Automated Fitness Guided Fault Localization

Proceedings of the 5th Annual ISC Research Symposium
April 7, 2011, Rolla, Missouri

Josh Wilkerson
Natural Computation Laboratory
Computer Science Department
Missouri University of Science and Technology
Rolla, Missouri
jw46@mst.edu

Abstract—Software fault localization is an essential and expensive process in software correction, which motivates the design of analysis tools that automate this process as much as possible. This paper presents the Fitness Guided Fault Localization (FGFL) system, a novel approach to fault localization that employs an ensemble of software analysis techniques guided by a fitness function to perform automated fault localization. The FGFL system focuses on software for which a fitness function can be derived either from formal specifications or in some other fashion. The system currently employs an ensemble of three fault localization techniques: trace comparison, a basic execution slice-based technique; trend based line suspicion, an enhanced version of the Tarantula technique designed to exploit the fitness function; and run-time fitness monitor, a technique that tracks changes in fitness during the execution of the program. These techniques are described in detail as well as the method for combining the results from the techniques into a unified recommendation for the fault location. Experimental results are presented that demonstrate the applicability of fitness guided fault localization to automate this important phase of software correction in general, and the potential of the FGFL system in particular.

Keywords—software/program debugging, automated fault localization, fitness function, Tarantula, execution slicing

I. INTRODUCTION

Fault localization is an essential step in software debugging and it is also the most expensive step in this process [1]. The high cost of fault localization is due to the fact that in many cases software errors are located manually employing software analysis tools and techniques. Therefore the task of fault localization would greatly benefit from automation. This serves as motivation for the development of automated tools and techniques that either assist in or autonomously accomplish fault localization.

This paper presents the Fitness Guided Fault Localization (FGFL) system, which focuses on fault localization in software for which a fitness function can be derived and source code is available. This derivation can be from formal (or informal) specifications, an oracle (e.g., the software developer), or some other source. The concept of the fitness function is borrowed from evolutionary algorithms, which work by maximizing a user supplied fitness function that encodes the problem being optimized. In the FGFL system, the fitness function should both indicate when the software is performing correctly and quantify the degree of error when software operates incorrectly. The FGFL system novelly employs the fitness function to guide dynamic analysis of the software in question.

The FGFL system employs an ensemble of dynamic analysis techniques to perform fault localization. Currently, three such techniques have been implemented with more planned in future work. The techniques currently implemented are: trace comparison, trend based line suspicion, and run-time fitness monitor.

Each FGFL technique can be activated or deactivated for a given run, which allows the user to omit a technique if it is expected to not be applicable for a particular program. The results for activated techniques are aggregated using a confidence based voting system in which each technique has a number of votes it can potentially apply to lines in the software suspected to contain the error.

This paper presents a series of preliminary experimentation on a prototype of the FGFL system in which it is tested against a variety of seeded software errors. Multiple technique configurations are also tested in an attempt to identify synergies between the techniques for various bug types. The results of these experiments demonstrate the potential of automated fitness guided fault localization and serve as a proof of concept for the FGFL system.

Of the three techniques currently implemented in the FGFL system, two are enhanced versions of established techniques; namely, execution slice comparison (or dicing) [2] and the Tarantula technique [3], [4], [5]. The FGFL versions of these techniques exploit the fitness function in order to strengthen the established techniques. The dicing technique achieves a higher degree of automation through the fitness function, which serves as an oracle that can indicate test cases that pass or fail at run time. The modification of the Tarantula technique to exploit the fitness function allows a higher degree of precision in the results by introducing a gradient to test case performance. The run-time fitness monitor is a completely novel technique, which tracks changes in fitness during the execution of the program.

This paper is further organized as follows: Section II discusses work related to the FGFL system; Section III presents a detailed discussion of the FGFL system as well as the techniques currently implemented in the system; Section IV details the experimental setup for the preliminary testing of the FGFL system and Section V presents the results of these experiments; Section VI summarizes the presented material; and Section VII discusses current work on, and plans for future improvements of, the FGFL system.

