technology and computer science. Also, since radiation-hardened flight computers remain years behind their commercial counterparts in speed and memory, flight software has typically been highly customized and tuned for each mission. In order to use software engineering resources more effectively and to sustain a quickened pace of missions, JPL initiated the MDS project in April 1998 to define and develop an advanced multi-mission architecture for an end-to-end information system for deep-space missions. MDS is aimed at several institutional objectives: earlier collaboration of mission, system and software design; simpler, lower cost design, test, and operation; customer-controlled complexity; and evolvability to in situ exploration and other autonomous applications.

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

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

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

analysis; these problem reports provided the information from which the number of repaired faults was computed.

## 5. Metrics Used in This Study

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

We have obtained measurement data from the Darwin<sup>TM</sup> system on the target software system [Cyla03]. These measurement data were obtained by checking out each build of the system from the configuration control system and then applying the measurement tool incorporated in the Darwin<sup>TM</sup> Network Appliance.

### 5.1. Static Metrics

The specific metrics used in this study are listed in Table 1. These metrics were obtained for both the C and the C++ code modules in the MDS system. The precise definition of each of these metrics and the standard used to measure them can be found in Munson [Muns03].

**Table 1 - Metrics Used in This Study**

Metric	Definition
Exec	Number of executable statements
NonExec	Number of non-executable statements
$N_1$	Total operator count
$\eta_1$	Unique operator count
$N_2$	Total operand count
$\eta_2$	Unique operand count
Nodes	Number of nodes in the module control flow graph
Edges	Number of edges in the module control flow graph
Paths	Number of paths in the module control flow graph
MaxPath	The length of the path with the maximum edges
AvePath	The average length of the paths in the module control flow graph
Cycles	Total number of cycles in the module control flow graph

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

### 5.1. Derived Metrics

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

First we will need to identify the distinct sources of variance. We will use principal components analysis to identify these new measurement domains. The results of this analysis are shown in Table 2.

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

**Table 2 - The Principal Components Analysis**

Metric	Domain		
	1	2	3
Exec	.60	.49	.47
NonExec	.64	.53	.18
$N_1$	.28	.64	.65
$\eta_1$	.49	<b>.70</b>	.07
$N_2$	.28	.64	.65
$\eta_2$	.35	<b>.90</b>	.04
Nodes	<b>.87</b>	.31	.27
Edges	<b>.88</b>	.31	.27
Paths	.17	-.10	<b>.89</b>
MaxPath	<b>.87</b>	.35	.29
AvePath	<b>.86</b>	.34	.33
Cycles	<b>.67</b>	.22	-.02
Eigenvalues	4.79	3.13	2.24

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

For measurement purposes, it will be necessary to standardize all original or raw metrics so that they are on the same relative scale. For the  $i^{\text{th}}$  module  $m_i^j$  on the  $j^{\text{th}}$  build of the system there will be a data vector  $\mathbf{x}_i^j = \langle x_{i1}^j, x_{i2}^j, \dots, x_{i12}^j \rangle$  of

12 raw complexity metrics for that module. We can standardize each of the raw metrics by subtracting the mean  $\bar{x}_1^j$  of the metric #1 over all modules in the  $j^{\text{th}}$  build and dividing by its standard deviation  $\delta_1^j$  such that  $z_{ii}^j = \frac{x_{ii}^j - \bar{x}_1^j}{\delta_1^j}$  repre-

sents the standardized value of the first raw metric for the  $i^{\text{th}}$  module on the  $j^{\text{th}}$  build.

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

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

In order to simplify the structure of software complexity even further than the orthogonal domains produced by the principal components analysis it would be useful if each of the program modules in a software system could be characterized by a single value representing some cumulative measure of complexity. Previous research has established that the fault index, FI, has properties that might be useful in this regard. The FI metric is a weighted sum of a set of uncorrelated attribute domain metrics[Muns03]. This metric represents each raw metric in proportion to the amount of unique variation contributed by that metric.

The FI of the factored program modules may be represented as follows:

$$FI = \sum_j l_j d_{ji}$$

where  $l_j$  is the eigenvalue associated with the  $j^{\text{th}}$  factor and  $d_{ji}$  is the  $j^{\text{th}}$  factor score of the  $i^{\text{th}}$  program module on the  $j^{\text{th}}$  domain. Each of the eigenvalues represents the relative contribution of its associated domain to the total variance explained by all of the domains. In essence, then, the FI metric is a weighted sum of the individual domain metrics. In this context, the FI metric represents each raw complexity metric in proportion to the amount of unique variation contributed by that complexity metric.

