Finding Fault with Faults: A Case Study

Allen P. Nikora
Jet Propulsion Laboratory
4800 Oak Grove Drive
Pasadena, CA 91109-8099
Mail Stop 264-805
vox: (818)393-1104
fax: (818)393-7830
Allen. P. Nikora@jpl.nasa.gov

John C. Munson
Computer Science Department
University of Idaho
Moscow, ID 83844-1010
vox: (208)885-7789
fax: (208)8885-9052
jmunson@cs.uidaho.edu

ABSTRACT

Over the past several years, significant effort has been devoted to the process of predicting software system fault content during the earlier development phases. Much of this work has involved relating structural characteristics of software systems (e.g. complexity measurements of the source code and design) to the number of faults in the system. We describe our effort in extending this work beyond the initial software construction. Our area of focus is determining the rate of fault injection over a sequence of successive builds, first observing that software faults may be seen to fall into two distinct classes - some faults are incorporated during the initial coding effort, while others are added in successive software builds. Experience in working with NASA software development efforts is discussed, including practical issues in obtaining data and assuring its validity. One of the most significant topics discussed is the methodology for the precise determination of a fault condition and the mapping of software faults to individual program modules. We examine the results obtained to date, and conclude with a description of our plans to extend this work in the future.
Coeur d'Alene, ID
May 11-13, 1997
Annual Oregon Workshop on Software Metrics

John C. Munson
CSU, Idaho
University of Idaho
Computer Science Department

Allen P. Nikora
JPL, NASA
Pasadena, CA
California Institute of Technology
Jet Propulsion Laboratory

A CASE STUDY
Finding Faults
with Faults.
Future Work
Risk Assessment
Rate of Fault Injection
Fault Surrogates
Counting Faults
Fault Content Model
Motivation

Overview
Motivation

- Development process using these improved measures
- Assert better control over the system structure and the system to assess operational risk
- Develop improved methods of measuring a software

Goals:
- To manage better a development effort, must be able to trade off between development process options, system structure and development process options, and quality, while development still in progress.
- Current methods of predicting software reliability don't account for system's structure and development process characteristics.
\( \bar{\nu} \) represents the number of faults already in the system.

\( \bar{\nu} \) represents characteristics of the software development process.

\( \text{SYS} \) represents characteristics of the software product.

\( f \) and \( g \) are functions:

General Model Formulation:

Fault Content Model
Fault Content Model (cont'd).

General Model Formulation (cont'd):
Ability to co-opt accurately faults.

Development and use of model requires the development of developers.

- Predictions are in terms meaningful to users and ability to compute confidence values.
- Ability to become available.
- Detailed information about product, risk, and process development effort.
- Ability to refine and update predictions as more data become available.
- Ability to make resource/risk tradeoffs earlier in the development of the new model.
Fault Types

Fault Type Composition

Post-Development Fault Identification

Fault vs. Failure Counts

Counting Faults
Actual situation is shown on next slide.

Variance distribution of number of faults per failure had low

Number of faults related to number of failures

Failure counts could be used if:

Fault vs. Failure Counts

Fault vs. Failure Counts

Fault vs. Failure Counts
Development - Post-Development Fault Identification

Identifying Faults

Available data

Institutional Problem Reporting Systems

SCCS files for all delivered versions of software

Assume changes in increment "x" to fault solely to fault

MD "null repair" in increment "x" in response to a failure

Identifying faults

Difference between "x-1" and "x" identifies changes

For each test increments in which faults occur

(cfaults)

Post-development fault identification is primarily a manual process
Fault Types

- Constant definition deletion
- Definition and use of new constants
- Faults involving constants
- Assignment of a different value to a variable
- Variable deletion
- Type from float to double
- Redefinition of existing variables (e.g., changing definition and use of new variables)
- Faults in variable usage

Response to failure reports
Taxonomy based on corrective actions taken in
Deletion of a procedure or function
Addition of a procedure or function
Incorrect order of execution
Removal of source code block

Redefinition of condition for execution (e.g., change "if!" to "if! \rightarrow = 9")

Addition of execution paths within a source code block within a source code block
Deletion of erroneous conditionally-executed path(s)
Addition of new source code block

Control Flow Faults

Fault Types (cont'd)
Counts as two faults, since two paths were removed.

