Identifying Error Proneness in Path Strata with Genetic Algorithms

Author
Birt, James, Sitte, Renate

Published
2005

Conference Title
Proceedings Asia-Pacific Software Engineering Conference

DOI
https://doi.org/10.1109/APSEC.2005.69

Copyright Statement
Copyright 2005 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

Downloaded from
http://hdl.handle.net/10072/2635

Link to published version
http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1607181
Identifying Error Proneness in Path Strata with Genetic Algorithms

James R. Birt
School of Information and Communication Technology Griffith University, Gold Coast Campus
PMB 50 Gold Coast Mail Centre, Gold Coast 9726, Queensland
Australia
j.birt@griffith.edu.au

Renate Sitte

Abstract

In earlier work we have demonstrated that GA can successfully identify error prone paths that have been weighted according to our weighting scheme. In this paper we investigate whether the depth of strata in the software affects the performance of the GA. Our experiments show that the GA performance changes throughout the paths. It performs better in the upper, less in the middle and best in the lower layer of the paths. Although various methods have been applied for detecting and reducing errors in software, little research has been done into partitioning a system into smaller, error prone domains for Software Quality Assurance. To identify error proneness in software paths is important because by identifying them, they can be given priority in code inspections or testing. Our experiments observe to what extent the GA identifies errors seeded into paths using several error seeding strategies. We have compared our GA performance with Random Path Selection.

Keywords: Software reliability, genetic algorithms, error seeding

1. Introduction

The purpose of this paper is to study the performance of Genetic Algorithms (GA) in identifying error proneness in path strata. This continues our strategic approach to increase the software reliability by focusing on potential errors with the purpose to enable the conditions for an informed inspection or choice in inspection or testing. We do this in two stages: (a) by attributing weights to the potential sources of errors in software; (b) by finding the most error prone regions in these parts using GA.

In this paper our focus of interest is to investigate whether the depth (layers) of the paths affects the performance of the GA. We do this by considering three strata of path depth: upper third, middle third and lower third and analyzing both the error proneness and the number of seeded errors found by the GA in these layers. We do this in accordance to our Source of Error (SOE) weighting scheme and a clustered error seeding strategy.

Our strategic method does not deal with the testing or test cases per se, but eliminates the guesswork of where to put the weight in testing and inspection, because exhaustive testing is rarely or never at all possible and inspection can be both timely and costly.

The past 30 years have seen a huge growth in the size, complexity and criticality of software code development. Consequently, software reliability and its associated costs for achieving better reliability have greatly increased. Several methods for measuring software reliability are commonly used. They include software reliability models [1], automated oracles [2] and fault detection [3]. Malaiya et al. [4], proposed a link between the levels of testing coverage (that is, the expectation to detect as many errors as possible with a test case) and the reliability of software.

Testing coverage techniques have the limitation that they potentially look at an entire program, but even with reduction methods a large program can still remain intractable for testing [5]. More often than not it depends on the intuition or experience of the tester or inspectors in deciding which parts will be examined first, rather than on information about the likelihood that those parts indeed contain yet undetected errors. Testing comes at a high price and typically requires more than half of the project resources to produce a working program [6]. However, a working program does not always guarantee a defect free program. Depending on the criticality of the program, different percentages of testing coverage are required. This implies that a more effective approach for coverage and testing in general is to focus on the paths most...
likely to reveal faults [7] or the most error prone paths [8].

In order to assess quantitatively the error proneness of software modules we use a method proposed by [8]. This proposed SOE method works in the following way: By representing the paths in the software as a SOE-weighted connectivity matrix, a search algorithm can be applied to these paths for their weight as potential error carriers. The goal is to identify those parts that are most error prone, or contribute most to the overall amount of potential errors.