The FI metric has a mean of zero and a variance proportional to the variance of the eigenvalues. To make it more meaningful, we have employed a simple transformation on FI to adjust it so that it has a mean of 100 and a standard deviation of 10. Let  $FI'$  represent this transformed measure. Then

$$FI' = FI \times 10 + 100.$$

## 6. The Measurement Baseline

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

As a software system changes over time, it is very difficult to understand and measure the effect of the changes. We would like to be able to describe, numerically, the way that each system increment, or build, is different from its successor and its predecessor. This is a very complex problem in that we are obtaining twelve measures on each program module. For any one build, there are tens of thousands of metrics collected on our target system.

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

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

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

As program modules change from one build to another, the attributes of the modified program modules change. This means that there are measurable changes in modules from one build to the next. Each build is numerically and measurably different from its predecessor with respect to a particular set of metrics. Thus, there is no such thing as measuring a software system but once. Many software developers who profess to be deeply committed to measurement are still tempted to represent a system by a set of measurements taken at one point in a system's evolution. The truth is, measurement is a process. Whenever changes are made to a system, those system elements that have changed must be re-measured.

In order to describe the complexity of a system at each build, it will be necessary to know the version of each of the modules that was in the program that failed. Each of the program modules is a separate entity. It will evolve at its own rate. Each build of the system will unify a set of program modules. Not all of the builds will contain precisely the same modules. Clearly there will be different versions of some of the modules in successive system builds. This complex process is described in detail in [Muns03].

**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
$\eta_1$	.002	.231	-.134	1.36	2.12
$N_2$	-.206	.199	.331	4.46	16.66
$\eta_2$	-.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

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

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

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

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  $\delta_i^B$  of standard deviations for this build. For the purposes of this study, the July 1, 2001 build was chosen as the baseline build. Table 3 shows the actual baseline that will be used to compute the derived metrics used in this study.

## 7. Measuring Change Activity

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.

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

We can represent the build configuration in a nomenclature that will permit us to describe the measurement process more precisely by recording module version numbers as vector elements in the following manner:  $v^j = \langle v_1^j, v_2^j, v_3^j, \dots, v_m^j \rangle$ . 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  $i^{\text{th}}$  module that went to  $n^{\text{th}}$  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  $n^{\text{th}}$  build. In this case the cardinality of the complete set of modules is represented by the index value  $m$ . This is also the number of modules in the set of all modules that have ever entered any build.

The management of the configuration of each of the program modules is one aspect of the software management process. Another vital piece is the build index vector. It is the only record of the module version that went to each build. This build index vector must be maintained in some type of a build management database. There are many sad stories in

the software maintenance community about software systems that have been delivered to a customer without such a record. It is almost impossible to interpret trouble reports from customers if the structure of the build that the customer is using is not known.

A natural way to capture the intermediate measurements for each build would be to incorporate the measurement tools within the configuration management system. Just as code deltas are maintained for each program module, so should deltas for the code attributes also be kept by the configuration management system.

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

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

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

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

Between build  $i$  and  $j$  module  $m_3$  was removed giving.

Thus,

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

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

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

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

the total amount of change the system has undergone across all of the builds to date.

It is now possible to compute the total domain change activity for the aggregate system within each of the system builds. Let  $d_{ia}^{B,j}$  represent the  $i^{\text{th}}$  domain score of the  $a^{\text{th}}$  module on build  $j$  baselined by build  $B$ . The total domain value of the system on build  $j$  on domain  $i$  is the sum of the domain scores for each of the modules present in this build.

This system domain value  $D_i^j$  is given by

$$D_i^j = \sum_{a \in V^j} d_{ia}^{B,j}.$$

We can now measure the nature of the change activity from one build to the next on each of the orthogonal domains. We will come to understand that not all changes are equal. Some change activity will increase the complexity of the program on the control flow attribute domain while other change activity may be neutral with respect to this domain but increase the size of the program.

It is also of interest to understand the precise nature of change activity on each of the program modules on each of the builds. We can establish the precise change activity of any module on any domain by the domain churn measure.

This measure of domain churn,  $\chi$ , for module  $m_a$  between any two sequential builds is simply  $\chi_{ia}^{j,j+1} = |d_{ia}^{B,j} - d_{ia}^{B,j+1}|$ .

