Toward A Quantifiable Definition of Software Faults

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

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

ABSTRACT

An important aspect of developing models relating the number and type of faults in a software system to a set of structural measurement is defining what constitutes a fault. By definition, a fault is a structural imperfection in a software system that may lead to the system's eventually failing. A measurable and precise definition of what faults are makes it possible to accurately identify and count them, which in turn allows the formulation of models relating fault counts and types to other measurable attributes of a software system. Unfortunately, the most widely-used definitions are not measurable – there is no guarantee that two different individuals looking at the same set of failure reports and the same set of fault definitions will count the same number of underlying faults. The incomplete and ambiguous nature of current fault definitions adds a noise component to the inputs used in modeling fault content. If this noise component is sufficiently large, any attempt to develop a fault model will produce invalid results.

As part of our on-going work in modeling software faults, we have developed a method of unambiguously identifying and counting faults. Specifically, we base our recognition and enumeration of software faults on the grammar of the language of the software system. By tokenizing the differences between a version of the system exhibiting a particular failure behavior, and the version in which changes were made to eliminate that behavior, we are able to unambiguously count the number of faults associated with that failure. With modern configuration management tools, the identification and counting of software faults can be automated.

1. Introduction

Unfortunately there is 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. In other words, 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 attributable to error committed in the design process. Finally, there are faults that are introduced directly in to 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. In a careful examination of software faults over the years, we have observed that the overwhelming number of faults that are recorded as code faults are really design faults. Some software faults are

really faults in the specification. The design implements the specification and the code implements the design. We must be very careful to distinguish among these fault categories.

There may be faults in the specification. The specification may not meet the customer's needs. If this problem first manifests itself in the code, it still is not a code fault. It is a fault in the program specification or a specification fault. The software design may not implement the software requirements specification. Again, these design problems tend to be made manifest during software testing. Any such design faults must be identified correctly as design faults. In a small proportion of faults, the problem is actually a code problem. In these isolated cases, the problem should be reported a code fault.

We observed an example of this type of problem recently in a project on a large embedded software system. The program in question was supposed to interrogate a status register on a particular hardware subsystem for a particular bit setting. The code repeatedly misread this bit. This was reported as a software problem. What really happened was that the hardware engineers had implemented a hardware modification that shifted the position of the status bit in the status register. They had failed to notify the software developers of this material change in the hardware specification. The software system did exactly what it was supposed to do. It is just that this no longer met the hardware requirements. Yet the problem remains on record to this date as a software fault.

2. Related Work

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 allows classification of the type of behavior exhibited by the anomaly (e.g., "precision loss") or the type of defect that led to the anomaly (e.g., "referenced wrong data variable"). This type of scheme is helpful in determining the underlying causes of faults and failures, so that the development process may be modified to 1) identify the types of faults on which fault detection and removal resources should be focused for the current development effort, and 2) minimize the introduction of the most common types of faults in future development tasks. However, classification standards do not provide enough information to help count the number of faults in the system. Returning to Table 3c of [IEEE93], we see that some of the anomaly types can readily be traced to a single fault (e.g., "Operator in equation incorrect"). However, the response 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.

An ad-hoc fault taxonomy was developed by the authors in [Niko97] and [Niko98] in 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.

Orthogonal Defect Classification (ODC), initially reported in 1992 [Chil92], 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 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.

In [Fran98], Frankl et al. develop a model for evaluating test methods by the delivered reliability of the system. The aspect of that work relevant to this paper is the rejection of the traditional notion of faults in favor of the idea of failure regions ("a collection of failure inputs that some change fixes exactly"). Frankl et al. observe that faults have no unique characterization – a software component fails for some testset, and is then changed so that it succeeds on that testset. A simple example illustrates the concept: suppose we have a program composed of two functions, A and B. Function A computes a real number, and then calls function B to compute the square root of that number. If the value computed by A is less than 0, the program will fail. There are two changes that can be made – either A can be changed so that it never passes a value less than 0 to B (e.g., call B only if the value is not negative), or B can be changed so that negative input values will not cause it to fail (e.g., compute the

square root of the input's absolute value). We agree that faults do not necessarily have a unique characterization. However, our experience with development efforts indicates that there is often one set of acceptable repair actions that is noticeably less costly than the alternatives, both in terms of the number of affected components and total amount of changes made. Based on our experience, we assume that in a production development environment, developers will seek the least costly alternatives that they believe will effect the required repairs in order to maintain the required delivery schedule. Under this assumption, we can consider the repair actions to be unique, which we can then use as the basis of a meaningful fault count.

3. Recognizing 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. The software fault size description problem is very similar to that confronted by civil engineers in the construction of a building. When structural concrete is poured to form the columns of a building, some voids will naturally occur in the concrete. There are big voids and there are little voids. All voids may potentially weaken the structure. Two factors must be considered in the determination of the structural consequences of voids in the pour. First, there is the amount of stress in the vicinity of the void. Second, there is the volume of the void. A small void in a highly stressed location will make the building unacceptable weak. A large void in the surface of a column may simply create visual problems.

