Automated Path Generation for Software Fault Localization

Tao Wang
School of Computing
National University of Singapore
wangtao@comp.nus.edu.sg

Abhik Roychoudhury
School of Computing
National University of Singapore
abhik@comp.nus.edu.sg

ABSTRACT

Localizing the cause(s) of an observable error lies at the heart of program debugging. Fault localization often proceeds by comparing the failing program run with some "successful" run (a run which does not demonstrate the error). An issue here is to generate or choose a "suitable" successful run; this task is often left to the programmer. In this paper, we present an efficient technique where the construction of the successful run as well its comparison with the failing run is automated. Our method constructs a successful program run by toggling the outcomes of some conditional branch instances in the failing run. If such a successful run exists, program statements for these branches are returned as bug report. In our experiments with the Siemens benchmark suite, we found that the quality of our bug report compares well with those produced by existing fault localization approaches where the programmer manually provides or chooses a successful run.

Categories and Subject Descriptors: D.3.4 [Programming Languages]: Processors–*Debuggers*; D.2.5 [Software Engineering]: Testing and Debugging–*Debugging aids*.

General Terms: Algorithms, Experimentation, Reliability.

Keywords: Automated debugging, Program comprehension.

1. INTRODUCTION

Program debugging is an age-old software engineering activity. Many programmers still use traditional source-level debuggers to localize the cause of a program bug. They use these tools to iteratively check program behaviors and hypothesize/confirm the error cause. To improve the state of debugging tools, we need to develop methods where the error cause can be identified from the observable error with a higher degree of automation. The work done in this paper contributes to such a research perspective — we seek to further automate the task of localizing software faults.

One approach to automatic fault localization, which has been explored in the recent times [3, 6, 8, 10, 11, 12, 15], considers certain execution traces of the buggy program itself as representative correct behavior. Fault localization progresses by comparing the failing execution run, which exhibits the observable error, with one that does not. Most of the research in this line of work has focused on how to compare the successful and failing execution runs. They exploit the successful run to find out points in the failing run which may be responsible for the error and for each of those points which variables may be responsible for the error. However, these works do not discuss how the successful run is obtained. Usually, they assume that a large number of successful runs are available and the programmer chooses one of them.

```
1.   v=0;
2.   if (x>0)
3.     u=5;
4.   else
5.     u=v;
6.   printf(''%d'',u);
```

Figure 1: A Simple Example.

In this paper, we automate the process of constructing one successful run from a program's failing run. Given a failing execution run π, our method systematically constructs runs by toggling the outcomes of some of the conditional branch instances in π, until a feasible successful run [1] is obtained. This, indeed, is the key idea of our approach. By evaluating some branch instances in the failing run differently, certain faulty statements may not be executed, leading to a successful run. Branch statements whose instances have been evaluated differently (*i.e.*, toggling of branch condition evaluation) are submitted to the programmer as bug report.

Consider the program fragment in Figure 1 where a `0` value is printed in line 6 corresponding to input `x=0`, deemed by the programmer as an observable "error". We can construct a successful run by altering the outcome of the branch `(x>0)`; such a run corresponds to a positive value of input `x`. Now, by comparing the successful run with the failing run we report the branch `if (x>0)` to the programmer; he/she may notice that the bug fix lies in weakening the branch condition to `if (x>=0)`.

After a run π' is constructed, we need to check whether π' is feasible and successful. Checking whether π' is feasible can be done automatically by using a constraint solver. However, checking whether π' is successful has to be done manually. Otherwise the programmer has to precisely characterize the properties of a successful run; this eases the task of fault localization but places an additional burden on the programmer. It is important to note that methods which require the programmer to choose a successful run from available successful runs (such as [3, 11]) will first require a classification of available program runs as failing or successful. This will typically require even more manual intervention.

[1]A feasible successful run is an execution run which is exercised by some program input and does not exhibit the bug being localized.

In the preceding example, the bug fix gets pinpointed straightaway from the bug report. In reality, there can be various scenarios of the bug report matching or not matching the actual source of bug. If the bug fix involves changing the value assigned to `v` at line 1 of Figure 1, our bug report will not be equally helpful. *How useful is our path generation method and the resultant bug report?* To evaluate our technique, we employed it to localize bugs of the Siemens benchmark suite [7], an established suite of programs with injected faults. Experiments show that our approach can automatically generate successful executions and bug reports within tolerable time. The quality of our bug report compares well with those by previous approaches [3, 11], which require the programmer to manually provide/choose a successful run.

Concretely, the main results of this paper are as follows. We develop a fault localization method based on comparing a failing program run with a successful program run. We automatically generate a feasible successful run according to the failing run. We return the sequence of branch instances evaluated differently in the two runs as bug report. We experimentally evaluate the quality of our bug report, the volume of our bug report and the time overheads of our fault localization method.