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

It is now possible to compute the total domain change activity for the aggregate system within each of the system builds. The total domain change activity (churn),  $X_i^{j,j+1}$ ,

of the system for module  $m_a$  on domain  $i$  for build  $j+1$  is the sum of the domain churn for this module across all modules in the build  $j+1$ .

$$X_i^{j,j+1} = \sum_{a \in V^{j+1}} \chi_{ia}^{j,j+1}.$$

The value of the domain churn for each module is, of course, dependent on the referent baseline build  $B$ .

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

## 8. Measuring and Understanding Change

We have developed two distinct measures of software evolution. First, there is the FI metric. This metric will permit us to understand the essential complexity of any one module in relation to any other module in the entire evolution of the software system. Thus, if we observe a module  $m_a$  on

the  $j^{th}$  with a value of 100, we know that this module is equal to an average module on the baseline build.

There are many different levels of granularity of observation that may be made on an evolving software system. It is our objective, in this paper, to characterize each build of the system as a whole. This will give us a management perspective of the evolving system. There are several ways that we can accomplish this goal. Let us begin by examining the system FI. To do this, we will add all of the FI values for each module in each system build. This will yield the total FI value for the system. In that the average value of FI was adjusted to 100 for the baseline build, then the system FI should simply be the product of the number of program modules in that system times 100. Thus, if there were 15,000 modules on the baseline build, then the system FI would be 1,500,000.

The system FI values for the MDS system are shown in Figure 1 for 421 builds of this system. The baseline build for this system was set at build number 174. For the purposes of clarity the values have been normalized to the FI of the last build so that the largest system FI value will be scaled to 100.

What is astonishing about this evolutionary sequence is that there is no point of inflection wherein the evolution of the system begins to slow down. If anything, there is an apparent point of inflection at about build 240 wherein the slope of the line increases.

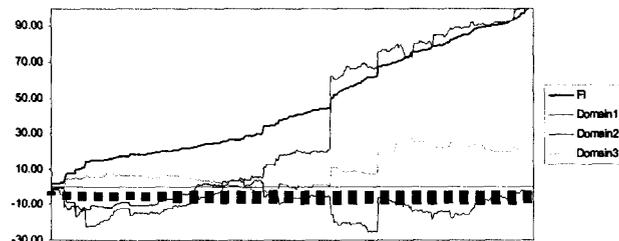


Figure 1. The System Overview

The FI value is, in the last analysis, a composite of the individual domain scores. In this case there are three orthogonal domains. Again, Domain 1 represents the control flow complexity of the MDS system, Domain 2 represents the size complexity of the modules as represented by their operator and operand counts, and Domain 3 represents the path complexity of the modules. For each of the builds, a composite domain value was computed. This was obtained by summing all of the domain scores for each of the program modules in the build for each of the domains. The composite domain scores for each of the builds are shown in Figure 1 together with the system FI values. As was the case for the FI values, the composite domain scores were normalized to the largest domain value that was the composite domain value for Domain 1 on the last build. All of the composite domain values were then adjusted to a maximum value of 100 for presentation purposes. Again, the baseline

system was chosen to be build 174. Any other build could have been used and equivalent results would have been obtained.

It is clear from examination of Figure 1, that not all builds are equivalent. Each build may be characterized by the specific nature of changes that it induces on each of the three orthogonal measurement domains. The baseline build represents a relatively arbitrary point in the evolutionary sequence. We can conclude, from Figure 1, that a substantial part of the change activity since the baseline build has been in Domain 1. That is the control structure of the program is changing much more rapidly than its size or path complexity. It is also apparent that there was a massive change in the program between builds 246 and 247. The specific nature of this change is quite obvious. There was a great increase in the control complexity, a slight increase in path complexity, and a decrease in the size domain. Oddly enough, this is a relatively common occurrence in the evolution of a system. It generally happens when there is a massive rewrite or redesign of the system to control for its growth in size. The problem is, change is a multivariate activity. Change activity must be understood in all of the attribute domains of a system.