Software faults, just like voids in concrete, also are large or small. The term, fault, has a size component just as does a structural void in the concrete pour. 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. The subject of this paper is the identification and enumeration of faults that occur in source code. We ought to be able to do this 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 the C programming language we will consider the structures shown in Figure 1 below to be executable statements.

```
<executable_statement> ::= <labeled_statement> |
        <expression> |
        <selection_statement> |
        <iteration_statement> |
        <jump_statement>
```

Figure 1 - Executable Statements in C

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.

Non-executable statements are shown below in Figure 2:

```
<declaration> ::= <declaration_specifiers> ;
    | <declaration_specifiers> <init_declarator_list> ';'
```

Figure 2 - Non-executable Statements in C

We will find faults within these statements. The granularity of measurement for faults will be in terms of tokens that have changed. Thus if one had typed the following statement in C:

a = b + c * d;

but had meant to type

a = b + c / d;

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

a = b + c * d;

was changed to

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

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

The important consideration with this fault measurement strategy is that there must be some indication as to the amount of code that has changed in resolving a problem in the code. We have regularly witnessed changes to tens or

even hundreds of lines of code recorded as a single "bug" or fault. 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.

4. Counting Software Faults

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.

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

(1) a = b + c;

There are five tokens on this line of code. They are B_1 = {<a>, <=>, <b>, <+>, <c>} where B_1 is the bag representing this token sequence. Now let us suppose that the design, in fact, required that the difference between b and c be computed:

(2) a = b - c;

There will again be five tokens in the new line of code. This will be the bag B2 = {<a>, <=>, <b>, <->, <c>}. The bag difference is B_1 - B_2 = {<+>, <-> }. The cardinality of B_1 and B_2 is the same. There are two tokens in the difference. Clearly, one token has changed from one version of the module to another. There is one fault.

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

(3) a = c - b;

The new bag for this new line of code will be B_3 = {<a>, <=>, <c>, <->, <b>}. The bag difference between (2) and (3) is B_2 - B_3 = { }. The cardinality of B_2 and B_3 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.

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

(4) a = 1 + c - b;

This will yield a new bag B_4 = {<a>, <=>, <1>, <+>, <c>, <->, <b>}. The bag difference between (3) and (4) is B_3 - B_4 = {<1>, <+>}. The cardinality of B_3 is five and the cardinality of B_4 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 faults should not be fixed by one version of the code module. Each version of the module represents should represent exactly one enhancement or one defect.

5. Examples

We will take a simple example and trace the evolution of a source code program through three successive revisions through the UNIX rcs program. The sample program is from Table 2 repeated here (with added line numbers for future reference).

```
1  int Sum(int upper)
2  {
3   int sum = 0;
4   int index = 0;
5
6   label:
7     if(index < upper)
8       {
9         index++;
10         sum = sum + index;
11         goto label;
12       }
13  return sum;
14 }
```

Figure 3 – Sample Program

Figure 3 above represents version 1.1 of the program. Successive updates to this will be 1.2, 1.3, etc. The rcs system will keep track of the version number, the date and time of the update, and the author of the rcs activity. An abridged version of the rcs module structure to record these data is shown in Table 1below.

```
1.4
date  2005.02.01.22.17.38;   author John Doe;
next  1.3;

1.3
date  2005.01.22.22.01.31;   author John Doe;
next  1.2;

1.2
date  2005.01.20.21.54.08;   author Sam Lee;
next  1.1;

1.1
date  2005.01.15.21.49.29;   author Mary Roe;
next  ;
```

Table 1 - Change History

For rcs, the most recent version, in this case 1.4, is kept at the top of the list and the list is numbered chronologically backwards in time. Each version keeps a pointer to the next version in the table.

The actual changes to the source code at each version are shown in Table 2 below. The rcs program will always keep the most recent version in the file. This is shown in the table entry beginning with, in this case, version 1.4. The second entry in the record for version 1.4 is an entry beginning with the word log and delimited by @'s. This is the log comment introduced by the developer. In our proposed model this log entry would begin with the word, fault, if the version increment were attributable to a fault fix or the word, change, if it were attributable to a change in design or requirements. The initial log entry, version 1.1, is for neither a change nor a fault fix but is the title of the program.

```
1.4
Log
@fault: fixed relational operator
@
Text
@int Sum(int upper)
{
 int sum = 0;
 int index = 0;
 label:
   if(index > upper)
     {
       index++;
       sum = sum + index;
       goto label;
     }
 update (index);
 Return sum;
@
1.3
Log
@fault: inserted call to update function
@
Text
@d7 1
a7 1
   if(index <= upper)
@
1.2
Log
@fault: found a problem with a relational operator
@
Text
@d13 1
@
1.1
Log
@Initial revision
@
Text
@d7 1
a7 1
   if(index < upper)
@
```

