Developing Fault Predictors for Evolving Software Systems

Allen P. Nikora
Jet Propulsion Laboratory,
California Institute of Technology
Mail Stop 125-233
Pasadena, CA, USA 91109-8099
Vox: +1 (818)393-1104
Fax: +1 (818)393-1362
Allen.P.Nikora@ipl.nasa.gov

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

ABSTRACT

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

We have begun a new investigation of this relationship 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 line of code. 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. The new fault standard also insures that the model so developed has greater predictive validity.

KEYWORDS: defect content estimation techniques, fault prediction, software metrics, software quality models
ABSTRACT

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

We have begun a new investigation of this relationship 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 line of code. 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. The new fault standard also insures that the model so developed has greater predictive validity.

KEYWORDS: defect content estimation techniques, fault prediction, software metrics, software measurement frameworks, software quality models

1. Introduction

Over the past several years, we have been investigating relationships between measurements of a software system's structural evolution and the rate at which faults are inserted into that system [Muns98, Niko98]. Measuring the structural evolution of a software system has proven to be a 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.

A fault, by definition, is a structural imperfection in a software system that may lead to the system's eventually failing. It is a physical characteristic of the system of which the type and extent may be measured using the same ideas used to measure the properties of more traditional physical systems. People making errors in their tasks introduce faults into a system. These errors may be errors of commission or errors of omission. There are, of course, differing etiologies for each fault. Some faults are attributable to errors in the specification of requirements. Some faults are directly at-
tibutable to error committed in the design process. Finally, there are faults that are introduced directly into the source code. There are two major types. There are faults of commission and faults of omission. Faults of commission involve implementing code that is not part of the specification or design. Faults of omission involve lapses wherein a behavior specified in the design was not implemented. In order to count faults, there must be a well-defined method of identification that is repeatable, consistent, and identifies faults at the same level of granularity as our static source code measurements.

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 Chokale 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 a system's measured structural evolution could predict 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. One particular table in this standard, Table 3c, allows classification of the type of behavior exhibited by the anomaly (e.g., "precision loss") or the type of defect that led to the anomaly (e.g., "referenced wrong data variable").

This type of scheme is helpful in determining the underlying causes of faults and failures, so that the development process may be modified to 1) identify the types of faults on which fault detection and removal resources should be focused for the current development effort, and 2) minimize the introduction of the most common types of faults in future develop-
Aaults at the module level. The recognition process for de-
that some of the anomaly types can readily be traced to a
single fault (e.g., “Operator in equation incorrect”). How-
however, the response to an “I/O Timing” anomaly may involve
changes to many lines of source code spread across multiple
source code files. In this case, the standard does not provide
enough information to allow counting the number of faults
at the module level.

Orthogonal Defect Classification (ODC), initially pub-
lished in 1992 [Chi92], provides a framework for 1) identi-
fying 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 faults 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
possible to use ODC alone to consistently identify and count
faults at the module level. The recognition process for de-
ficts 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 predicting fault content based on structural char-
acteristics that can be used by production software develop-
ment efforts to help them better manage the quality of the
systems they create. We chose to search for relationships
between the rate at which faults are inserted into source
code and the measured structural evolution of the source
code. 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 mea-
ured can be easily established.

Through the analysis of the structural evolution of a soft-
ware 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.

We worked in collaboration with the Mission Data Sys-
tem (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. For every failure reported against the
MDS, we were also able to identify the changes made to each
module in response to that failure, and thereby count the
number of faults that had been repaired. These measure-
ments were inputs to regression analyses to identify relation-
ships between the measured structural evolution and the
number of faults discovered.

4. A Description of the Mission Data System

The rationale and summary description of the MDS
provided here is taken 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. With increasingly powerful flight qualified
processors, there is an opportunity to migrate capabilities
from ground-based systems to spacecraft. There is also
an increasing need for such migration to accomplish
missions that must react quickly to events, without earth-
in-the-loop light-time delays, such as autonomous landing
on a comet and rover explorations on Mars.