---

1 This is an abbreviated version of a paper submitted to the 2011 IEEE Computer Software and Applications Conference.
II. RELATED WORK

Automatic fault localization is a very active research field with a vast amount of research literature. Due to space considerations, only the most pertinent research to the FGFL system is reviewed here. For a more in depth discussion of this field see [6], [7].

Static slicing [8] was proposed as a technique to assist in bug localization by isolating possible areas that can contain an error. Static slicing uses static code analysis techniques to determine the statements that may influence a variable at a given point in the program. Dynamic slicing [9], [10], [11] is similar to static, except that more information is used to determine which statements actually influence a variable reference, rather than the lines that may. An execution slice [12] is the set of lines that are executed for a given test case. The trace comparison technique implemented in the FGFL system is an enhanced version of execution slice comparison [2], which takes the set difference between the execution slices of a passed and a failed test case, termed the dice. The dice for the negative test case (i.e., the lines unique to the failed test case execution slice) are indicated as likely containing the error. This technique is described in more detail in Section III-A. Also, a new FGFL technique is being considered, which combines dynamic slicing techniques with the fitness monitor; this is discussed further in Section VII.

Delta debugging [13], [14] is an approach that has some conceptual similarities to the trace comparison technique included in the FGFL system. In delta debugging, positive and negative test cases are sought after that minimize the difference between their execution traces. This is accomplished by studying cause-effect relationships by isolating portions of the test case input that are related to the observed error. Iterative delta debugging [15] applies the delta debugging technique to more complex errors that are masked by other errors in the software. Though similar conceptually, the implementation of the trace comparison and the assumptions made regarding the trace are different between the FGFL technique and delta debugging. Primarily, the assumption that minimizing the difference between positive and negative execution traces is beneficial is not held in the FGFL system, an error can just as easily be the result of the code that was not executed as the code that was. The FGFL trace comparison technique takes a more conservative approach, using the largest difference between execution traces in order to avoid overlooking an error. This approach results in a sometimes increased search space versus delta debugging, which minimizes the search space at the cost of sometimes overlooking an error.

The Tarantula debugging tool [3], [4], [5] is a coverage-based technique that monitors, for each line, how many positive and negative test cases execute the line. Lines are assigned suspicion levels based on the proportions of passed and failed test cases that executed the line in question. The trend-based line suspicion technique in the FGFL system is an enhanced version of this technique. In the Tarantula technique, the contribution a test case has to an executed line is binary, i.e., the test case was either passed or failed. With the addition of a fitness function, a gradient is achieved when assessing the test cases that executed a line; test cases that performed very poorly contribute more suspicion to the line than test cases that are near correct. Also, test cases that are passed reduce suspicion in the lines executed. The results are then adjusted using confidence values based on the minimum and maximum possible suspicion obtainable, which is similar to the H heuristic in Tarantula.

III. FITNESS GUIDED FAULT LOCALIZATION

The FGFL system currently operates on a subset of C++ programs (no focus has been placed on fault localization in object oriented software). During execution, the system takes the source code of the software to be corrected as input. Next, a copy of the source code is made that has been instrumented to produce technique-specific output that is used for analysis by the selected techniques. An automatically defined and instantiated class is added to the source code, which is responsible for managing the output details as well as various file streams used to output execution information. Calls to the member functions of this class are automatically placed throughout the source code as is appropriate for the activated techniques.

The FGFL system currently generates random test cases for the program in question, which cover the full range of possible fitness values. The test case generation component of FGFL is defined in a modular fashion, since this portion of the system is highly problem specific. This design decision provides flexibility in the test case generation phase, allowing any user-selected test case generation technique to be used with the system.