Table 2 – Sample Program Versions

Following the log entry is the text entry. In the case of rcs, the topmost text entry is the most recent version of the program. Each of the subsequent table entries shows the changes that must be made to the most recent program to change it to a previous version. All changes are made, in rcs, by adding or deleting whole lines. Thus, to return to version 1.3 from version 1.4, the text part of record 1.3 tells us to go to line 7 (relative to 1) of the program and delete one line. That is what the line d7 1 tells us. The next text line says that we must add one line, a7 1, again at line 7. The text that must be added is on the following line. Thus, version 1.3 will have the appearance shown in Figure 4 below.

Line number 7 has been changed on version 1.3. Let

B_2 = {<if>, <(>, <index>, <<=>, <upper>, <)>}.

represent this bag of tokens. On version 1.4 the bag of tokens is

B_1 = {<if>, <(>, <index>, <>>, <upper>, <)>}.

The bag difference is B2 - B1 = {<<=>, <>>}. The cardinality of B_2 is 6 and the cardinality of B_1 is 6. The cardinality of the bag difference is two. Therefore, one token has changed and we will record one fault.

To return to version 1.2 from version 1.3 we see that we must delete line 13. All of the tokens on this line were place there in remediation of a fault. The bag representing this line of tokens is

B_3 = {<update>, <(>, <index>, <)>, <;> } .

There are five tokens on this line. There was no former version of this line in version 1.2. Therefore all of the tokens on this line were put into the program to fix a defect in the program. We will then record 5 faults for this fix.

Finally, to return to the initial version, 1.1, of the program we must delete line 7 and add a new line represented by the bag

B_4 = {<if>, <(>, <index>, <<>, <upper>, <)>} .

This is similar to the transition between versions 1.3 and 1.4. Only one token has changed. We will record one fault for this module version.

```
1 int Sum(int upper)
2 {
3   int sum = 0;
4   int index = 0;
5
6   label:
7     if(index <= upper)
8       {
9         index++;
10         sum = sum + index;
11         goto label;
12       }
13  update (index);
14  return sum;
15 }
```

Figure 4 – Sample Program, Version 1.3

6. Current Application

The fault definition given above is currently being applied to a JPL software development effort. The goal of this effort is to extend previous work in identifying relationships between measurements of the system's structural evolution and the number and type of faults inserted into the system during its development.

Structural measurements are obtained in a straightforward manner, as described in [Niko01]. For the system being investigated, we are fortunate in having the problem reporting system integrated with the configuration management repository. For each failure reported through the system, a "change package" is automatically opened in the repository. Developers then check repairs into the change package, and commit the completed change package to the repository when the repairs have been completed. In this way, it is possible to identify the changes that were made in response to each failure that was reported.

Using the fault definition presented above in conjunction with the information about the repairs that were made, we are able to analyze the differences between the modules associated with a failure and the repaired versions of those modules, and count the number of faults. The configuration management system we are using to track the system's history, 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 5. The leftmost numbers indicate the version in which each line was inserted. We use this capability to determine the version into which each fault was inserted. Since we have a complete history of the system's structural evolution, we can determine for each module in the system 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 are currently extracting the fault information from the more than 6000 affected modules, and will be using this information over the next several months to:

- Determine empirical distributions relating the number of faults inserted to the amount of structural change relating to the software.
- Identify relationships between different types of structural change (as indicated by the domain scores measuring structural change) and the number of faults inserted.
- Identify relationships between the type of fault inserted and the type of structural change that was made.

```
1.28 (jim 21-Mar-01): int watcher_parse_wrench_options (char * args)
1.28 (jim 21-Mar-01): {
1.28 (jim 21-Mar-01):   RListOfRBuf * options;          /* The separated options. */
1.28 (jim 21-Mar-01):   RListOfRBufEntry * option_entry;  /* An entry in the list of options. */
1.31 (matt 04-May-01):  RListOfRBuf * option_parts;  /* Option entry broken into name and value. */
1.28 (jim 21-Mar-01):   RListOfRBufEntry * option_name;  /* The Name of the current option. */
1.28 (jim 21-Mar-01):   RListOfRBufEntry * option_val;   /* The Value (if applicable) of the
1.31 (matt 04-May-01):                                    * current option. */
1.28 (jim 21-Mar-01):   ENode * command_line;           /* Node containing command line options. */
1.31 (matt 04-May-01):  int well_formed = TRUE;   /* Is this set of options well-formed? */
1.31 (matt 04-May-01):
1.28 (jim 21-Mar-01):
1.35 (cory 23-Jan-02):  options = rstr_split (args, ":", 0);
1.28 (jim 21-Mar-01):   command_line = watcher_config_get_command_line ();
1.28 (jim 21-Mar-01):
1.28 (jim 21-Mar-01):   RLIST_FOREACH (options, option_entry)
1.28 (jim 21-Mar-01):   {
1.35 (cory 23-Jan-02):    option_parts = rbuf_split (option_entry->buf, "=", 1);
1.28 (jim 21-Mar-01):    option_name = rlist_nth (option_parts, 0);
1.28 (jim 21-Mar-01):    option_val = rlist_nth (option_parts, 1);
1.28 (jim 21-Mar-01):
1.31 (matt 04-May-01):
1.29 (jim 23-Mar-01):    if (rbuf_equal_str (option_name->buf, "allow-nodrivers"))
1.28 (jim 21-Mar-01):    {
1.28 (jim 21-Mar-01):     /* Wrench will want to be able to start watcher up before any
1.31 (matt 04-May-01):      * drivers are installed. */
1.29 (jim 23-Mar-01):     enode_attrib_str (command_line, "allow-nodrivers", "true");
1.28 (jim 21-Mar-01):     printf ("* Will allow starting without drivers..\n");
1.28 (jim 21-Mar-01):    }
1.29 (jim 23-Mar-01):    else if (rbuf_equal_str (option_name->buf, "only-transport"))
1.28 (jim 21-Mar-01):    {
1.28 (jim 21-Mar-01):     /* Wrench wants to start watcher with only the specified transport
1.31 (matt 04-May-01):      * (ie, most likely a unix socket or something, to send it commands
1.31 (matt 04-May-01):      * without trying to connect to any GUIs) */
1.28 (jim 21-Mar-01):     if (NULL == option_val)
1.28 (jim 21-Mar-01):      {
1.29 (jim 23-Mar-01):       printf ("only-transport requires a value.\n");
1.31 (matt 04-May-01):        well_formed = FALSE;
1.28 (jim 21-Mar-01):       break;
1.28 (jim 21-Mar-01):      }
1.29 (jim 23-Mar-01):     enode_attrib (command_line, "only-transport", option_val->buf);
1.28 (jim 21-Mar-01):    }
```

Figure 5 – CVS Annotation

7. Discussion and Future Work

We have proposed 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 mod-

ules maintained by the configuration control system and measure those changes specifically attributable to failure reports.

We are currently working with a task whose goal is the development of a comprehensive set of on-board control and ground support software components that can be adapted from mission to mission. We have in hand the complete structural history of the software that has been developed to date. For each of the more than 1000 reported failures, we know the set of modules that affect the repair, as well as the versions of those modules.

It is clear that noise in the fault measurements may have a significant effect on our results, so it will be necessary to quantify it as accurately as possible. In the section "Current Application", we have described the way in which the failure reporting and tracking system is integrated into the configuration management system. A significant source of noise could involve developers 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. Part of our analysis must involve selecting an appropriate subset of the reported failures and interviewing developers about the changes made in response to those failures. We must be careful to select representative failures from all system components to control for the noise inserted by each development team. We must 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. 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.

Note that this approximation may not always accurately indicate the version in which a fault was introduced. Consider 2 modules, A and B. Module A computes a real number and passes it to module B, which determines the square root of that number and returns that value. Versions 1-4 of module A are constrained to return values greater than or equal to 0, and because of that constraint, module B does not test the input to determine whether it is less than 0. However, a change in the requirements removes module A's constraint in later versions. Suppose that at the same time that version 5 of module A is created, version 2 of module B is created in response to a request to change the formatting of its output, but that no provision is made to determine whether its input is less than 0. We later discover that module B does not respond as expected to inputs from module A, and we change module B to first test the input value before extracting the square root. In this situation, we see that the fault was inserted at the same time that version 2 of module B was created, although the approximation described above would indicate that the initial version of module B contained the fault. Only a detailed examination of a selected subset of the failure reports with which we are working will be able to indicate the amount of uncertainty introduced by this approximation.

Finally, we wish to expand the categorization of fault types in future work. The categorizations we have provided above are quite simple – addition or removal of tokens, as well as a limited capabilities to detect tokens that have changed value or execution order. We would like to improve our ability to detect these types of changes.

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

The technique also does not identify the number of tokens that have been reordered. Consider again the situation illustrated by comparing (2) and (3). We see that the ordering of <b> and <c> 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.

We would also like to provide a more detailed categorization – for example, we could count the number of tokens for which the execution ordering was changed. Finally, we might wish to distinguish between those changes made in a condition controlling the execution of a block of statements, as opposed to changes made in other executable statements.

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.

References

[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/.

[Fran98] P. Frankl, D. Hamlet, B. Littlewood, L. Stringini, "Evaluating Testing Methods by Delivered Reliability", IEEE Transactions on Software Engineering, vol. 24, no. 8, August 1998, pp. 586-601.

[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.

[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, "Software System Defect Content Prediction From Development Process And Product Characteristics", Doctoral Dissertation, Department of Computer Science, University of Southern California, May, 1998.