- MDS is founded upon a state-based architecture,
where state is a representation of the momentary
condition of an evolving system. System states include
device operating modes, device health states, resource
levels, temperatures, pressures, etc, as well as
environmental states such as the motions of celestial
bodies and solar flux. Some aspects of system state are
best described as functions of other states; e.g., pointing
can be derived from attitude and trajectory. In all cases
state is accessible through state variables (as opposed to a
program’s local variables), and state evolution is
described on state timelines. State timelines provide the
fundamental coordinating mechanism since they describe
both knowledge and intent. A state-based architecture
implies the need for models since models describe how a
system’s state evolves. Together, state and models supply
what is needed to operate a system, predict future state,
control toward a desired state, and assess performance.
Domain knowledge is expressed explicitly in models rather than implicitly in program logic. Much of what makes software different from mission to mission is domain knowledge about instruments and actuators and sensors and plumbing and wiring and many other things. This knowledge includes relationships such as how power varies with solar incidence angle, conditions such as the fact that gyros saturate above a certain rate, state machines that prescribe safe sequences for valve operation, and dynamic models that predict how long a turn will take. Conventional practice has been to develop programs whose logic implicitly contains such domain knowledge, but this expresses the knowledge in a "hidden" form that is hard to validate and hard to reuse. In contrast, MDS advocates that domain knowledge be represented more explicitly in inspectable models. Such models can be tables or spreadsheets or rules or state machines or any of several forms, as long as they separate the domain knowledge from the general logic for applying that knowledge to solve a problem. The task of customizing MDS for a mission, then, becomes largely a task of defining and validating models.

Missions are to be operated via specifications of the desired state rather than sequences of actions. Traditionally, spacecraft have been controlled through linear (non-branching) command sequences that have been carefully designed on the ground. Such design is difficult for two reasons. First, the ground must predict spacecraft state for the time at which the sequence is scheduled to start, and that's difficult to know with confidence because of flight/ground communication limitations (data rate and light-time delay). Second, in the event that the actual spacecraft state is different than the predicted state, the sequence should be designed to fail rather than chance doing something harmful. MDS, in contrast, controls state - both flight and ground state - via "goals". A goal is defined as a prioritized constraint on the value of a state variable during a time interval. A goal differs from a command in that it specifies intent in the form of desired state. Such goal-directed operation is simpler than traditional sequencing because a goal is easier to specify than the actions needed to accomplish it. Importantly, goals specify only success criteria; they leave options open about the means of accomplishing the goal and the possible use of alternate actions to recover from problems. A goal is a unifying concept that encompasses daily operations, maintenance and calibration, resource allocation, flight rules, and fault responses.

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

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

5.2. Derived Metrics

As has been clearly established from our previous work, these metrics are highly correlated [Muns90, Hall01]. 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 of the underlying orthogonal domains have been shown in boldface type in this table.

Table 2 - The Principal Components Analysis

<table>
<thead>
<tr>
<th>Metric</th>
<th>Domain 1</th>
<th>Domain 2</th>
<th>Domain 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exec</td>
<td>.60</td>
<td>.49</td>
<td>.47</td>
</tr>
<tr>
<td>NonExec</td>
<td>.64</td>
<td>.53</td>
<td>.18</td>
</tr>
<tr>
<td>N_t</td>
<td>.28</td>
<td>.64</td>
<td>.65</td>
</tr>
<tr>
<td>N_f</td>
<td>.49</td>
<td>.70</td>
<td>.07</td>
</tr>
<tr>
<td>N_y</td>
<td>.28</td>
<td>.64</td>
<td>.65</td>
</tr>
<tr>
<td>N_z</td>
<td>.35</td>
<td>.90</td>
<td>.04</td>
</tr>
<tr>
<td>Nodes</td>
<td>.87</td>
<td>.31</td>
<td>.27</td>
</tr>
<tr>
<td>Edges</td>
<td>.88</td>
<td>.31</td>
<td>.27</td>
</tr>
<tr>
<td>Paths</td>
<td>.17</td>
<td>.10</td>
<td>.89</td>
</tr>
<tr>
<td>MaxPath</td>
<td>.87</td>
<td>.35</td>
<td>.29</td>
</tr>
<tr>
<td>AvePath</td>
<td>.86</td>
<td>.34</td>
<td>.33</td>
</tr>
<tr>
<td>Cycles</td>
<td>.67</td>
<td>.22</td>
<td>.02</td>
</tr>
<tr>
<td>Eigenvalues</td>
<td>4.79</td>
<td>3.13</td>
<td>2.24</td>
</tr>
</tbody>
</table>

