A DECISION THEORY FRAMEWORK FOR SOFTWARE FAULT CORRECTION

Wade H. Shaw and Duard S. Woffinden

Department of Electrical and Computer Engineering
Air Force Institute of Technology
Dayton, Ohio 45433

ABSTRACT
This paper presents the use of decision theory concepts in detecting and correcting software faults. The concepts of software faults and failure, symptoms and corrective actions are presented as a way of viewing the software design and test portion of a software development effort. We show how probabilistic estimates of software failure based on project histories can be used in a decision making setting to reduce fault correction cost and time. An algorithm which computes the decision procedure is discussed as well as computational considerations. The impact of the decision theory model on large scale software development efforts is presented.

INTRODUCTION
The cost and effort to maintain software is a well documented phenomenon which has received a great deal of attention [Pressman 1987; Vick 1984; Boehm 1981]. A number of models exist which attempt to estimate delivered errors in software. A particularly fruitful area of research developed in recent years is the application of reliability theory to software fault phenomenon [Musa, 1987]. The concept that software fails in a statistically prescribed fashion is a powerful tool in describing the nature of the software lifecycle.

Useful as reliability concepts may be, they tend to describe the aggregate behavior of software over time and do not attempt to relate the failure process to management or engineering intervention. Specifically, how does one use the probabilistic nature of software failures in choosing a fault correction plan?

To date, little has been done to apply probabilistic models to determine the optimal course of action when a software fault occurs. Although not directly applicable to software failures, some success at prescribing maintenance and repair plans to filed devices has been presented [Pate-Cornell 1987]. This fascinating paper describes the use of Markov models to determine a maintenance policy based on characteristics of the process and the type of maintenance. Unfortunately in the software environment, maintenance is not performed on a schedule, rather at the time of failure!

An alternative approach is to simply use the failure history of a software development effort as a proxy for the future. The idea being that the past is the best estimator of the future. The underlying concept behind the decision theory approach is that knowledge of process's failure history coupled with information about the actions taken to resolve the failure can be used to form a rational repair plan which minimizes the cost to repair code when a software failure is detected.

An important concept currently popular in the literature is the distinction of a software failure versus a software fault. A failure is considered a dynamic departure of the software results from the requirements definition. This general description has to do with the behavior of the software which changes as specifications and the operating environment are changed. A fault, on the other hand, is a defect which causes a failure to occur. That is, a fault is the well known "bug". Clearly, a fault can cause multiple failures and faults are not detected until the system fails to meet specifications in some way.

We choose to view the failure of a software system as symptomatic of an underlying fault(s). The objective of this paper is to propose a method of relating symptoms with alternative debugging actions which lead to the minimal cost/effort to find and correct the root fault. A similar approach has been proposed (Gupta 1987) to maintain mechanical hardware subject to wear-out and breakdown.

The cost and effort expended to correct a software fault is directly dependent on the
experience and insight the designer has in dealing with symptoms in the past. This history should be maintained and used in classical decision theory to predict a least cost course of action given any set of symptoms. This optimal course of action should, in the long term, lead to the lowest cost management action. The accumulated relationships established between symptoms, faults and actions provide insight into more complex design and performance issues.

**DECISION THEORY MODEL**

The decision theory model is characterized by three fundamental components. First, software is considered to possess faults which are a product of design or specification error. These faults are the cause of system failure as defined in the introduction. Second, the failure of software is viewed as a set of symptoms of underlying faults. That is, the software fault causes a failure to occur which the designer or user can perceive as a symptom of the fault. Clearly, the domain of both faults and symptoms is not known but can be built as a dynamic vector based on experience and testing. Finally, there exist a set of debugging actions which the user or designer can take to clear faults. These actions can be considered as responses to symptoms and possess some cost or time to implement. Mathematically, we use a payoff matrix to represent the cost to implement the set of debugging actions and a probability matrix to associate the set of symptoms with the underlying faults.