When randomly generating test cases, the FGFL system first generates a user defined number N of positive test cases, where a positive test case is defined as a test case that does not demonstrate the buggy behavior according to the fitness function. Next, the range of possible fitness values is divided into equal length segments, defining subset ranges based on the problem specific granularity of the fitness function. The number of regions is user configurable; empirical investigation showed that good results were achieved with four regions (i.e., [0, 0.2], [0.2, 0.4], [0.4, 0.8], and [0.8, 1.0]), which was used for the presented experiments. Further investigation is necessary to determine whether the optimal number of regions is problem dependent. For each subset range, N negative test cases are to be randomly generated that fall into that range. Ideally, this will generate a comprehensive set of test cases according to the fitness function. The system iteratively generates test cases until either all of the segments have been covered or a user defined number of test cases have been considered, in which case the system proceeds with the generated test case set.

Through testing all generated test cases, each selected technique generates a suspicion level for each line in the program being considered. The techniques are given an equal number of freely distributable votes, which are used to indicate where the error is according to the results obtained by the technique. These votes are applied to lines in the program based on the technique’s confidence that the line contains an error. The confidence for a line is a function of the suspicion level for that line and both the maximum and minimum possible suspicion obtainable for the technique, which are described in the detailed technique descriptions later in this section. The number of votes allotted to each technique is equal to the Lines Of Code (LOC) count in the program being considered. Each technique can distribute its votes as it sees fit, with no restriction placed on the number of votes that must be cast (i.e., the techniques can use a fractional portion of the votes allowed). After each technique has distributed its votes across the program,
the votes for each line from all techniques are averaged to obtain the final suspicion levels for each line of code. This aggregation scheme allows for a great deal of flexibility in the FGFL system and also allows the techniques to contribute to the result as much as is appropriate.

A. Trace Comparison Technique

The trace comparison technique in the FGFL system is conceptually based on the execution slice comparison technique presented by Agrawal et al. in [2], though there are a few notable variations in the FGFL version. The trace comparison technique compares the traces of each positive/negative test case pairing, attempting to find where in the execution the negative test case diverged from the positive. Rather than doing a strict set difference to find the negative execution dice, divergent execution paths in the negative traces are detected using a version of the dynamic programming solution to the Longest Common Substring (LCS) problem (where the strings being considered are the lists of line numbers executed when using the test cases) that has been modified to interpret the results differently.

The table generated by the modified LCS algorithm contains a great deal of information regarding the execution traces. Methods to further exploit this information are being investigated and are discussed in Section VII.

The most notable weakness of Agrawal’s technique (and as such the trace comparison technique in FGFL) is the assumption that any branching that occurs in the execution is relevant to the performance of the program. However, it is possible for a divergent path to exist between two positive test cases due to a benign (relative to program performance) branch in the execution. A benign branch that is distant (in terms of executed statements) from the actual error could affect the divergent path a great deal. If this problem is a possibility for a given program, a technique that can overcome this issue is to generate the execution dices between all positive test cases obtained; any lines that are present in one of the generated dices are marked as invalid boundaries for the divergent path. This technique can be further strengthened by generating additional positive test cases.

Suspicion is added evenly to line numbers found in the divergent paths of negative test cases. This technique is not useful in the event that there is no difference between positive and negative test cases, such as when the error is not a branch error.

Cases in which a strict set difference can be used to determine lines unique to the negative execution slice are still being investigated, as this would help to tighten the boundary for the bug location in some situations. For example, consider a program in which there is an error (that, if executed, always results in incorrect results) on a line contained within the then portion of a branch statement. For this program, positive test cases will be those that do not result in the execution of the then portion of the branch containing the error while negative test cases will be those that do execute these statements. In this case, it is safe to remove from consideration all lines that are not unique to the negative test cases, which can be expected to tighten up the accuracy of this technique as well as nullify the effects that many other branch statements have on the technique.