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-th module \( z_i \) on the j-th build of the system there will be a data vector \( x_i = < x_{i1}, ..., x_{i2} > \) of 12 raw complexity metrics for that module. We can standardize each of the raw metrics by subtracting the mean \( \bar{x}_i \) of the metric #i over all modules in the j-th build and dividing by its standard deviation \( \sigma_i \) such that \( z_{ij} = (x_{ij} - \bar{x}_i) / \sigma_i \) represents the standardized value of the first raw metric for the i-th module on the j-th build.

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

For each module, there are now three new metrics, each representing one of 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.

5.3. Measuring Software Faults

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

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

\[ a = b + c \cdot d; \]

but had meant to type

\[ a = b + c / d; \]

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

\[ a = b + c \cdot d; \]

was changed to

\[ a = b + c * d; \]

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

The important consideration with this fault measurement strategy is that there must be some indication as to the amount of code that has changed in resolving a problem in the code. We have regularly witnessed changes to tens or even hundreds of lines of code recorded as a single "bug" or fault. The only measurable index of the degree of the change is the number of tokens that have changed to ameliorate the original problem. 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 will cause a failure when the compiled code that implements the source code token is executed.

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

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

Each line of text in each version of the program can be seen as a bag of tokens. That is, there may be multiple tokens of the same kind on each line of the text. When a software developer changes a line of code in response to the detection of a fault, either through normal inspection, code review processes, or as a result of a failure event in a program module, the tokens on that line will change. New tokens may be added. Invalid tokens may be removed. The sequence of tokens may be changed. Enumeration of faults under this definition is simple, straightforward. Most important of all, this process can be automated. Measurement of faults can be performed very precisely, which will eliminate the errors of observation introduced by existing ad hoc fault reporting schemes [Muns02, Muns03].

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

\[ a = b + c; \]

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

\[ a = b - c; \]

There will again be five tokens in the new line of code. This will be the bag \{ <a>, <e>, <b>, <e>, <c> \}. The bag difference is \[ B1 - B2 = \{ <a>, <> \} \]. The cardinality of B1 and B2 is the same. There are two tokens in the difference.

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

\[ a = c - b; \]

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

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

\[ a = 1 + c - b; \]

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

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

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

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, they 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].

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

\begin{table}[h]
\centering
\caption{The Measurement Baseline}
\begin{tabular}{|l|c|c|c|c|}
\hline
Metric & Domain & Mean & Stdev \\
\hline
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 \\
AvgPath & .251 & -.123 & .012 & 3.31 & 4.65 \\
Cycles & .269 & -.094 & -.179 & 0.11 & 0.50 \\
\hline
\end{tabular}
\end{table}
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 \( \overline{X}^B \), and a vector \( B^s \) 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: \( m_1, m_2, m_3, \ldots, m_n \). 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: \( \nu^i = <\nu_1^i, \nu_2^i, \nu_3^i, \ldots, \nu_n^i> \). This build index vector will allow us to preserve the precise structure of each for posterity. Thus, \( \nu_i^i \) in the vector \( \nu^i \) 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 \( \nu^i \) 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 \( \nu^i \) to \( \nu^{i+1} \) or even \( \nu^i \) to \( \nu^{i+k} \) or \( \nu^{i-k} \). 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, \( M^i_{ij} \), 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, \( M^j_{ij} \), is the set of modules that were in the early build, \( i \), and were removed prior to the later build, \( j \). The final set, \( M^f_{ij} \), 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 \( m_j \) was removed giving. Thus,

\[
M^j = M^i \cup M^i_{ij} - M^j_{ij}
\]

\[
= \{m_1, m_2, m_3, m_4, m_5\} \cup \{\} - \{m_j\}
\]

\[
= \{m_1, m_2, m_3, m_4\}
\]