The objective of the decision theory approach is to determine what set of specific actions should be completed to remove a fault given that a failure symptom has been detected. Clearly, the goal is to specify a finite set of actions over all known symptoms so that the expected cost to remove the potential set of faults is minimized. The mathematical model is given as follows.

Let the fault matrix be be given as:

\[ \pi_i \in \Pi, \; i = 1 \ldots L, \]

where \( L \) is the number of identified faults. For example a fault may be described as a data structure error in software module. Since the fault vector is not known a priori then the faults are initially described in general terms and modified over time.

Let the failure or more accurately the symptoms matrix be defined as:

\[ \theta_j \in \Theta, \; j = 1 \ldots M, \]

where \( M \) is the number of identified symptoms. In software engineering this would be considered the failure mode or any documented error report. Again, the model would begin with broad definitions of failure and fine tune the definitions as a result of testing and debugging.

Let the debugging action matrix be given as:

\[ \omega_t \in \Omega, \; t = 1 \ldots N, \]

where \( N \) is the number of possible actions available to correct a fault. These might be described as "test the loop structure of module x." The actions may be sets of actions or general recommendations. Again, the set of actions is learned over time and the model is updated over the course of software development, test and use.

The cost matrix can be expressed as a two dimensional matrix of payoffs where each element corresponds to the payoff of performing a selected actions given that a fault has been detected. This matrix in effect measures the effort to clear a fault for each potential action that could be taken. Let the matrix be:

\[ \lambda_{n \in A, \; t = 1 \ldots N}, \; i = 1 \ldots M. \]

To determine an optimal set of actions for specified symptom we must estimate the probabilities that a fault is related to a specific symptom. These probabilities are not known a priori but are estimated by trouble reports, debugging efforts or by Bayesian methods based on previous experience, perhaps with other software projects. Generally, the probabilities would be specified by a frequency table or cross tabulation. Regardless of the source or method, the probabilities can be estimated and updated over time. Let this matrix be given as:

\[ p_{ij} \in \Phi = P(\pi_i | \theta_j). \]

The long run probabilities of each fault are easily computed from \( \phi \) and are denoted \( \phi \). The long run probabilities for observing a failure symptom are also available from \( \phi \) and are denoted \( \mu \).

These definitions complete the decision theory model. The objective is to determine a procedure for assigning a debugging action to each failure symptom in order to correct a software fault such that the cost to correct the faults over time is
There will exist \( \delta \in \mathcal{N}^2 \) possible decisions to choose from. We define \( \delta \) as the \( k \)th decision procedure in the set seek a optimal procedure composed of set of actions denoted \( \delta' \) which minimizes the expected average cost.

The procedure is summarized first and then completed step by step. First, the cost as defined in \( A \) is matched with each element in \( \delta \) and the symptoms matrix \( A \) to determine the long run cost to repair a fault given that a failure (symptom) has occurred. Second, the long run costs are then weighted by the probability that a failure will occur. Finally, the set of actions in \( \delta \) which minimize the total cost is selected as \( \delta' \).

Let:

\[
K(\delta_i, \pi_j) \forall \delta_i \in \delta, \forall \pi_j \in \Pi
\]

be the long run cost to repair fault \( \pi_j \) for decision \( \delta_i \). Since we know the probability of a failure \( \delta_i \in \delta \), substitute the cost \( A \) for each action \( \omega_j \), specified by \( \delta_i \), and compute the individual cost per fault per decision.

Mathematically:

\[
K(\delta_i, \pi_j) = A^{(\omega_j)} p_{ji}, \forall i, j.
\]

Step 2 requires the computation of the expected cost over each \( \delta_i \). This expected cost is given as:

\[
EC(\delta_i) = \sum_{j=1}^{n} K(\delta_i, \pi_j) p_{ji}, \forall \delta_i.
\]

Now the minimum cost decision is given as:

\[
\delta' = \min \{ EC(\delta_i) \}.
\]

The minimal value of the set of weighted, long run cost functions represents the set of debugging actions which, in the long run, results in the repair of software faults evidenced by an observed failure. The procedure generates a finite set of recommend action that should be taken for each of the specified failure symptoms. As time elapses and the various matrices are updated with better estimates and more detailed entries, the procedure generates more detailed recommendations for fault correction.