However, consider a program that has an error in a branch statement. In this case both positive and negative test cases will execute the branch statement. Removal of all lines that are not unique to negative test cases from consideration will result in the removal of the line containing the error, likely resulting in misleading results from the technique.

Votes are distributed in this technique evenly amongst the lines indicated by corresponding suspicion levels. There is not currently a mechanism implemented for this technique to apply only a portion of its votes. The binary nature of this technique makes the development of such a mechanism difficult, as a line was either executed by the negative test case or it was not. Possibilities for this addition are discussed in Section VII.

B. Trend Based Line Suspicion Technique

The Trend Based Line Suspicion (TBLS) technique employs the fitness to determine the amount of suspicion to add to all lines in a given execution. Ideally, lines that are executed more frequently in low performance executions will tend to accumulate more suspicion than those executed by both positive and negative test cases. This technique is sensitive to both branch and loop related errors.

The TBLS technique is an enhanced version of the Tarantula technique presented by Jones et al. in [3], [4], [5]. As mentioned earlier, the TBLS technique exploits the fitness function in order to provide additional gradient the the Tarantula technique. In Tarantula, the $H$ heuristic is used to calculate line suspicion based on the number of passed and failed test cases that execute the line. In the TBLS technique, the amount of suspicion applied to a line is a function of the fitness for the test case (i.e., how the program performed with the test case as input). The addition of the fitness function allows for a higher degree of precision in the application of suspicion to program lines.

The TBLS technique uses an equation that is a function of the fitness for an execution and calculates a Suspicion Adjustment Amount (SAA). Currently, it is assumed that the fitness function is normalized to fall in $[0, 1]$, which implies that the fitness function needs to be bounded; however, a modification is possible to allow unbounded fitness functions and is discussed in Section VII. The range of this function should be centered about the midpoint in the fitness range and should output positive SAA for low fitness values and a negative SAA for high fitness values. The linear equation to achieve these characteristics in the SAA is:

$$SAA = -2 \cdot fitness + 1$$ (1)

In this function, the SAA is calculated by inverting the fitness and scaling it to fall between [-1,1]. Preliminary experimentation has shown that this linear equation performs well for the current experimental problem set. Non-linear versions of this equation may be investigated in the future if experimental results indicate a need for this.

Each line in the program has an associated suspicion level (zero initially). The program is executed for a set number of test cases; an even sampling of high/maximum fitness and low/minimum fitness runs is ideal, which is achieved by the segmented test set. For each execution, the SAA is calculated and added to the suspicion levels of the lines in the trace for the run. Ideally, lines that are executed more (or even solely) by the low/minimum fitness executions should have a higher suspicion level, whereas lines executed by either both low and high or just high fitness executions should have a low suspicion.
Algorithm 1 Algorithm for Determining Confidence Values

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>if ( \text{Min}(S) &lt; 0 ) then</td>
</tr>
<tr>
<td>2:</td>
<td>for ( i = 1 ) to ( \text{LOC} ) do</td>
</tr>
<tr>
<td>3:</td>
<td>( S[i] = S[i] +</td>
</tr>
<tr>
<td>4:</td>
<td>end for</td>
</tr>
<tr>
<td>5:</td>
<td>end if</td>
</tr>
<tr>
<td>6:</td>
<td>for ( i = 1 ) to ( \text{LOC} ) do</td>
</tr>
<tr>
<td>7:</td>
<td>( C[i] = S[i] \cdot \text{LOC} )</td>
</tr>
<tr>
<td>8:</td>
<td>( C[i] = C[i] \cdot \frac{</td>
</tr>
<tr>
<td>9:</td>
<td>end for</td>
</tr>
</tbody>
</table>

- \( C \): array that ultimately contains confidence values for each line
- \( S \): array that contains the calculated line suspicion values
- \( S_{\text{max}} \) and \( S_{\text{min}} \): the maximum and minimum suspicion values possible, respectively
- \( \text{Min}() \): returns the minimum value in the argument array
- \( \text{Sum}() \): returns the sum of the values in the argument array
- \( \text{LOC} \): the number of lines in the source program