Then between builds \( j \) and \( k \) two new modules, \( m_8 \) and \( m_9 \) are added and module \( m_6 \) is deleted giving

\[
M^f = M^j \cup M^i_{jk} - M^j_{jk}
\]

\[
= \{m_1, m_2, m_3, m_4\} \cup \{m_8, m_9\} - \{m_6\}
\]

\[
= \{m_1, m_2, m_3, m_4, m_8, m_9\}
\]

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

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

Let \( d_{ia}^j \) represent the \( i \)-th domain score of the \( a \)-th module on build \( j \) baselined by build \( B \). The new measure of domain churn, \( x \), for module \( m \) is simply \( x_a = |d_{ia}^j - d_{ia}^{j+1}| \). That is, the domain churn may be established by computing the baselined domain scores for any two builds and then find the absolute difference between these values. This represents the relative amount of change activity that there has been on each of the three domains between any two builds.

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

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

\[
X_a = \sum_{j=0}^{L} x_{ia}^{j+1}.
\]

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

Let us also observe that if module \( m \) were not present on builds \( j \) and \( j+1 \), then \( x_{ia}^{j+1} = 0 \). Also, if module \( m \) had been introduced on build \( j+1 \) then \( x_{ia}^{j+1} = |d_{ia}^j - d_{ia}^{j+1}| \).

8. The Relationship Between Software Faults and Change Activity

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

Code does not always change just to fix faults. Some changes to code during its evolution represent enhancements, design modifications, or changes in the code in response to evolving requirements. These incremental code enhancements may also result in the introduction of still more faults. Thus, as a system progresses through a series of builds, the domain scores of each program module that has been altered must also change. The rate of change in these domains should serve as a good index of the rate of fault introduction. That is the conjecture that we wish to explore.

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

The next step in this investigation was to compute the fault count for each program module. The driving force behind this measurement process was the Internal Anomaly Report (IAR). All changes to the software were tracked under the CCC Harvest version control system (now incorporated into Computer Associates' CM systems – see [CA02]). Each change to a program module was made either as an enhancement or in response to a particular IAR. If a module code delta was attributed to an IAR, then the faults attributed to that change were calculated using the token bag difference methodology described earlier. Thus, for each module version after the initial version, it was possible to track very precisely the change activity to that module and a very precise count of the fault tokens.

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

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

With the data from the remaining 563 modules, a multiple linear regression model was developed with the cumulative fault count as the dependent variable and the domain churn values as independent variables. The regression ANOVA for this analysis is shown in Table 4. It is clear from this analysis that there is a significant relationship between the domain churn and a module's fault burden.

The final regression model is shown in Table 5. Domain 1 clearly dominates this model. Domains 2 and 3 do not contribute to our understanding of the fault introduction.
process. The regression coefficients for these terms are not significant (p>0.01).

<table>
<thead>
<tr>
<th>Source</th>
<th>Sum of Squares</th>
<th>df</th>
<th>Mean Square</th>
<th>F</th>
<th>Sig.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regression</td>
<td>10091546</td>
<td>3</td>
<td>3333848</td>
<td>293</td>
<td>p&lt;0.01</td>
</tr>
<tr>
<td>Residual</td>
<td>6430656</td>
<td>560</td>
<td>11.483</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>1652203</td>
<td>563</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5 - The Regression Model

<table>
<thead>
<tr>
<th>Model</th>
<th>Coefficients</th>
<th>t</th>
<th>Sig.</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Constant)</td>
<td></td>
<td>18.24</td>
<td>3.5</td>
</tr>
<tr>
<td>Domain 1 Churn</td>
<td>21.63</td>
<td>17.3</td>
<td>P&lt;.01</td>
</tr>
<tr>
<td>Domain 2 Churn</td>
<td>-.59</td>
<td>-3</td>
<td>p&gt;.01</td>
</tr>
<tr>
<td>Domain 3 Churn</td>
<td>.93</td>
<td>.7</td>
<td>p&gt;.01</td>
</tr>
</tbody>
</table>

The constituent metrics for Domain 1 were established in Table 2. The metrics most closely associated with this domain were Nodes, Edges, MaxPath, AvePath, and Cycles. All of these metrics are attributes of the control flow graph representation of a program module. From this we can infer that the fault burden attributed to change activity is most closely associated with the change activity in those modules that had the greatest change made to their control structure.