[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.

Toward A Quantifiable Definition of Software Faults

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

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

ABSTRACT

An important aspect of developing models relating the number and type of faults in a software system to a set of structural measurement is defining what constitutes a fault. By definition, a fault is a structural imperfection in a software system that may lead to the system's eventually failing. A measurable and precise definition of what faults are makes it possible to accurately identify and count them, which in turn allows the formulation of models relating fault counts and types to other measurable attributes of a software system. Unfortunately, the most widely-used definitions are not measurable – there is no guarantee that two different individuals looking at the same set of failure reports and the same set of fault definitions will count the same number of underlying faults. The incomplete and ambiguous nature of current fault definitions adds a noise component to the inputs used in modeling fault content. If this noise component is sufficiently large, any attempt to develop a fault model will produce invalid results.

As part of our on-going work in modeling software faults, we have developed a method of unambiguously identifying and counting faults. Specifically, we base our recognition and enumeration of software faults on the grammar of the language of the software system. By tokenizing the differences between a version of the system exhibiting a particular failure behavior, and the version in which changes were made to eliminate that behavior, we are able to unambiguously count the number of faults associated with that failure. With modern configuration management tools, the identification and counting of software faults can be automated.

1. Introduction

Unfortunately there is 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. In other words, 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 attributable to error committed in the design process. Finally, there are faults that are introduced directly in to 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. In a careful examination of software faults over the years, we have observed that the overwhelming number of faults that are recorded as code faults are really design faults. Some software faults are

really faults in the specification. The design implements the specification and the code implements the design. We must be very careful to distinguish among these fault categories.

There may be faults in the specification. The specification may not meet the customer's needs. If this problem first manifests itself in the code, it still is not a code fault. It is a fault in the program specification or a specification fault. The software design may not implement the software requirements specification. Again, these design problems tend to be made manifest during software testing. Any such design faults must be identified correctly as design faults. In a small proportion of faults, the problem is actually a code problem. In these isolated cases, the problem should be reported a code fault.

We observed an example of this type of problem recently in a project on a large embedded software system. The program in question was supposed to interrogate a status register on a particular hardware subsystem for a particular bit setting. The code repeatedly misread this bit. This was reported as a software problem. What really happened was that the hardware engineers had implemented a hardware modification that shifted the position of the status bit in the status register. They had failed to notify the software developers of this material change in the hardware specification. The software system did exactly what it was supposed to do. It is just that this no longer met the hardware requirements. Yet the problem remains on record to this date as a software fault.

2. Related Work

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 allows classification of the type of behavior exhibited by the anomaly (e.g., "precision loss") or the type of defect that led to the anomaly (e.g., "referenced wrong data variable"). This type of scheme is helpful in determining the underlying causes of faults and failures, so that the development process may be modified to 1) identify the types of faults on which fault detection and removal resources should be focused for the current development effort, and 2) minimize the introduction of the most common types of faults in future development tasks. However, classification standards do not provide enough information to help count the number of faults in the system. Returning to Table 3c of [IEEE93], we see that some of the anomaly types can readily be traced to a single fault (e.g., "Operator in equation incorrect"). However, the response 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.

An ad-hoc fault taxonomy was developed by the authors in [Niko97] and [Niko98] in 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.

Orthogonal Defect Classification (ODC), initially reported in 1992 [Chil92], 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 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.

In [Fran98], Frankl et al. develop a model for evaluating test methods by the delivered reliability of the system. The aspect of that work relevant to this paper is the rejection of the traditional notion of faults in favor of the idea of failure regions ("a collection of failure inputs that some change fixes exactly"). Frankl et al. observe that faults have no unique characterization – a software component fails for some testset, and is then changed so that it succeeds on that testset. A simple example illustrates the concept: suppose we have a program composed of two functions, A and B. Function A computes a real number, and then calls function B to compute the square root of that number. If the value computed by A is less than 0, the program will fail. There are two changes that can be made – either A can be changed so that it never passes a value less than 0 to B (e.g., call B only if the value is not negative), or B can be changed so that negative input values will not cause it to fail (e.g., compute the

square root of the input's absolute value). We agree that faults do not necessarily have a unique characterization. However, our experience with development efforts indicates that there is often one set of acceptable repair actions that is noticeably less costly than the alternatives, both in terms of the number of affected components and total amount of changes made. Based on our experience, we assume that in a production development environment, developers will seek the least costly alternatives that they believe will effect the required repairs in order to maintain the required delivery schedule. Under this assumption, we can consider the repair actions to be unique, which we can then use as the basis of a meaningful fault count.

3. Recognizing 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. The software fault size description problem is very similar to that confronted by civil engineers in the construction of a building. When structural concrete is poured to form the columns of a building, some voids will naturally occur in the concrete. There are big voids and there are little voids. All voids may potentially weaken the structure. Two factors must be considered in the determination of the structural consequences of voids in the pour. First, there is the amount of stress in the vicinity of the void. Second, there is the volume of the void. A small void in a highly stressed location will make the building unacceptable weak. A large void in the surface of a column may simply create visual problems.