The rest of this paper is organized as follows. The next section introduces related work on fault localization. Our path generation algorithm is discussed in Sections 3. Section 4 presents experimental results. Section 5 concludes the paper.

2. RELATED WORK

Recently, there has been a lot of interest in program error localization by comparing successful and failing runs of the buggy program [1, 3, 6, 8, 10, 11, 12, 15]. These techniques often differ in which characteristic of execution runs is used for comparison. The characteristic can be acyclic paths [12], potential invariants [10], executed statements [1, 8], transitions [6], basic block profiling [11] or program states [3, 15]. Jones et al. [8] and Ruthruff et al. [14] have proposed to mine a set of successful and failing runs, and color statements according to the likelihood that the statement is faulty. Liblit et al.'s technique [9] discovers abnormal return value of methods from many runs. Unlike our method, these works require that both failing and successful runs are available before defect localization.

Software fault localization via model checking has also been studied [2, 5]. These works seek to explain the counter-example produced by model checking by invoking an optimization problem. The optimization generates a successful run which is "closest" to the counter-example; this is typically accomplished by an external constraint solver. Note that for these approaches, either the program model needs to closely reflect the behaviors of the actual program, or the approaches risk generating a spurious successful run (not corresponding to any program execution) which necessitates further refinement of the optimization problem.

Zeller et al. present the *delta debugging* algorithm to automatically simplify the erroneous input by removing part of this input [16]. The reduced input (which can be a successful input) usually corresponds to a shorter execution, which may be easier to debug. However, the approach is more suitable for debugging language/text processing programs like compilers or web-browsers where we can get program inputs by deleting parts of a program input.

3. PATH GENERATION ALGORITHM

In this section, we discuss our technique for generating a feasible successful run from a failing run of a program. Recall that a "failing" run is an execution trace which exhibits a specific behavior which the programmer deems as a "bug"; a successful run is an execution trace (for a different program input) which does not demonstrate this bug. A feasible run of a program is a path in the program's control flow graph from start to end which is executed for some program input.

How do we construct a successful run by evaluating differently conditional branch instances of a failing run π? Recall that these branches will be returned as bug report for fault localization. We seek to construct execution runs by evaluating differently branches in π which appear close to the end of π (where the error is observed). Furthermore, the number of branches in π that are evaluated differently should be small. Consequently, given the failing run π, we will first try to evaluate differently the last branch occurrence (call it b_{last}) in π to construct a run π_1. Among all the branch occurrences in π, clearly b_{last} is nearest to the end of π. If π_1 is a successful and feasible run, we return π_1 as the successful run. Otherwise we successively construct other runs by evaluating b_{last} as well as other branch occurrences of π differently. If none of these runs is a feasible successful run, this indicates that the branch at b_{last} might have little relationship with the error cause. So, there is no point in evaluating b_{last} differently. Instead, we evaluate the second last branch occurrence in π differently and carry out the above steps again. This process goes on until a feasible successful run is obtained.

```
1.   if (a)
2.     i=i+1;
3.   if (b)
4.     j=j+1;
5.   if (c)
6.     if (d)
7.       k=k+1;
8.     else
9.       k=k+2;
10.  ......
```

Figure 2: A program segment.

Branches evaluated differently	*Execution run*
$\langle 6\rangle$	$\langle 1,3,5,6,9,10\rangle$
$\langle 3,6\rangle$	$\langle 1,3,4,5,6,9,10\rangle$
$\langle 1,3,6\rangle$	$\langle 1,2,3,4,5,6,9,10\rangle$
$\langle 1,6\rangle$	$\langle 1,2,3,5,6,9,10\rangle$
$\langle 5\rangle$	$\langle 1,3,5,10\rangle$
$\langle 3,5\rangle$	$\langle 1,3,4,5,10\rangle$
$\langle 1,3,5\rangle$	$\langle 1,2,3,4,5,10\rangle$
$\langle 1,5\rangle$	$\langle 1,2,3,5,10\rangle$
$\langle 3\rangle$	$\langle 1,3,4,5,6,7,10\rangle$
$\langle 1,3\rangle$	$\langle 1,2,3,4,5,6,7,10\rangle$
$\langle 1\rangle$	$\langle 1,2,3,5,6,7,10\rangle$

Table 1: Order in which candidate execution runs are tried out for the failing run $\langle 1,3,5,6,7,10\rangle$ in Figure 2

Let us take the program segment in Figure 2 as an example. Assume that the failing run $\pi = \langle 1,3,5,6,7,10\rangle$. The branch occurrences appearing in this run are at lines $1,3,5,6$. Note that the execution run π does not contain multiple occurrences of any program statement; so we do not need to worry about distinguishing different occurrences of the same statement in a path as far as this example is concerned. Now, our method tries to evaluate some of the branches in lines $1, 3, 5, 6$ differently from the failing run π, thereby constructing new execution runs.