Table 6 - Quality of the Regression Model

<table>
<thead>
<tr>
<th>Model</th>
<th>R</th>
<th>R Square</th>
<th>Adjusted R Square</th>
<th>Std. Error of the Estimate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summary</td>
<td>.782</td>
<td>.611</td>
<td>.609</td>
<td>107.16024</td>
</tr>
</tbody>
</table>

Next we want to know something about the relative quality of the regression model that we have developed. We will use the R² statistic as an indicator of the quality of the model. These data are shown in Table 6. We can see from this table that the adjusted is approximately 0.61. This means, roughly, that we can account for approximately 60% of the variation in the cumulative fault count with the cumulative domain churn for Domain 1. This is a very respectable value for the limited metric set that the Darwin tool currently uses.

9. 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. 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, about 60% of the variation in the cumulative fault count was explained by this set of measurements. This is a sufficiently large value for development efforts to start using these measurements as a management tool. Software managers should be able to use these measurements to:

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

The Darwin™ 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.

To date, we have been able to develop models relating the cumulative number of faults repaired to the cumulative measured structural change. For future work, we would like to be able to model the fault insertion rate as a function of the amount of structural change that occurs between the insertion of faults. In order to do this, we must be able to identify the version of the module in which a fault first appeared. CVS, the configuration management system used in the Darwin™ appliance, allows us to do this. CVS can generate reports identifying the version in which each line was inserted into the system. A fragment of this type of report is shown below in Figure 1. The leftmost numbers indicate the version in which each line was inserted. We will use this capability to determine the version into which each fault was inserted. Since we have a complete history of a system's structural evolution, we will be able to compute the amount of change that occurred between each group of faults inserted into that module, and thereby determine an empirical distribution of the number of faults inserted per unit change.

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.

The context in which the repair is made to determine the first version of the module in which the absence of the line would have constituted a fault. As an approximation, we can determine when the lines bounding the repair first appeared in the module. For instance, suppose that repairing module A involves adding one line between lines 99 and 100 of version 11. The new line now becomes line 100, and line 100 becomes line 101. After committing the change to the repository as version 12, we can use the revision control system's reporting capabilities to identify the first version in which both lines 99 and 101 appear — we will take this version to be the one in which the fault originally occurred.

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.

As mentioned above, the determination of when a fault was initially inserted into a component is based on the ability of the revision control system to identify the version in which each line first appeared in the module. For faults whose repair involves removing or modifying a line, determination of when the fault was introduced into the module is straightforward. However, if the repair activity involves adding a line, determining the version into which the fault was inserted is more complicated. We need to examine the context in which the repair is made to determine the first version of the module in which the absence of the line would have constituted a fault. As an approximation, we can determine when the lines bounding the repair first appeared in the module.

The technique described above 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:

\( (5) \) \( a = b + c; \)

which is changed during repair to

\( (6) \) \( a = b - c + d; \)

The six tokens representing \( (5) \) is \( B_5 = \{<a>, <c>, <b>, <+>, <-, <d>\} \), and the eight tokens representing \( (6) \) is \( B_6 = \{<a>, <c>, <b>, <c>, <-, <+, <d>\} \). We see that what has happened is that \( <+> \) in \( (5) \) has been changed to \( <-> \) in \( (6) \), and that \( <-, <c>, <b>, <d> \) have been added in \( (6) \). However, the bag difference \( B_6 - B_5 = \{<+, <b>, <d>, <d>\} \), indicating the addition of two new tokens, but failing to indicate that one token was replaced by another. We are currently developing techniques to resolve this issue.

The technique also does not identify the number of tokens that have been reordered. Consider again the situation illus-
trated by comparing (2) and (3). We see that the ordering of $<$b> </b> and $<e>$ has changed from (2) to (3), 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, our definition could again lead to undercounting the number of faults repaired.

Acknowledgments

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

References