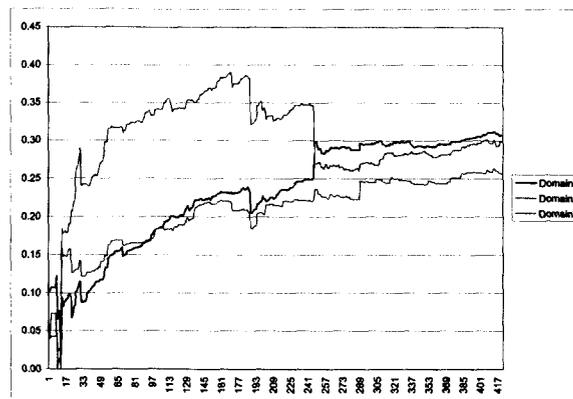


Figure 2. Domain Churn

Yet another way that change may be characterized is by examination of the domain churn of the system during its evolution. The domain churn data for the MDS system are shown in Figure 2. These data present a different view of the change activity. As was the case for the cumulative change data, we see a substantial amount of change activity in the between builds 246 and 247. What is striking from this analysis is that there was a rather substantial decrease in the size complexity of the program, a view not nearly so obvious from the cumulative change data. It is also apparent that the relative rate of change, as measured by domain churn, has slowed down substantially after build 247.

Another interesting perspective that domain churn gives us, is the degree of change activity during the initial stages of program development. We can see that there is considerable fluctuation in the several domains during this initial period. After build 247, the size component is no longer the dominant

ing feature of domain churn. They program has reached a relatively stable change plateau for size. Now, however, the dominant feature is the churn in the control complexity. Unfortunately, as we shall see, control complexity is highly correlated with fault counts.

The bottom line is that not all system changes are equal. Through the use of principal components analysis on the raw metric data for the evolving software, we are able to isolate the orthogonal sources of software variation, software attributes, and analyze the specific nature of the change activity.

### 9. The Relationship Between Change Activity and Software Quality Attributes

Generally we measure with a purpose. In general, we really don't want to know how many lines of code a system has. We really aren't interested in the number of paths that a module might have. However, we can easily measure these things. What we really wish to know is something about the rate of fault insertion and the failure potential of the software. Unfortunately, we cannot measure these things directly.

Let us observe from the MDS data that there is a very definite relationship between our software metric data and the historical software quality that we have obtained from this system. We have applied our new fault measurement methodology to the change history of the MDS system [ISSRE2003]. We also have the complete failure history of the MDS system at each of the builds. We would now like to study the relationship between our evolutionary measurement data and these software quality data.

In Figure 3 the system FI data for the evolving MDS system are shown together with the cumulative fault count and the cumulative failure counts. We see that these plots are very similar. Indeed the correlation coefficient between cumulative faults and FI is 0.98. Similarly, the correlation coefficient between cumulative failures and FI is 0.97.

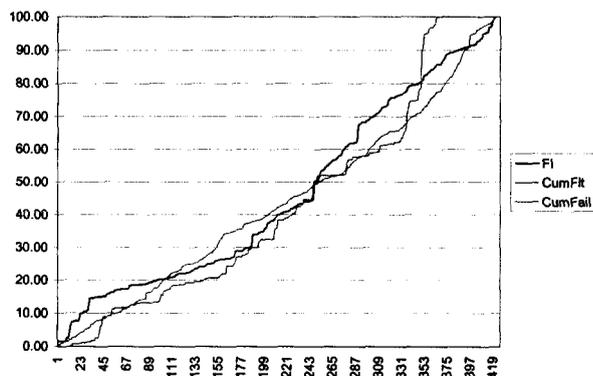


Figure 3. The FI Metric v. Quality Data

Now if we look at the individual attribute domains in regards to the index of cumulative faults we see that not all attributes are equally associated with faults. The cumulative

domain values for each of the system builds is plotted in Figure 4 together with the cumulative FI for each build. It is quite apparent from this figure that Domain 1, the control domain, is most closely associated with the cumulative fault measure. Indeed, the correlation coefficient between Domain 1 cumulative domain values and the cumulative fault values is 0.94. The correlation between Domain 2 and cumulative faults is -0.20. Finally, correlation between Domain 3 and cumulative faults is 0.71.

Perhaps the most disturbing feature on the software development landscape is the fact that the variation in the fault insertion rate is not constant and does not improve over time. To expose this characteristic, we will compute a moving standard deviation through the fault data associated with each of the system builds. To compute this moving standard deviation, fault data were grouped in sets of ten beginning with the first ten builds, followed by builds 2 through 11 and so forth. The standard deviation for each of these groups was then plotted in Figure 5. There is an apparent trend in these data. That is, the variation in the number of faults appears to increase directly with the increasing complexity of the system. Indeed, the correlation coefficient between the cumulative churn, shown in Figure 5 and the moving fault standard deviation is 0.44. Cumulative churn is derived from FI. This relationship can also be clearly seen in Figure 5. Again, the correlation between FI and the variance of the faults is 0.44.