Software faults, just like voids in concrete, also are large or small. The term, fault, has a size component just as does a structural void in the concrete pour. 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. The subject of this paper is the identification and enumeration of faults that occur in source code. We ought to be able to do this 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 the C programming language we will consider the structures shown in Figure 1 below to be executable statements.

```
<executable_statement> ::= <labeled_statement> |
        <expression> |
        <selection_statement> |
        <iteration_statement> |
        <jump_statement>
```

Figure 1 - Executable Statements in C

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.

Non-executable statements are shown below in Figure 2:

```
<declaration> ::= <declaration_specifiers> ;
    | <declaration_specifiers> <init_declarator_list> ';'
```

Figure 2 - Non-executable Statements in C

We will find faults within these statements. The granularity of measurement for faults will be in terms of tokens that have changed. Thus if one had typed the following statement in C:

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

but had meant to type

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

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

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

was changed to

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

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

The important consideration with this fault measurement strategy is that there must be some indication as to the amount of code that has changed in resolving a problem in the code. We have regularly witnessed changes to tens or

even hundreds of lines of code recorded as a single "bug" or fault. 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.

4. Counting Software Faults

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.

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

(1) a = b + c;

There are five tokens on this line of code. They are B_1 = {<a>, <=>, <b>, <+>, <c>} where B_1 is the bag representing this token sequence. Now let us suppose that the design, in fact, required that the difference between b and c be computed:

(2) a = b - c;

There will again be five tokens in the new line of code. This will be the bag B2 = {<a>, <=>, <b>, <->, <c>}. The bag difference is B_1 - B_2 = {<+>, <-> }. The cardinality of B_1 and B_2 is the same. There are two tokens in the difference. Clearly, one token has changed from one version of the module to another. There is one fault.

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

(3) a = c - b;

The new bag for this new line of code will be B_3 = {<a>, <=>, <c>, <->, <b>}. The bag difference between (2) and (3) is B_2 - B_3 = { }. The cardinality of B_2 and B_3 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.

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

(4) a = 1 + c - b;

This will yield a new bag B_4 = {<a>, <=>, <1>, <+>, <c>, <->, <b>}. The bag difference between (3) and (4) is B_3 - B_4 = {<1>, <+>}. The cardinality of B_3 is five and the cardinality of B_4 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 faults should not be fixed by one version of the code module. Each version of the module represents should represent exactly one enhancement or one defect.

5. Examples

We will take a simple example and trace the evolution of a source code program through three successive revisions through the UNIX rcs program. The sample program is from Table 2 repeated here (with added line numbers for future reference).

```
1  int Sum(int upper)
2  {
3    int sum = 0;
4    int index = 0;
5
6    label:
7      if(index < upper)
8        {
9          index++;
10         sum = sum + index;
11         goto label;
12       }
13   return sum;
14 }
```

Figure 3 – Sample Program

Figure 3 above represents version 1.1 of the program. Successive updates to this will be 1.2, 1.3, etc. The rcs system will keep track of the version number, the date and time of the update, and the author of the rcs activity. An abridged version of the rcs module structure to record these data is shown in Table 1below.

```
1.4
date  2005.02.01.22.17.38;   author John Doe;
next  1.3;

1.3
date  2005.01.22.22.01.31;   author John Doe;
next  1.2;

1.2
date  2005.01.20.21.54.08;   author Sam Lee;
next  1.1;

1.1
date  2005.01.15.21.49.29;   author Mary Roe;
next  ;
```

Table 1 - Change History

For rcs, the most recent version, in this case 1.4, is kept at the top of the list and the list is numbered chronologically backwards in time. Each version keeps a pointer to the next version in the table.

The actual changes to the source code at each version are shown in Table 2 below. The rcs program will always keep the most recent version in the file. This is shown in the table entry beginning with, in this case, version 1.4. The second entry in the record for version 1.4 is an entry beginning with the word log and delimited by @'s. This is the log comment introduced by the developer. In our proposed model this log entry would begin with the word, fault, if the version increment were attributable to a fault fix or the word, change, if it were attributable to a change in design or requirements. The initial log entry, version 1.1, is for neither a change nor a fault fix but is the title of the program.

```
1.4
Log
@fault: fixed relational operator
@
Text
@int Sum(int upper)
{
 int sum = 0;
 int index = 0;
 label:
   if(index > upper)
     {
       index++;
       sum = sum + index;
       goto label;
     }
 update (index);
 Return sum;
@
1.3
Log
@fault: inserted call to update function
@
Text
@d7 1
a7 1
   if(index <= upper)
@
1.2
Log
@fault: found a problem with a relational operator
@
Text
@d13 1
@
1.1
Log
@Initial revision
@
Text
@d7 1
a7 1
   if(index < upper)
@
```