Fault Types (cont'd)

Execution paths from a code block.

Control flow fault examples - removing.
Counts as three faults, since three paths were added.

Control flow examples (cont'd) - addition fault

Fault Types (cont'd)
Introduction of Faults

As faults

Programs

Designs

Specifications

These errors are manifested in

Programmers

Systems Designers

System Analysts

Their tasks

People make errors in the interpretation of
Anticipates distribution of faults in modules
Varies directly with faults
Obtained estimate from past development efforts

We seek to develop a fault surrogate
Fault count in any reasonable manner
May only use past experience to anticipate
Found
Can never know when all faults have been

Faults and Uncertainty
Fault Granularity

- Faults should be maintained at the module level.
- Configuration management is at the module level.
- Complexity measurements are at the module level.
- Changes to metrics are measured at the module level.
- The granularity of fault measurement must be the same as other metrics.
Mapping of Faults

- Global data
- Components
- Include problem
- One fault - several modules
  Fault extent within single module
- One fault - one module
Deriving a Fault Surrogate From Complexity Metrics
### Standardized Metrics are Transformed to become:

<table>
<thead>
<tr>
<th>3.84</th>
<th>0.89</th>
<th>0.54</th>
<th>0.18</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.32</td>
<td>0.52</td>
<td>1.41</td>
<td></td>
</tr>
<tr>
<td>1.47</td>
<td>2.42</td>
<td>5.64</td>
<td>3.78</td>
</tr>
<tr>
<td>3.15</td>
<td>1.73</td>
<td>0.97</td>
<td>0.68</td>
</tr>
</tbody>
</table>

### When Standardized become:

<table>
<thead>
<tr>
<th>5</th>
<th>54</th>
<th>2</th>
<th>45</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>16</td>
<td>179</td>
<td>159</td>
<td>48</td>
</tr>
<tr>
<td>17</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### The nineteen original metrics:

- Identify distinct sources of variation
- Principal components analysis used to
- Selected for their relationships to faults
- Software metrics are highly correlated

---

**Surrogate Selection of Metrics for Fault**
domain metrics

- Relative Complexity is a weighted sum of the
  domain so related
- Compute domain metrics for each complexity
  related to software faults
- Identity complexity of mail so that are closely
  relationship with measure of faults
- Each complexity domain has a distinct

A Unitary Measure of Software Complexity
\[ \chi \left( \frac{\gamma}{f} \right) \zeta = \xi \]

For each program module as follows:

- A relative complexity value, \( \gamma \), will be computed.
- Metric (factor scores) will be taken on selected metric primitives.
- For each program module, a set of measurements will be the domain of standardization.
- Transformation coefficients will map the raw metric onto a set of domain metrics.

Computation of Relative Complexity
Correlates well (0.90) with measures of software faults
Validation of the relative complexity concept
Relative complexity is an extensible metric
Models modules may be ordered by relative complexity
Surrogate Relative Complexity As a Fault
Faults will also change during development, then the number of latent faults will contain a large number of faults. If the relative complexity of a module is high then it is selected because they were related to faults. The metrics that comprises relative complexity will contain a large number of faults. If the relative complexity of a module is high then its surrogate Relative Complexity As a Fault.
<table>
<thead>
<tr>
<th>Module</th>
<th>Domain 1</th>
<th>Domain 2</th>
<th>Domain 3</th>
<th>Domain Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.36</td>
<td>0.37</td>
<td>0.36</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0.34</td>
<td>0.35</td>
<td>0.34</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>0.32</td>
<td>0.33</td>
<td>0.32</td>
<td>6</td>
</tr>
<tr>
<td>9</td>
<td>0.30</td>
<td>0.31</td>
<td>0.30</td>
<td>9</td>
</tr>
<tr>
<td>12</td>
<td>0.28</td>
<td>0.29</td>
<td>0.28</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>0.27</td>
<td>0.28</td>
<td>0.27</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>0.26</td>
<td>0.27</td>
<td>0.26</td>
<td>14</td>
</tr>
</tbody>
</table>