Table 1 shows the order in which the branches of failing run π will be evaluated differently leading to new execution runs. Let us assume that none of the new execution runs is a feasible successful run, so that we can elaborate all possible runs constructed by our algorithm. We first evaluate differently the branch at line 6, since this branch is the last one in the failing run. In the next step, the algorithm intends to evaluate differently a branch before line 6 as well as the branch at line 6. It appears that we should now choose line 5, the branch which is the closest to line 6. However, the algorithm cannot choose line 5 at this time, since line 6 is control dependent on line 5. If line 5 is evaluated differently, line 6 cannot be executed. Instead, line 3 is chosen, and the second run is constructed by evaluating differently branches at line 3 and 6. After this, the algorithm tries to evaluate differently a branch before line 3 as well as branches at line 3,6. Thus, line 1 is selected, and branches at line 1,3,6 are evaluated differently. Now all branches before line 3 and 6 have been considered, and no feasible successful run can be constructed. This means that line 3 and 6 might not be related to the error cause at the same time. The algorithm continues trying to evaluate differently branches before line 6 as well as the branch at line 6. After branches at lines 1, 6 have been evaluated differently, all branches before line 6 have been evaluated differently together with the branch at line 6. Corresponding runs have been shown in the first segment of the Table 1 (the segments are separated by horizontal lines). Thus, at this point the algorithm concludes that the branch at line 6 might have little bearing with the actual error cause. The algorithm gives up line 6, and evaluates differently the second last branch at line 5 as well as branches before line 5, as shown in the second segment of Table 1. After this, our algorithm considers the third last branch at line 3, and so on.

Incremental Path Generation: So far, we have clarified the order in which the execution runs will be generated in our search for a successful run. In Table 1 we have shown the order of the generated execution runs for a given failing run and branches evaluated differently from the failing run. Indeed, our algorithm generates these execution runs in a incremental fashion for efficiency. Let us consider the first two execution runs tried out in Table 1. They are

Execution run	*branches evaluated differently*
$\pi_1 = \langle 1, 3, 5, 6, 9, 10 \rangle$	$\langle 6 \rangle$
$\pi_2 = \langle 1, 3, 4, 5, 6, 9, 10 \rangle$	$\langle 3, 6 \rangle$

Recall that the failing run is $\pi = \langle 1, 3, 5, 6, 7, 10 \rangle$ and the buggy program is shown in Figure 2. The run π_2 shares a common suffix with run π_1 (the subpath $\langle 5, 6, 9, 10 \rangle$); the runs also share a common prefix (the subpath $\langle 1, 3 \rangle$). Run π_2 can be obtained by evaluating the branch at line 3 differently over and above π_1. Thus, run π_1 is constructed by modifying failing run π at line 6 (the last branch occurrence of π). Run π_2 is then constructed by incrementally modifying run π_1 in the branch at line 3, that is, we do not construct run π_2 from scratch by modifying the failing run π at lines 3 *and* 6. This incremental path construction is crucial for constructing our bug report efficiently.

Complications due to Nested Branch Statements: Note that when a branch in the failing run is evaluated differently, several execution runs may be obtained due to nested branch statements. For example, if the failing run is $\langle 1, 2, 3, 4, 5, 10 \rangle$ for the program in Figure 2, our algorithm will first try to evaluate branch 5 differently since it is the last branch in the failing run. However, this produces two execution runs $\langle 1, 2, 3, 4, 5, 6, 7, 10 \rangle$ and $\langle 1, 2, 3, 4, 5, 6, 9, 10 \rangle$ due to the nested branch statement at line 6. Our algorithm will check whether *either* of these two runs is feasible and successful

```
Global Variable:  sop, the program's initial statement instance
                  eop, the program's statement instance after which
                       the erroneous state is observable
                  $\pi_f$, the program's failing run to debug

generatePaths (paths, last, diff)
Input: paths, a set of execution runs
       last, a branch instance
       diff, branch instances which have been evaluated differently
Output: a feasible successful run, or Null

1.  br= branch instance just prior to last in $\pi_f$;
2.  while (br is defined)
3.      if ( no branch instance in diff is dynamically
              control dependent on br)
4.          newpaths= {}; /* empty set */
5.          for each $\pi$ in paths do
6.              $de = pde(br, \pi)$;
7.              $subpaths = get\_all(br, de, \pi)$;
8.              $\pi_1$ = sub-path of $\pi$ from sop to br;
9.              $\pi_2$ = sub-path of $\pi$ from de to eop;
10.             for each $\pi'$ in subpaths do
11.                 if ($\pi'$ o $\pi_2$ is infeasible)
12.                     continue;
13.                 $\pi_w = \pi_1$ o $\pi'$ o $\pi_2$;
14.                 if ($\pi_w$ is feasible and successful)
15.                     return $\pi_w$;
16.                 else
17.                     insert $\pi_w$ into newpaths;
18.         if (newpaths is not empty set)
19.             $diff' = \langle br \rangle \circ diff$;
20.             $\pi_r$ = generatePaths(newpaths, br,
                                         $diff'$);
21.             if ($\pi_r$ != Null)
22.                 return $\pi_r$;
23.     else
24.         for each $\pi$ in paths do
25.             $\pi_3$ = sub-path of $\pi$ from br to eop;
26.             if ($\pi_3$ is infeasible)
27.                 remove $\pi$ from paths;
28.         if (paths is empty set)
29.             return Null;
30.     br= branch instance just prior to br in $\pi_f$;
31. return Null;
```