Finding a path in a large environment can quickly get out of hands even for medium sized software. Therefore, we have to optimize our search environment. Various searching algorithms have been developed for different purposes. GA were developed and formalized by Goldberg [10][11]. The motivation for choosing the GA approach was based on its heuristics for optimizing large and complex search environments, which are characteristics in software code. The different paths in a software program are of different lengths, but traditional GA require fixed length environments to search. To overcome this problem we have made an extension [12] to the traditional GA by adopting strategies from Goldberg et al. [11] Messy GA.

In earlier work we have examined the suitability of GA to find the most error prone path, using the SOE weights in relatively small software examples [12]. We compared the performance of several GA strategies including crossover type, mutation probability rates and selection strategy. The goal was to measure the effectiveness of the GA at identifying a SOE potential error prone path within the top 10% and 25% of all SOE error prone paths. We have compared the performance of the resulting GA to both local search, and an exhaustive search technique. We concluded from those experiments that the GA selection performance was better in more complex search environments and that population size greatly influences the effectiveness of the GA. We also conducted a study to examine the effectiveness of the GA, using different GA strategies, and randomly seeded errors [13]. Our measure of success was the amount of seeded errors in the most SOE error prone paths identified by the GA. From that experiments we concluded that GA can identify the parts in a program that contain on average more than 65% of randomly seeded errors.

In this paper we take a different approach to error seeding by using a clustered error seeding strategy. Our clustered approach is derived from existing models that were developed based on experimental research results [3][14][15]. These authors conducted different experiments to determine the ratio of errors and modules, or conversely, the number of modules containing a given ratio of errors. Their experiments revealed that in most cases a small number of modules contain the majority of errors.

It should be noted that our method aims only on the quantitative identification of the most error prone part to make it possible that they be given priority in inspection or testing, but it does not deal with testing or inspection itself, let alone with the design of test cases. This provides the information for an informed decision. The implications and benefits of determining the most error prone parts are many. By identifying these parts it increases the efficiency of inspections and testing, by giving priority in software quality assurance activities. At the same time it contributes to more accurate effort and cost estimations that in turn lead to improvements in software reliability.

The paper is structured in the following way: Section 2 provides a brief background on the SOE framework, which is the groundwork for quantifying error prone paths. Section 3 outlines the search environment and the details the GA implementation. Section 4 describes the experiments. Section 5 analyses and discusses the results, and finally draws the conclusions in section 6.

2. Quantifying error proneness overview

This section explains briefly the SOE framework as proposed by [8] for calculating the error proneness in software modules and how it collaborates with the GA. We summarize it here again, for the sake of completeness and better understanding.

There are several problems with the current methods of testing coverage; they are (i) testing coverage covers only a portion of the software for the level of coverage required; (ii) exponential time to test or testing all paths and (iii) cost effectiveness, that is, the process might not justify the time taken to test. At first glance this does not appear to be a problem. However, how do we know that what we are testing is the best portion, that is, the most error prone? If we have a coverage level of 80%, does this really mean the most error prone 80% has been covered or is it just any 80% overall (with the most error prone possibly remaining in the untested 20%)? The answer is that we do not know, unless we have quantitative information about the contents of errors. We do know that some software is easier to write than others, and is less error prone, while more difficult and complex software is more prone to error.

A range of complexity measures have been designed, but they do not provide a quantitative answer.
that translates on which parts of the software the testing efforts should be concentrated. To be able to find the most error prone portions in a quantitative way, it is necessary to assess the chances that a programmer can introduce an error in an instruction. This is done by applying a SOE analysis as shown in the example of "Figure 1".

Figure 1: Analysis of simple loop illustrating the potential for at least four SOE [8]

This construct, for example, has at least four SOE (if those were the only possible sources of error). For initial simplicity an equal probability for each error is given, though this is intended to be refined at a later stage to involve actual statistical occurrence. All other programming constructs can be analyzed in a similar way, yielding SOE values that are specific to an instruction [8]. The method provides the framework for SOE quantification; it is simple but powerful and can be obtained from the code in an automated way. While this method provides the minimum theoretical SOE, additional information from test design strategies, risk analysis and historical data (if available) can be used to calibrate and fine tune the model by adding weights to the theoretical SOE values.