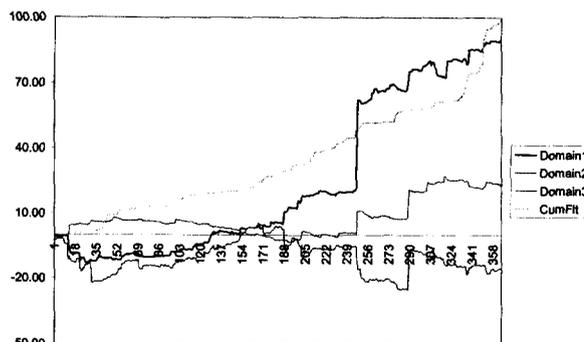


Figure 4. Cumulative Domain Scores v. Cumulative Faults

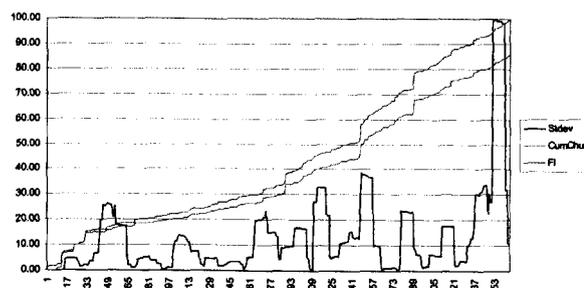


Figure 5. Variation in Cumulative Faults

Not only do software systems change during their evolution. These changes are closely linked to software quality

criteria. Changes to some software attributes will have a much greater impact than similar changes in other attributes.

## 10. Discussion and Future Work

We have seen that structural measurements of a system's structural evolution can serve as useful predictors of the number of faults inserted into a system during its development. Furthermore, some types of change are more likely than others to result in the introduction of faults in the system. By identifying the principal domains in which a change occurred, we are able to determine whether the change is likely to induce faults or is relatively innocuous. In a very real sense, then, we did meet our objective in developing a practical method of predicting software fault content based on the structural characteristics of the MDS software system. Although the number of measurements used in this study was rather limited, over 90% of the variation in the cumulative fault count was explained by the cumulative value of a specific domain. This is a sufficiently large value for development efforts to start using these measurements as a management tool. Software managers should be able to use these measurements to:

- Determine whether a given set of changes is likely to result in the insertion of faults into the system, or whether the changes are relatively benign.
- By applying these techniques to subset of the system and comparing the results, identify the modules which have had the most faults inserted
- Determine how many more faults a given module has had inserted into it than another module.

Future work will involve enlarging the set of measurements taken by Darwin<sup>TM</sup> and determining the effect of the enlarged set on the accuracy of the fault predictors. For instance, Darwin<sup>TM</sup> does not currently take any measurements specifically related to objects (e.g., number of methods, depth of an object in the class hierarchy). Future versions of Darwin<sup>TM</sup> might implement the object-oriented measures proposed by Chidamber and Kemerer [Chid94].

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

We have developed a definition of software faults that can be applied to source code. The definition allows faults to be unambiguously measured at the level of individual

modules. Since faults are measured at the same level at which structural measurement are taken, it becomes more feasible to construct meaningful models relating the number of faults inserted into a software module to the amount of structural change made to that module. Because of the way in which faults are defined, the task of counting faults is easily automated, making it much more practical to analyze large software systems such as those developed to support NASA flight missions. In other words, the faults may be quantified by a software tool that can analyze the deltas in code modules maintained by the configuration control system and measure those changes specifically attributable to failure reports.

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.

The fault counting technique described in [Muns02] does not currently allow us to identify all situations in which a given token has been replaced by another, which may lead to undercounting the number of faults that have been corrected. Consider the following example, for which the original statement is:

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

which is changed during repair to

$$(2) a = b - c + d;$$

The six tokens representing (1) is  $B_1 = \{<a>, <=>, <b>, <+>, <c>\}$ , and the eight tokens representing (2) is  $B_2 = \{<a>, <=>, <b>, <->, <c>, <+>, <d>\}$ . We see that what has happened is that  $<+>$  in (1) has been changed to  $<->$  in (2), and that  $<c>$ ,  $<+>$ , and  $<d>$  have been added in (2). However, the bag difference  $B_2 - B_1 = \{<->, <d>\}$ , indicating the addition of two new tokens, but failing to indicate that one token was replaced by another.