Table 2 – Sample Program Versions

Following the log entry is the text entry. In the case of rcs, the topmost text entry is the most recent version of the program. Each of the subsequent table entries shows the changes that must be made to the most recent program to change it to a previous version. All changes are made, in rcs, by adding or deleting whole lines. Thus, to return to version 1.3 from version 1.4, the text part of record 1.3 tells us to go to line 7 (relative to 1) of the program and delete one line. That is what the line d7 1 tells us. The next text line says that we must add one line, a7 1, again at line 7. The text that must be added is on the following line. Thus, version 1.3 will have the appearance shown in Figure 4 below.

Line number 7 has been changed on version 1.3. Let

B_2 = {<if>, <(>, <index>, <<=>, <upper>, <)>}.

represent this bag of tokens. On version 1.4 the bag of tokens is

B_1 = {<if>, <(>, <index>, <>>, <upper>, <)>}.

The bag difference is B2 - B1 = {<<=>, <>>}. The cardinality of B_2 is 6 and the cardinality of B_1 is 6. The cardinality of the bag difference is two. Therefore, one token has changed and we will record one fault.

To return to version 1.2 from version 1.3 we see that we must delete line 13. All of the tokens on this line were place there in remediation of a fault. The bag representing this line of tokens is

B_3 = {<update>, <(>, <index>, <)>, <;> } .

There are five tokens on this line. There was no former version of this line in version 1.2. Therefore all of the tokens on this line were put into the program to fix a defect in the program. We will then record 5 faults for this fix.

Finally, to return to the initial version, 1.1, of the program we must delete line 7 and add a new line represented by the bag

B_4 = {<if>, <(>, <index>, <<>, <upper>, <)>} .

This is similar to the transition between versions 1.3 and 1.4. Only one token has changed. We will record one fault for this module version.

```
int Sum(int upper)
{
  int sum = 0;
  int index = 0;

  label:
    if(index <= upper)
      {
        index++;
         sum = sum + index;
         goto label;
      }
  update (index);
  return sum;
}
```

Figure 4 – Sample Program, Version 1.3

6. Current Application

The fault definition given above is currently being applied to a JPL software development effort. The goal of this effort is to extend previous work in identifying relationships between measurements of the system's structural evolution and the number and type of faults inserted into the system during its development.

Structural measurements are obtained in a straightforward manner, as described in [Niko01]. For the system being investigated, we are fortunate in having the problem reporting system integrated with the configuration management repository. For each failure reported through the system, a "change package" is automatically opened in the repository. Developers then check repairs into the change package, and commit the completed change package to the repository when the repairs have been completed. In this way, it is possible to identify the changes that were made in response to each failure that was reported.

Using the fault definition presented above in conjunction with the information about the repairs that were made, we are able to analyze the differences between the modules associated with a failure and the repaired versions of those modules, and count the number of faults. The configuration management system we are using to track the system's history, 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 5. The leftmost numbers indicate the version in which each line was inserted. We use this capability to determine the version into which each fault was inserted. Since we have a complete history of the system's structural evolution, we can determine for each module in the system 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 are currently extracting the fault information from the more than 6000 affected modules, and will be using this information over the next several months to:

- Determine empirical distributions relating the number of faults inserted to the amount of structural change relating to the software.
- Identify relationships between different types of structural change (as indicated by the domain scores measuring structural change) and the number of faults inserted.
- Identify relationships between the type of fault inserted and the type of structural change that was made.

```
1.28 (jim 21-Mar-01): int watcher_parse_wrench_options (char * args)
1.28 (jim 21-Mar-01): {
1.28 (jim 21-Mar-01):  RListOfRBuf * options;          /* The separated options. */
1.28 (jim 21-Mar-01):  RListOfRBufEntry * option_entry;  /* An entry in the list of options. */
1.31 (matt 04-May-01):  RListOfRBuf * option_parts;   /* Option entry broken into name and value. */
1.28 (jim 21-Mar-01):  RListOfRBufEntry * option_name;   /* The Name of the current option. */
1.28 (jim 21-Mar-01):  RListOfRBufEntry * option_val;    /* The Value (if applicable) of the
1.31 (matt 04-May-01):                                    * current option. */
1.28 (jim 21-Mar-01):  ENode * command_line;          /* Node containing command line options. */
1.31 (matt 04-May-01):  int well_formed = TRUE;   /* Is this set of options well-formed? */
1.31 (matt 04-May-01):
1.28 (jim 21-Mar-01):
1.35 (cory 23-Jan-02):  options = rstr_split (args, ":", 0);
1.28 (jim 21-Mar-01):  command_line = watcher_config_get_command_line ();
1.28 (jim 21-Mar-01):
1.28 (jim 21-Mar-01):  RLIST_FOREACH (options, option_entry)
1.28 (jim 21-Mar-01):   {
1.35 (cory 23-Jan-02):     option_parts = rbuf_split (option_entry->buf, "=", 1);
1.28 (jim 21-Mar-01):     option_name = rlist_nth (option_parts, 0);
1.28 (jim 21-Mar-01):     option_val = rlist_nth (option_parts, 1);
1.28 (jim 21-Mar-01):
1.31 (matt 04-May-01):
1.29 (jim 23-Mar-01):     if (rbuf_equal_str (option_name->buf, "allow-nodrivers"))
1.28 (jim 21-Mar-01):     {
1.28 (jim 21-Mar-01):      /* Wrench will want to be able to start watcher up before any
1.31 (matt 04-May-01):       * drivers are installed. */
1.29 (jim 23-Mar-01):      enode_attrib_str (command_line, "allow-nodrivers", "true");
1.28 (jim 21-Mar-01):      printf ("* Will allow starting without drivers..\n");
1.28 (jim 21-Mar-01):     }
1.29 (jim 23-Mar-01):     else if (rbuf_equal_str (option_name->buf, "only-transport"))
1.28 (jim 21-Mar-01):     {
1.28 (jim 21-Mar-01):      /* Wrench wants to start watcher with only the specified transport
1.31 (matt 04-May-01):       * (ie, most likely a unix socket or something, to send it commands
1.31 (matt 04-May-01):       * without trying to connect to any GUIs) */
1.28 (jim 21-Mar-01):      if (NULL == option_val)
1.28 (jim 21-Mar-01):       {
1.29 (jim 23-Mar-01):        printf ("only-transport requires a value.\n");
1.31 (matt 04-May-01):         well_formed = FALSE;
1.28 (jim 21-Mar-01):        break;
1.28 (jim 21-Mar-01):       }
1.29 (jim 23-Mar-01):      enode_attrib (command_line, "only-transport", option_val->buf);
1.28 (jim 21-Mar-01):     }
```

Figure 5 – CVS Annotation

7. Discussion and Future Work

We have proposed 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 mod-

ules maintained by the configuration control system and measure those changes specifically attributable to failure reports.

We are currently working with a task whose goal is the development of a comprehensive set of on-board control and ground support software components that can be adapted from mission to mission. We have in hand the complete structural history of the software that has been developed to date. For each of the more than 1000 reported failures, we know the set of modules that affect the repair, as well as the versions of those modules.

It is clear that noise in the fault measurements may have a significant effect on our results, so it will be necessary to quantify it as accurately as possible. In the section "Current Application", we have described the way in which the failure reporting and tracking system is integrated into the configuration management system. A significant source of noise could involve developers 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. Part of our analysis must involve selecting an appropriate subset of the reported failures and interviewing developers about the changes made in response to those failures. We must be careful to select representative failures from all system components to control for the noise inserted by each development team. We must 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. 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.

Note that this approximation may not always accurately indicate the version in which a fault was introduced. Consider 2 modules, A and B. Module A computes a real number and passes it to module B, which determines the square root of that number and returns that value. Versions 1-4 of module A are constrained to return values greater than or equal to 0, and because of that constraint, module B does not test the input to determine whether it is less than 0. However, a change in the requirements removes module A's constraint in later versions. Suppose that at the same time that version 5 of module A is created, version 2 of module B is created in response to a request to change the formatting of its output, but that no provision is made to determine whether its input is less than 0. We later discover that module B does not respond as expected to inputs from module A, and we change module B to first test the input value before extracting the square root. In this situation, we see that the fault was inserted at the same time that version 2 of module B was created, although the approximation described above would indicate that the initial version of module B contained the fault. Only a detailed examination of a selected subset of the failure reports with which we are working will be able to indicate the amount of uncertainty introduced by this approximation.

Finally, we wish to expand the categorization of fault types in future work. The categorizations we have provided above are quite simple – addition or removal of tokens, as well as a limited capabilities to detect tokens that have changed value or execution order. We would like to improve our ability to detect these types of changes.

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

The technique also does not identify the number of tokens that have been reordered. Consider again the situation illustrated by comparing (2) and (3). We see that the ordering of <b> and <c> 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.

We would also like to provide a more detailed categorization – for example, we could count the number of tokens for which the execution ordering was changed. Finally, we might wish to distinguish between those changes made in a condition controlling the execution of a block of statements, as opposed to changes made in other executable statements.

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.

References

[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/.

[Fran98] P. Frankl, D. Hamlet, B. Littlewood, L. Stringini, "Evaluating Testing Methods by Delivered Reliability", IEEE Transactions on Software Engineering, vol. 24, no. 8, August 1998, pp. 586-601.

[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.

[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, "Software System Defect Content Prediction From Development Process And Product Characteristics", Doctoral Dissertation, Department of Computer Science, University of Southern California, May, 1998.

[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.