By summing the SOE values (weighted or theoretical) for each instruction of a software module, it is possible to quantify the SOE for the different paths that the software could step through in its execution. The software structure is then represented as a matrix whose heaviest paths can be identified. We do this by applying the GA on the constructs sparse matrix representation. With this information, the paths can be chosen strategically from the most error prone paths first to the less error prone ones to whatever required extent e.g. as a percentage, and to the satisfaction of the coverage requirements of the software.

3. Search environment and GA

This section briefly explains the preparation of the search environment for the GA and provides a brief introduction to GA and extensions for variable length GA.

3.1. Search environment for GA

To apply a search algorithm to find the most error prone paths the source code being examined must be transferred into an environment for searching that is suitable to apply the GA to the different paths. For simplicity a tree structure is used to represent the executing sequences in the software [6][16]. This tree is represented as a binary tree with paths of varying length and stored as a connectivity matrix for searching. For efficiency we use a sparse matrix. This environment is suitable for the GA search [12][13].

3.2. Path Identification with GA

GA are mathematical constructs based on survival of the fittest. Each generation consists of a population of character strings, or variables, in terms of an alphabet that are analogous to the chromosomes in DNA. Each individual character or gene represents a point in a search space and a possible solution. A fitness score is assigned to each solution representing the abilities of an individual to compete. The individual with the optimal (or more generally the near optimal) fitness score is sought.

The GA aims to use selective breeding of the solutions to produce offspring’s that are more fit than their parents by combining information from the chromosomes. This is achieved by using crossover and mutation of the strings. The choice or design of crossover and mutation strategies influences the effectiveness of a GA to the individual application case. By using different crossover strategies we can study the effect of diversity on the population. Eventually, when the population has converged and is not producing offspring’s that are noticeably different from those in previous generations, the algorithm is said to have converged to a set of solutions to the problem.

Using the SOE as a basis of fitness, a GA can be employed to find the most error prone paths that is, the paths with the highest SOE. The advantage of our method is that the structure of the software, or its components, can be represented as a connectivity matrix, whose paths are represented by a string that is made of zeroes and ones.
We used variable length GA as they perform better in a large and variable sized search environment, and map to the paths of different lengths in a program. The genomes used for the GA are the paths constructed through the connectivity matrix by a 0-1 string. They represent a right (0) or left (1) direction using the tree structure of the software system under examination.

Using the small example from Figure 2 this would give us chromosomes such as the paths 1-3-5 = 00 or 1-3-4-6 = 011. It is this sequence of zero-and-one genes that is used to develop a chromosome for application with the GA search approach. Because nodes are weighted with the SOE, a path is explored and its corresponding SOE values read and summed. If it is heavy in SOE, that is, if the path has potentially high error proneness, then the GA will reward it. Paths are drawn from the result of the GA algorithm. The paths can now be sorted in decreasing order according to their SOE content. This information together with the cumulative percentage of the code that corresponds to the paths selected by the GA, is now ready to be used in the Software Quality Assurance process in whatever way for a desired coverage.

There are two main problems facing a GA in this environment, (i) variable length chromosomes and (ii) local minima. We have investigated strategies for resolving local minima include using different selection, crossover and mutation types; varying rates of mutation, crossover probabilities and population size [12][13].

In terms of variable length chromosomes, with a standard fixed length GA and balanced tree structure, the evaluation of the chromosomes can occur without problem; this is because all gene values (directions in a tree) are valid. The issue with using a fixed length GA is that a source code environment is not fixed in chromosome (path) length. This can be seen clearly in Figure 2 where paths of different lengths are shown. Therefore, special strategies need to be applied.

We have used variable length GA to solve this problem [12][13]. Variable length GA uses techniques to overcome the variable nature of the chromosomes and strategies for evaluation. We use similar techniques to the splicing and recombination strategies of Goldberg et al. Messy GA [11].

By using this process the variable path lengths are removed and the functionality of fixed length GA can be adopted.