The technique also does not identify the number of tokens that have been reordered. Consider the situation illustrated by comparing the faulty statement (3) to the repaired statements (4), shown below.

$$(3) a = b - c;$$

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

The bag difference is  $B_3 - B_4 = \{ \}$ , the cardinality of which is 0. We see that the ordering of  $<b>$  and  $<c>$  has changed from (3) to (4), for which we could count 2 faults. However, our examination of the bag difference leads only to the conclusion that at least 1 token has changed, for which we count 1 fault according to our definition. In this situation,

1 fault according to our definition. In this situation, our definition could again lead to undercounting the number of faults repaired. Resolution of these fault-counting issues is also a part of our planned future work. We intend to investigate a technique originally devised for merging two different versions of a software component reported at the 2002 International Conference on Software Maintenance [Hunt02] which may be relevant.

## Acknowledgments

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

## References

- [CA02] Computer Associates, "AllFusion Harvest Change Manager Features, Descriptions & Benefits", Feb. 11, 2002, available at: [http://www3.ca.com/Files/FactSheet/af\\_harvest\\_cm\\_fdb.pdf](http://www3.ca.com/Files/FactSheet/af_harvest_cm_fdb.pdf)
- [Chil92] R. Chillarege, I. Bhandari, J. Chaar, M. Halliday, D. Moebus, B. Ray, M.-Y. Wong, "Orthogonal Defect Classification - A Concept for In-Process Measurement", IEEE Transactions on Software Engineering, November, 1992, pp. 943-946.
- [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, [www.cylant.com](http://www.cylant.com)
- [Dvo99] D. Dvorak, R. Rasmussen, G. Reeves, A. Sacks, "Software Architecture Themes In JPL's Mission Data System", AIAA Space Technology Conference and Exposition, September 28-30, 1999, Albuquerque, NM.
- [Ghok97] S. S. Gokhale, M. R. Lyu, "Regression Tree Modeling for the Prediction of Software Quality", proceedings of the Third ISSAT International Conference on Reliability and Quality in Design, pp 31-36, Anaheim, CA, March 12-14, 1997
- [Hall00] G. A. Hall and J. C. Munson, "Software evolution: code delta and code churn", Journal of Systems and Software 54 (2) (2000) pp. 111-118
- [Hunt02] J. Hunt, W. Tichy, "Extensible Language-Aware Merging", proceedings of the 2002 International Conference on Software Maintenance, Montréal, Canada, Oct 3-6, 2002, pp 511-520
- [IEEE83] "IEEE Standard Glossary of Software Engineering Terminology", IEEE Std 729-1983, Institute of Electrical and Electronics Engineers, 1983.
- [IEEE88] "IEEE Standard Dictionary of Measures to Produce Reliable Software", IEEE Std 982.1-1988, Institute of Electrical and Electronics Engineers, 1989.
- [IEEE93] "IEEE Standard Classification for Software Anomalies", IEEE Std 1044-1993, Institute of Electrical and Electronics Engineers, 1994.
- [Khos01] T. Khoshgoftaar, "An Application of Zero-Inflated Poisson Regression for Software Fault Prediction", proceedings of the 12th International Symposium on Software Reliability Engineering, pp 66-73, Hong Kong, Nov, 2001.
- [Khos01a] T. M. Khoshgoftaar, E. B. Allen, "Modeling Software Quality with Classification Trees", in H. Pham (ed), Recent Advances in Reliability and Quality Engineering, Chapter 15, pp 247-270, World Scientific Publishing, Singapore, 2001.
- [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.
- [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.
- [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.
- [Muns03] J. Munson, Software Engineering Measurement, CRC Press, 2003.
- [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 Computer Society Press.
- [Niko01] A. Nikora, J. Munson, "A Practical Software Fault Measurement and Estimation Framework", Industrial Presentations proceedings of the 12th International Symposium on Software Reliability Engineering, Hong Kong, Nov 27-30, 2001.
- [Schn97] N. F. Schneidewind, "Software Metrics Model for Integrating Quality Control and Prediction", proceedings of the 8th International Symposium on Software Reliability Engineering, pp 402-415, Albuquerque, NM, Nov, 1997.
- [Schn01] N. F. Schneidewind, "Investigation of Logistic Regression as a Discriminant of Software Quality", proceedings of the 7th International Software Metrics Symposium, pp 328-337, London, April, 2001.