Algorithm 1 shows how confidence values (and ultimately, votes) are calculated from line suspicions. This algorithm is conceptually similar to the \( H \) heuristic used in the Tarantula technique. The \( S \) array is filled using the traces and SAA values generated by test case executions, after which Algorithm 1 is employed. After the application of this algorithm, the values in the \( C \) array indicate the number of votes that will be applied to the corresponding lines. Lines 1-5 make all suspicion values positive, if necessary. Lines 6-9 are where confidence levels are calculated. The calculation on line 7 determines how many votes will be applied to line \( i \) based on the proportion of suspicion the line contributes to the sum of all suspicion obtained. However, this calculation does not take into account the confidence of the results, i.e., at this point a small suspicion level (relative to the maximum possible suspicion) can receive a large number of line votes if it represents a large portion of the total suspicion that was obtained. This discrepancy is accounted for on line 8, which adjusts the vote amount obtained on line 7 based on how much suspicion line \( i \) obtained relative to the total suspicion possible.

The minimum possible suspicion value, \( S_{\text{min}} \), is attained when a line is in the execution trace of every test case that generates a negative SAA and in no test case traces that generate a positive SAA. Similarly, the maximum possible suspicion value, \( S_{\text{max}} \) is attained when a line is in the execution trace of every test case that generates a positive SAA and in no test case traces that generate a negative SAA. The FGFL system determines these values as it generates test cases.

C. Fitness Monitor Technique

The fitness monitor technique tracks fitness values during program execution. For each test case, the technique calculates the difference between the fitness before and after each line, which shows the effect that the line has on the fitness value. The lines that cause a change in fitness (termed fluctuation lines) and the surrounding lines are of interest in order to monitor them as units rather than individual lines, which is further explained later in this section.

When using the fitness monitor technique, the user supplies information on the variable(s) in the source program that are necessary for fitness calculation (using constructs provided by the FGFL system). Using the provided FGFL constructs this process is simply accomplished in four statements. When creating the instrumented version of the source program, the FGFL system creates member functions in the generated class that are used to output the indicated variables to a file along with line numbers to indicate the last two lines that were executed. Calls to these functions are then placed throughout the source code. After execution, the file containing this information is analyzed and fluctuation lines are noted by the system.

For each fluctuation line, fluctuation regions are determined. A fluctuation region is defined as a set of lines during which the fitness is changing. Each region begins with a single fluctuation line, and additional lines are added to the region until the fitness becomes stable surrounding the region (i.e., adding additional lines would not affect the overall fitness change across the region). It is possible for fluctuation regions to overlap, which allows emphasis to form on lines that are commonly considered to contribute to changes in fitness. The purpose of the fluctuation regions is to attempt to monitor the overall effect of code segments, rather than single lines.

Lines contained in fluctuation regions that overall cause a decrease in fitness receive increased suspicion (incremented by one). Currently, regions that cause an increase in fitness do not affect line suspicion values; however, the possibility of these lines receiving a decrease in suspicion is being considered in future experimentation. After all test cases have been executed, the suspicion values are converted to raw votes and then to confidence based votes using Algorithm 1. \( S_{\text{max}} \) is defined as the total number of fluctuation lines found in all test cases. This value is achieved if the line is contained in every fluctuation region that causes a decrease in fitness. \( S_{\text{min}} \) is zero in this technique, which is achieved if a line is either never included in a fluctuation region or is only included in those that cause an increase in fitness.

D. Result Combination

After all test cases have been executed, the results are combined and presented. The overall votes for each line are calculated by taking the average votes applied to that line by each technique. Averaging was chosen over other aggregation techniques to counter false positives in the individual techniques that were not caught by the confidence based vote adjustments.