4. Experiments

In this section we describe our experiments to examine the GA performance in different strata of paths and specific clustered error seeding strategies. These experiments use refined GA strategies and results are compared with random path selection (RP). We do this because RP is often used as a benchmark.

This involves several sets of experiments. We want to find out how many seeded errors or clusters would be found in selecting 80% of the total SOE potential in a set of sample programs. It is important to be aware that 80% of potential SOE errors does not equal 80% of the search environment. Our question follows closely to the Pareto principle: we want to find out whether in our case most times 80% of errors (seeded error clusters) would be found by testing 20% of the most error prone paths in the software.

From this selection of paths containing 80% errors we further determine the potential error proneness of the proportioned upper third, middle third and lower third software paths identified by our GA approach and the number of seeded errors contained in these layers.
By obtaining an average from the 80% selection we also identify how many similar SOE paths are present in the sample set by examining one standard deviation from the mean.

The reason for dividing into strata is to determine if more errors could be located in the top middle or bottom portion of the paths. This quantitative information can then be used to further enhance our method by determining where potential strategy changes can be made or which regions can be weighted with more potential errors.

Before we go any further we have to make the following distinction for the sake of clarity, despite its redundancy: we refer to SOE errors to the potential sources of errors as a theoretic magnitude that we attribute according to our (or any) weighting scheme. We refer to seeded errors as those that we seeded into the code according to an error seeding strategy.

Table 1 summarizes our refined GA strategies adopted for the following experiments.

**Table 1: GA-strategies for identifying potential error prone paths**

<table>
<thead>
<tr>
<th>Crossover probability</th>
<th>70%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutation probability</td>
<td>1/pathlength [%]</td>
</tr>
<tr>
<td>Selection type</td>
<td>Roulette</td>
</tr>
<tr>
<td>Crossover type</td>
<td>Uniform Crossover</td>
</tr>
<tr>
<td>Population size</td>
<td>30</td>
</tr>
</tbody>
</table>

For the experiments a crossover probability of 70% using uniform crossover and a mutation probability of the reciprocal of the path length is used. The selection strategy adopted was Roulette wheel with a population size of 30. These values have been selected based on insight gained from our previous research [12][13].

Using the strategies in Table 1 to initialize the GA, we conducted three comparison experiments identifying the effectiveness of the GA vs. the RP technique.

In experiment one we compare the performance in the upper-third layer. In experiment two we compare the performance in the middle-third layer. Finally in experiment three we examine the lower-third of the paths.

In all experiments we mimicked realistic error occurrences by seeding in clusters following a range of clustering models developed by Fenton et al. [3], Basili et al. [14] and Endres [15]. By using the average errors per module from these papers we derive our error clustering model. Table 2 shows our error distribution percentage per module.

The samples shown in Table 2 refer to our experimental samples listed in Table 3. For example if we take sample A, when the errors are seeded two modules will contain 87.48%, four (8.37%), six (2.07%), eight (1.56%), ten (0.40%) and twelve (0.23%) of the seeded errors.

Table 2: Error distribution per module

<table>
<thead>
<tr>
<th>Sample</th>
<th>Number of modules seeded</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2 4 6 8 10 12</td>
</tr>
<tr>
<td>B</td>
<td>1 2 3 5 6 7</td>
</tr>
<tr>
<td>C</td>
<td>2 4 5 7 9 11</td>
</tr>
<tr>
<td>D</td>
<td>2 3 5 7 9 10</td>
</tr>
<tr>
<td>E</td>
<td>3 5 8 11 14 16</td>
</tr>
<tr>
<td>F</td>
<td>3 6 9 12 15 18</td>
</tr>
<tr>
<td>G</td>
<td>1 2 3 4 5 6</td>
</tr>
<tr>
<td>H</td>
<td>2 4 6 8 10 12</td>
</tr>
<tr>
<td>I</td>
<td>1 2 2 3 4 5</td>
</tr>
<tr>
<td>J</td>
<td>1 3 5 6 8 9</td>
</tr>
</tbody>
</table>

For this experiment we seeded the errors according to the distribution in Table 2. The lines of code for all modules to be seeded are placed in a pool for seeding. We did this to allow all lines of code in the selected modules an equal chance to be error seeded. It must be noted that the lines of code to be seeded are randomly selected in the chosen modules. Future work may examine prioritizing these lines of code from most to least error prone using the SOE analysis.