Relative Complexity
Sample Hall/S Programs Ordered by

DR Count
of each program module

- We must identity and measure each version

- Sense

- They are different programs in a very real over time

- We are really working with many programs

- Maintaining a program (maintaining a program)

- We assume that we are developing

Moving Target

Software Evolution: Measuring a
build must be removed from the current count
Faults attributed to modules not in the current build
main sequence to the current build
Fault count should contain only those faults on the build
These faults must be removed from the main sequence
Some faults are eliminated on branch builds
pruned
These fault counts must be removed when the branch is
Some faults are inserted during branch builds

 Managing Fault Counts During Evolution
Baseline

Measure change in fault surrogate from initial

Relative complexity of each build

Initial build as a baseline

Change across builds

Measurements, such as relative complexity,

Software changes over software builds

Project

Baseline a Software Development
Process

Change as a Fault Injection
Relative Complexity
Measuring Software Evolution by
We are continually adding faults to the software in proportion to the complexity of the changes.

The complexity of the system at the first immediately after the first integration test of a software system is its complexity will rise in relation to the baseline complexity of the system at the first build.

The fault injection process during Software Development.
Thus

\[ \frac{R}{d} = \lambda \]

The fault potential of a module is proportional to the fault surrogate module at the first build. Initially each program module has a number of faults proportional to the fault surrogate. Initially the relative complexity fault surrogate represents the proportion of faults in the system.

\[ \sum_{N}^{1=d} R = \text{Total system complexity is initially} \]

Baseline Fault Assessment
If the changes to code to fix faults have not
the system
faults removed in the module on the build of

Then the module will have had faults removed
at the build of the system

Let represent the total number of faults to and

Changes to Faults

![Diagram of a diamond shape with text]
New estimate for proportion of remaining faults is

\[ \frac{1}{1+f} s = \frac{1}{1+f} g \]

The total change over \( I + f \) builds is

\[ \frac{1}{1+f} \nabla \sum_{\gamma=2}^{\kappa} \frac{1}{1+f} s \]

The change relative complexity from build \( f \) to build \( I + f \)

\[ \left| \frac{1}{1+f} d - \frac{1}{f} d \right| = \frac{1}{1+f} \nabla \]

New faults will be injected into the system in

Rate of Fault Injection
Faults can be mapped to program sets of modules
Different functionalities execute differently
A fault can only cause a failure if it is
\[
\begin{array}{cccccc}
\top & \top & \top & \top & \top & \top \\
\top & \top & \top & \top & \top & \top \\
\top & \top & \top & \top & \top & \top \\
\top & \top & \top & \top & \top & \top \\
m_1 & m_2 & m_3 & m_4 & m_5 & m_6 \\
\end{array}
\]

ASSIGNS (f, m)

defign
Program Functionality

- The set of relations on relation `F`.
- The set of relations on the relations in a set of operations, `F`.
- Programs implement the operations in terms of a set of users specifying their needs.
- There is a relation implementing over `F`.
- There is a relation `f(y)` true if `F`.
- Functionality `f : F` is used to implement operation `O ∈ O`.
- Program Functionality
Operation 0₂ is implemented using functions \( f₂ \) and \( f₄ \)

Operation 0₁ is implemented using functions \( f₁ \) and \( f₂ \)

\[
\begin{array}{ccc|c}
1 & 1 & 1 & 0₂ \\
1 & 1 & 1 & 0₁ \\
\text{f₁} & \text{f₂} & \text{f₄} & \Delta \times 0
\end{array}
\]

\text{IMPLEMENTATION (o,')\text{SPECIFICATION}
\{ i > (m', f) d > 0 \lor (m', f) \} \text{ASSIGNS} \cdot \mathcal{P} \ni f \in M : m = d_n\\

Some program modules may potentially execute when a given function is expressed.