IV. EXPERIMENTAL SETUP

The FGFL system has been tested on a statistical analysis program that reads in a set of values, performs a set of statistical calculations on the input values, sorts the input values, and then performs additional statistical calculations on the values that require the values to be sorted. Three versions of the program were used, each of which uses a different sorting technique: bubble sort, insertion sort, and selection sort. The programs are all between 46 and 50 lines long. The bugs for these programs were seeded in and are described in Table I. These experiments are to function as both a proof of concept of the system as a whole and a test to expose strengths, weaknesses, and synergies within the system.
For the presented experimentation a test case was defined as a set of seven values (this length was arbitrarily selected). 10 sets of randomly generated test cases were used to test each program. Each test case set consisted of 75 test cases, of which 15 were positive. The fitness range was divided up into four regions [0, 0.2), [0.2, 0.4), [0.4, 0.8), and [0.8, 1.0), with each region also consisting of 15 test cases.

The specifications for this program would contain assertions indicating that the first set of statistical results are correct, that the values were correctly sorted, and that the second set of statistical results are correct. As such, the fitness function for these programs would consist of components responsible for checking each set of assertions. For simplicity, the sorting portion of this fitness function has been focussed on, that is the section of the program where all of the errors were seeded. The fitness function shown in Algorithm 2 was used in the presented experiments, with the modification to allow descending sorting as well as ascending (the result is scored using both the ascending version (shown) and a descending version and the better result is selected). This modification was made under the rationale that difference between the two results is just a matter of result interpretation and has no bearing on the correctness of the calculations.

Algorithm 2 Ascending Version of Fitness Function Used in Experimentation

\[
\text{Score} \leftarrow 0 \\
\text{MaxScore} \leftarrow \frac{\text{SIZE}^2 - \text{SIZE}}{2} \\
\text{for } i \leftarrow 0 \text{ to } \text{SIZE} - 1 \text{ do} \\
\quad \text{for } j \leftarrow i + 1 \text{ to } \text{SIZE} - 1 \text{ do} \\
\quad \quad \text{if } \text{data}[j] \geq \text{data}[i] \text{ then} \\
\quad \quad \quad \text{Score} \leftarrow \text{Score} + 1 \\
\quad \end{if} \\
\end{for} \\
\text{Fitness} \leftarrow \frac{\text{Score}}{\text{MaxScore} - \text{numMissing(data, input)}}
\]

V. RESULTS

The results obtained from the proof of concept experiments are summarized in Table II. This table shows the average rank of the bug line(s) in the results for each possible FGFL configuration, where a line's rank is defined as the number of lines whose votes are greater than or equal to that of the line in question (if the bug spans multiple lines, then the minimum votes between the lines is used).

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>ID</th>
<th>Bug Type</th>
<th>Bug Line(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble Sort</td>
<td>BBL1</td>
<td>Incorrect Array Index</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>BBL2</td>
<td>Incorrect Array Index</td>
<td>27</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>INS1</td>
<td>Incorrect Index Variable</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>INS2</td>
<td>Incorrect Loop Predicate</td>
<td>25</td>
</tr>
<tr>
<td>Selection Sort</td>
<td>SEL1</td>
<td>Copy &amp; Paste Error</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>SEL2</td>
<td>Incorrect Branch Predicate</td>
<td>27,28</td>
</tr>
<tr>
<td></td>
<td>SEL3</td>
<td>Incorrect Loop Predicate</td>
<td>25</td>
</tr>
</tbody>
</table>

From these results it is apparent that the system has trouble identifying errors in control predicates (e.g., INS2, SEL2, and SEL3). This result, however, is not unexpected given the nature of the techniques in the FGFL prototype. Errors in control statements often require special consideration in fault localization techniques, and no such consideration was made in the FGFL prototype. None of the techniques were able to narrow down the location of the bug in SEL2. This error, in particular, caused a branch predicate to always evaluate to false. This result reveals the system’s need for a technique focusing on statement reachability, which could implicate incorrect predicates as a result.