This set of experiments was conducted on a standard Pentium 4 using MatlabTM. All experiments were conducted using 40 seeded errors per KLOC. We chose this number based on the research by Humphrey [17], which states that non PSP trained people produce around 40 errors per KLOC. For the experiments ten sample open source C programs were used; they are summarized in Table 3.

It is important to note that complexity in our case does not mean cyclomatic complexity, but the number of decision points relating to the length of the SOE paths. Branches refer to the number of branching statements (if...else...switch) in the code. Maximum nesting depth is important as this can greatly affect the number of SOE in a construct and is therefore
represented. Calls refer to user function calls in the software. Conditions refer to conditional statements.

**Table 3: Complexity of sample cases used**

<table>
<thead>
<tr>
<th>Sample</th>
<th>Branches</th>
<th>Max Nesting Depth</th>
<th>Calls</th>
<th>Conditionals</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>77</td>
<td>4</td>
<td>46</td>
<td>9</td>
<td>2675</td>
</tr>
<tr>
<td>B</td>
<td>109</td>
<td>5</td>
<td>57</td>
<td>13</td>
<td>3032</td>
</tr>
<tr>
<td>C</td>
<td>167</td>
<td>4</td>
<td>40</td>
<td>17</td>
<td>3052</td>
</tr>
<tr>
<td>D</td>
<td>193</td>
<td>4</td>
<td>56</td>
<td>44</td>
<td>4435</td>
</tr>
<tr>
<td>E</td>
<td>307</td>
<td>7</td>
<td>46</td>
<td>8</td>
<td>9675</td>
</tr>
<tr>
<td>F</td>
<td>217</td>
<td>6</td>
<td>44</td>
<td>22</td>
<td>11442</td>
</tr>
<tr>
<td>G</td>
<td>293</td>
<td>6</td>
<td>97</td>
<td>102</td>
<td>11713</td>
</tr>
<tr>
<td>H</td>
<td>462</td>
<td>7</td>
<td>219</td>
<td>103</td>
<td>11919</td>
</tr>
<tr>
<td>I</td>
<td>498</td>
<td>4</td>
<td>462</td>
<td>36</td>
<td>13261</td>
</tr>
<tr>
<td>J</td>
<td>652</td>
<td>5</td>
<td>512</td>
<td>102</td>
<td>16718</td>
</tr>
</tbody>
</table>

We conducted the experiments by initializing the GA with the program’s total SOE, which is calculated during search environment conversion. The GA was set to identify and select paths to constitute 80% of the total SOE of the sample environments. We repeated this process fifty times using the different error seeds and averaging the results.

5. Results and discussion

This section discusses the results from our experiments. Overall the results from analyzing the whole path support our method quite strongly. We obtained an average detection rate over 85%. This answers positively our question whether 80% of seeded errors can be detected by analyzing approximately 20% of the code. We now look at the experiments with the path strata, for each stratum, and finally we compare GA and RP performance.

In the first experiment, we focus on the top level stratum, that is, the upper third of the paths. Results from experiment one were interesting (see Figure 3). We found that 70% of the selected paths have similar potential SOE values. In this proportion the GA selected an average of 27.1% of the seeded errors. From analyzing the code related to this portion it was found that these paths on average are in the high level (decision modules) functions with typical code constructs that are less error prone. In a speculative manner this may be attributed to decision modules and function calls being less complex that multiply nested loops that would appear more in the lower algorithm strata. We can study this in the future. The conclusions raised from this experiment is that (a) having similar SOE values makes it indifferent for the GA to select the more error prone regions and (b) the distribution of seeded errors is low due to the code statements in this portion.