\{ i = (m', f) d \leq (m', f) \} \text{ASSIGNS} \cdot \mathcal{P} \ni f \in M : m = d_n\\

Some program modules are indispensable associated with a functionality.

\{(m', f) \} \text{ASSIGNS} \cdot \mathcal{P} \ni f \in M : m = 0\\

The functionality of some program modules will execute regardless of

Program Modules

Functional Classification of
where

<table>
<thead>
<tr>
<th>${\cdot^1 m}$</th>
<th>${\cdot^6,^9 m}$</th>
<th>${\cdot^3 m}$</th>
<th>${\cdot^4 m}$</th>
<th>$\cdot^4 f$</th>
</tr>
</thead>
<tbody>
<tr>
<td>${\cdot^6,^9 m,^1 m}$</td>
<td>${\cdot^9 m}$</td>
<td>${\cdot^3 m}$</td>
<td>${\cdot^4 m}$</td>
<td>$\cdot^3 f$</td>
</tr>
<tr>
<td>${\cdot^9 m,^3,^6 m}$</td>
<td>${\cdot^3 m}$</td>
<td>${\cdot^3 m}$</td>
<td>${\cdot^4 m}$</td>
<td>$\cdot^2 f$</td>
</tr>
<tr>
<td>${\cdot^7,^3,^6 m}$</td>
<td>${\cdot^3 m}$</td>
<td>${\cdot^4 m}$</td>
<td>${\cdot^4 m}$</td>
<td>$\cdot^1 f$</td>
</tr>
<tr>
<td>${\cdot^1 m}$</td>
<td>${\cdot^4 m}$</td>
<td>${\cdot^3 m}$</td>
<td>${\cdot^4 m}$</td>
<td>$\cdot^0 f$</td>
</tr>
</tbody>
</table>

FUNCTION

**Functions**

Relationship of Modules to
The probability distribution of the operational profile is multinomial.

The operations are mutually exclusive. Executing an operation $i$ is the probability that the user is by a user probabilities of each of the operations being executed.

Thus, $P_r(o) = 0$ for $r \neq i$.

The Operational Profile is the set of unconditional operational profiles.
The probability distribution of the functional profile is multinomial.

The functions are mutually exclusive.

Executing program functionality is the probability that the system is functional profile being processed by an operation unconditionally. The functional profile of each of the software is the set of functional profile.
We must measure to determine the distribution program design.

This distribution is directly determined by the underlying distribution is multinomial.

Fixed functionality represents this probability for a

\[ \left( f = A \mid w = \mathcal{N} \right) \]

Let of executing a module / given a certain functionality /

An execution profile is the conditional probability.

Execution Profile
The Module Profile is the unconditional probability that a module will be executed:

\[ \Pr(M) = \sum_j \Pr(M_j) = \sum_j \Pr(M, F_j) = \Pr(F_j) \Pr(M, F_j) \]
(failure potential) will be high prone modules with high probability, the risk

- If a functionality is executed that will run fault

\[ \sum_{u} \mathcal{Q}_{f}^{d} = \Phi \]

- The functional risk of this execution profile is

- Each functionality has a distinct execution profile

\[ \Phi = \Phi - \mathcal{D} \]

- Remaining faults is

At each build, an estimate for the proportion of

**Surrogate as a Loss Function**

**Risk Assessment with Fault**
Reliability

- New potential for modeling software based on risk assessment
- New methodology for regression testing
- Risk assessment for software test
- Functional standards for fault recording

Future Work