The single technique that seemed to perform the best was the run-time fitness monitor, with the exception of SEL1. In this program the error was in variable assignments which indirectly influence a drop in fitness; this indirect effect on the fitness caused the technique to report lines that were just a symptom of the true bug. In light of this result, a technique that combines the concept of the run-time fitness monitor with a dynamic slicing technique may be beneficial to the system; this is discussed further in Section VII.

In general, the addition of the trace comparison or run-time fitness monitor technique to another technique (including each other) appears to result in a lower average bug line rank for the technique. The addition of the TBLS technique to a technique, however, seemed to in many cases increase the average bug line rank for the technique. Inspection of the experiment result data indicates that this is largely due to benign branching in the statistical calculation portions of the program. This observation indicates that a similar mechanism as the one described in Section III-A (to account for benign branching) needs to be added to the TBLS technique.

On the non-control oriented bugs the system performed very well. With all techniques active, the system averaged a bug line rank of less than 5 (i.e., a 90% or more reduction in lines from the original source). In many cases the combination of all techniques outperformed the techniques operating independently, which indicates that the ensemble approach of the FGFL system is effective.

VI. CONCLUSION

A prototype of the FGFL system is presented in this paper, which consists of an ensemble of automated fault localization techniques that exploit a fitness function for the faulty software in question. Experimentation was presented that served as proof of concept for both fitness guided fault localization in general and the FGFL.
system in particular. The experimentation was conducted on seven programs with different seeded bugs. The results indicate that all techniques currently in the FGFL system have trouble dealing with control based faults. However, on other fault types the system performed quite well. Individually, the novel run-time fitness monitor technique performed the best out of the three techniques, resulting in 82% or more of the lines being removed from suspicion from the source program (with control faults omitted). With all techniques active, the system yielded a 90% or more reduction in lines from the source. The final conclusion is that employing a fitness function has the ability to improve the existing state of the art in automated fault localization. The results presented here merit investigating all other state of the art automated fault localization techniques for potential enhancement employing a fitness function.

VII. FUTURE WORK

The FGFL system is in a state of active development. The future steps for the system are:

- **FGFL:** The next major step for the FGFL system in general is to begin testing on commonly used program test suites in order to do direct comparisons with state of the art fault localization methods. The Siemens Test Suite is the first set of programs that will be focussed on, after which additional programs from the Software-artifact Infrastructure Repository will be used for further testing.
- **Trace Comparison Technique:** The values in the region of the LCS table that indicate the divergent path could be used to determine substrings within the divergent path that are executed in the positive test case as well. The lines in these substrings could generate less suspicion since they were executed by both positive and negative test cases. This approach would allow a gradient to be used when applying votes for this technique as well as help reduce the assumptions made regarding the behavior of the program in question.
- **Trend-Based Line Suspection Technique:** Currently, this technique assumes that the fitness function is bounded. An extension to allow unbounded fitness functions could use approximated boundaries based on observed performance during testing. After all test cases have been executed, a function based on the maximum and minimum observed fitnesses could generate approximated boundaries. Using these boundaries, the observed fitness values could be normalized to fall in the range of [0,1], after which the technique could generate suspicion and votes as normal.

Also, the experimental results presented indicate a need for a mechanism to reduce the effect that benign branching can have on the TBLS technique. This mechanism could operate similarly as the one described for the trace comparison technique with the same purpose.

- **Dynamic Slicing in the FGFL system:** Through the experimental results it was shown that the performance of a technique can be dramatically effected by errors that indirectly influence the fitness value. A possible new technique for the FGFL system could use dynamic slicing techniques to backtrack to statements that influence highly suspicious lines. This technique could be applied automatically when a notable discrepancy between technique results is detected, like, for example, the results for the fitness monitor on program SEL1 in the presented results.

REFERENCES