In experiment two we look at the middle stratum. Results from this experiment were surprising (see Figure 4). We found that 50% of the paths have similar SOE values. This results in a slightly better distinction between the individual paths containing the SOE. In this proportion the GA selected an average of 44.1% of the seeded errors although only 17% are new errors with 27.1% found in the first-third portion. From analyzing the code related to these paths we found that these regions focus more on function calls to lower level algorithms resulting in more distinction in the potential SOE but again low weighted potential for errors. The conclusion raised about this lower than expected error seeding identification compared with the first-third experiment is that the upper stratum contains more shorter paths with higher SOE weighted code instructions.

In experiment three we look at the lower third stratum. Results from experiment three, were very positive (see Figure 5). We found that the similar SOE path values lowered to 35%. The GA selected an average of 85.4% of the seeded errors with 41.3% of these errors identified in this portion. From analyzing the code related to this portion it is clear why this occurs. On average the majority of algorithms and nested structures occur in this portion. These have high potential to SOE. The conclusion from this is that although we still have 35% similar SOE paths the majority of the SOE weight is present in this portion. This results in higher error seeds and GA identification.

In all experiments the random path performance was significantly lower than the GA this is shown in Table 4.

**Table 4: Comparison of GA performance vs RP performance**

<table>
<thead>
<tr>
<th>Stratum</th>
<th>GA performance %</th>
<th>RP performance %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top 1/3</td>
<td>27.1</td>
<td>5</td>
</tr>
<tr>
<td>Middle 1/3</td>
<td>17</td>
<td>4.2</td>
</tr>
<tr>
<td>Bottom 1/3</td>
<td>41.3</td>
<td>7.6</td>
</tr>
<tr>
<td>TOTAL</td>
<td>85.4</td>
<td>16.8</td>
</tr>
</tbody>
</table>

While the GA was able to identify regions with 85.4% of the seeded error clusters, the RP could only find 16.8%. This is expected as the random path selection does just that: it samples randomly from the search environment, without any strategy. We only use the random path selection as a baseline to our GA approach.
Figure 3: Comparison of GA and RP performance in identifying seeded errors in the top path layer. Ten samples are examined with the GA selecting 27.1% and RP 5% errors.

Figure 4: Comparison of GA and RP performance in identifying seeded errors in the middle + top path layers. Ten samples are examined with the GA selecting an additional 17% and RP 4.2% errors in this layer.

Figure 5: Comparison of GA and RP performance in identifying seeded errors in the whole path strata. Ten samples are examined with the GA selecting an additional 41.3% and RP 7.6% errors in this layer.
6. Conclusions

In this paper we have investigated whether performance in finding error prone regions of the GA is different in higher or lower strata of the paths of a program. We conducted our experiments on ten Open Source C programs using previously developed and refined GA strategies to investigate the effectiveness of the SOE-GA technique. To test the performance we have used well known realistic clustered error seeding methods. In our strata experiments we have compared Genetic Algorithms with Random Path selection. The goal of our experiments was to find (a) how many seeded errors would be contained in a selection of the 20% most SOE-heavy paths and (b) how many errors is in each stratum of the selected paths. The results revealed that an average of greater than 85% of the errors seeded to the most potentially error prone regions are within the 20% most SOE heaviest paths. We found that the GA was more effective at selecting paths containing seeded errors when compared with the capability of random path selection. We also found that in these 85%, the lower third of the paths contained 41.3% of the errors. This clearly identifies that the bottom stratum in the paths is the major contributor to error proneness. The identification of error prone paths using Genetic Algorithms and potential Sources of Errors in software is an important pre-processing step to enable informed decision because it allows priority to be given to these regions in subsequent inspection or testing. Last but not least, we do not claim that our method of allocating SOE is the only possible way, but the results of our experiments clearly demonstrate that there is merit in allocating a quantitative measure of errors to code, because only then a method of identifying the most error prone parts can be applied. Future research will be aimed at refining our methods for faster performance.

7. References