A SIMPLE EXAMPLE

Although the mathematical model appears complex, in reality it is a straight-forward application of matrix algebra. Note that no nonlinear functions are used. This restriction simply implies that payoff costs are additive and linear over the range of interest. Clearly, this assumption may be relaxed by allowing a nonlinear cost function and somewhat more complex mathematics. To illustrate the decision theory approach, a simple example is included. We choose to conserve space and show the results of the computations. A computer algorithm was developed to implement the model and reduce the chances of computational error.

Consider a simple, three module development effort where data has been collected as testing of software modules was completed. Suppose that we allow only two possible failure modes: first, we could display the wrong output but the output could be computed correctly, or we could get the right output but computed incorrectly. If we allow three debugging actions, one for each module, and record four types of software faults as data type error, data structure error, loop control error, and control logic error, then the following tables depict the application of the decision theory approach to determine which module should be debugged based on the state of the system output.

Table 1 shows the cost in terms of days to perform each action given that a fault is in that module.

Table 1. Cost Matrix in Terms of Days to Perform

<table>
<thead>
<tr>
<th>Action (Debug)</th>
<th>Data Type</th>
<th>Data Struc</th>
<th>Loop Control</th>
<th>Control Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mod. 1</td>
<td>0.1</td>
<td>0.5</td>
<td>1.0</td>
<td>2.0</td>
</tr>
<tr>
<td>Mod. 2</td>
<td>0.5</td>
<td>0.2</td>
<td>1.0</td>
<td>0.5</td>
</tr>
<tr>
<td>Mod. 3</td>
<td>2.0</td>
<td>0.5</td>
<td>0.5</td>
<td>0.2</td>
</tr>
</tbody>
</table>

Table 2 depicts the frequency and probability of a failure being caused by a specific fault. This data may have been determined by early testing and changes as experience with the software is accumulated.

Since there are three (N) actions and two failure symptoms (M), there will be nine possible decisions ranging from perform Action 1 for both symptoms to perform Action 3 for both symptoms and all combinations in between. Table 3 shows the results of the \( K(\delta_i, \pi_j) \) calculations for each decision procedure \( \delta_i \).
Now the individual $\Xi(\delta, \pi)$ vectors are post-multiplied by the fault probability vector determined from Table 2 to determine the expected cost ($E(C(\delta))$. Table 4 shows the result of these calculations.

Table 2. Frequency of Faults by Failure Symptom

<table>
<thead>
<tr>
<th>Fault Type</th>
<th>Wrong Result</th>
<th>Incorrect Result</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Type Error</td>
<td>2</td>
<td>12</td>
<td>39%</td>
</tr>
<tr>
<td>Data Structure Error</td>
<td>3</td>
<td>6</td>
<td>25%</td>
</tr>
<tr>
<td>Loop Control Error</td>
<td>2</td>
<td>0</td>
<td>6%</td>
</tr>
<tr>
<td>Control Logic Error</td>
<td>9</td>
<td>2</td>
<td>30%</td>
</tr>
</tbody>
</table>

Table 3. $\Xi(\delta, \pi)$ Calculations

<table>
<thead>
<tr>
<th>k</th>
<th>Data Type</th>
<th>Data Struc</th>
<th>Loop Control</th>
<th>Control Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.10</td>
<td>0.50</td>
<td>1.00</td>
<td>2.00</td>
</tr>
<tr>
<td>2</td>
<td>0.16</td>
<td>0.40</td>
<td>1.00</td>
<td>0.77</td>
</tr>
<tr>
<td>3</td>
<td>0.37</td>
<td>0.50</td>
<td>0.50</td>
<td>0.53</td>
</tr>
<tr>
<td>4</td>
<td>0.44</td>
<td>0.30</td>
<td>1.00</td>
<td>1.73</td>
</tr>
<tr>
<td>5</td>
<td>0.50</td>
<td>0.20</td>
<td>1.00</td>
<td>0.50</td>
</tr>
<tr>
<td>6</td>
<td>0.71</td>
<td>0.30</td>
<td>0.50</td>
<td>0.25</td>
</tr>
<tr>
<td>7</td>
<td>1.73</td>
<td>0.50</td>
<td>1.00</td>
<td>1.67</td>
</tr>
<tr>
<td>8</td>
<td>1.79</td>
<td>0.40</td>
<td>1.00</td>
<td>0.45</td>
</tr>
<tr>
<td>9</td>
<td>2.00</td>
<td>0.50</td>
<td>0.50</td>
<td>0.20</td>
</tr>
</tbody>
</table>

Table 4 indicates that if decision $\delta'$ is taken that the minimal cost (0.453) days will elapse on average to correct a software fault. Therefore $\delta'$ is 2. This decision is summarized in Table 5.

<table>
<thead>
<tr>
<th>Software Failure</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symptom</td>
<td></td>
</tr>
<tr>
<td>Wrong Result</td>
<td>Debug Module 2</td>
</tr>
<tr>
<td>Right Result</td>
<td>Debug Module 1</td>
</tr>
<tr>
<td>Symptom 1</td>
<td>(Symp. 1)</td>
</tr>
<tr>
<td>Symptom 2</td>
<td>(Symp. 2)</td>
</tr>
</tbody>
</table>

Table 5. Final Conclusion

CONCLUSION

This paper has presented a systematic methodology which applies decision theory concepts to the complex problem of software fault correction. The procedure is straightforward, albeit computation intensive. The algorithm is easily automated and generates specific recommended actions based on accumulated information about a project's lifecycle fault behavior.

The limitations of the methodology are intuitive. Where do the choices for actions, faults and failure symptoms come from? Clearly, these attributes are not known in advance! However, this limitation is echoed repeatedly in the software engineering literature and is often quoted as a major obstacle in software reliability research. Nevertheless, this approach is significantly different than the traditional approach of statistically describing the failure characteristics of code.

We believe that the use of decision theory methodology allows the failure history of a project to be coupled with the corrective actions taken to correct failures so that measurements of the effectiveness of corrective action can be obtained. A by-product of this approach is a concise history of the project lifecycle providing considerable insight into the complex nature of software failure, diagnosis and repair. Certainly, one would expect that a specific software effort has failure characteristics which may be used to advantage on similar projects.

In summary, the application of decision theory to software engineering provides an interesting, alternative point of view which relates failure characteristics of software to alternative corrective actions.
This approach is contingent on the premise that there exist rational actions which a software designer can take when software fails and that within this set of actions there exists a particular mapping of actions and failure symptoms which lead to minimal cost management of the fault correction process. This mapping is not constant over time; rather, it adjusts to accumulated experience and re-estimates of associated probability vectors. It is worth noting that the decision theory approach to fault repair is easily ported to any maintenance environment where failure of a system is evidenced by symptoms of underlying faults.

REFERENCES


ABOUT THE AUTHORS

Wade H. Shaw, Jr. is an Assistant Professor of Electrical Engineering at the Air Force Institute of Technology in Dayton, Ohio. He completed his Ph.D. at Clemson University. Dr. Shaw has published several papers in the areas of computer performance evaluation, simulation, decision support systems, software engineering, and project management. He is a member of the IEEE, AIIE, DSI, ORSA, TIMS, and SCS. He is registered professional engineer and has been selected to the rank of Major in the US Army Signal Corps.

Duard S. Woffinden recently completed duty as an Instructor of Electrical and Computer Engineering at the Air Force Institute of Technology. He completed his M.S. degree in Computer Science at the Naval Postgraduate School in 1984. During his assignment as a faculty member at AFIT, he has been involved in teaching, research, and consulting in the areas of software engineering, computer programming languages, and artificial intelligence. He is a Major in the US Army Signal Corps and currently completing studies at the Armed Forces Staff College.