Figure 3: Algorithm to generate a successful run from the failing run

before proceeding to construct any other runs, since these two runs will lead to the same bug reports. We now explain our path generation algorithm in details.

Algorithm Description: Our path generation algorithm is presented in Figure 3. Some of the variables used in the algorithm are pictorially explained in Figure 4. The algorithm proceeds by employing a recursive procedure `generatePaths`. This procedure is invoked at the top level with the parameters $\{\pi_f\}$, e_{last} and $\langle\rangle$, where π_f refers to the failing run, e_{last} refers to the last statement instance in the π_f, and $\langle\rangle$ stands for the empty sequence.

As shown in Figure 3, the three parameters of `generatePaths` are *paths*, *last* and *diff*. The `generatePaths` procedure constructs new execution runs from the runs captured in $paths$ by evaluating branch instances before the instance $last$ differently. All runs in $paths$ have been constructed by evaluating differently branch instances in $diff$ w.r.t. the failing run π_f. Let us re-visit the example in Table 1 which shows the order of path generation for the program in Figure 2 corresponding to the failing run $\pi_f = \langle 1, 3, 5, 6, 7, 10 \rangle$. The left column shows the $diff$ for all invocations of `generatePaths` except the first (where $diff$ is

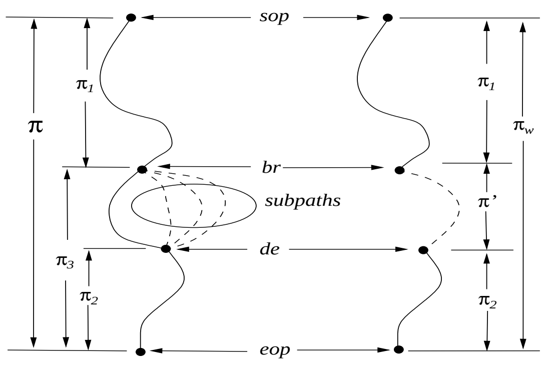

Figure 4: Explanation of algorithm in Figure 3. Solid line refers to subpath of the run π, and broken line refers to the subpaths constructed by evaluating the branch br differently from π.

the empty sequence). The right column shows the value of $paths$ for each invocation of `generatePaths` except the first (where $paths$ only contains the failing run). In this example, for every invocation of `generatePaths`, $paths$ contains a single run.

The `while` loop in the `generatePaths` procedure iteratively retrieves a branch instance prior to the instance $last$ in failing run π_f and assigns it to br (at line 30 of the algorithm). If there are no more branch instances, br is undefined, and `generatePaths` returns Null (*i.e.* we cannot find a successful run). Each loop iteration of `generatePaths` tries to evaluate branch br differently along with other branch occurrences prior to br in failing run π_f.

In each iteration of the `while` loop of `generatePaths`, we first check whether any branch instance in $diff$ is dynamically control dependent on br. If it is so, the branches in br as well as the branches in $diff$ cannot all be evaluated differently from the failing run π_f. To illustrate this point, let us look at the path generation example presented in Table 1; this table shows the order of path generation for the program in Figure 2 corresponding to the failing run $\pi_f = \langle 1, 3, 5, 6, 7, 10\rangle$. Lines 5 and 6 of Figure 2 cannot be evaluated differently together w.r.t. the failing run.

If no branch instance in $diff$ is dynamically control dependent on br, the algorithm generates new runs by evaluating differently the br instance over and above the branches captured by $diff$. Thus, $diff$ is updated to $diff'$ by adding br to $diff$. Recall that the path-set $paths$ captures the set of paths obtained by evaluating branches in $diff$ differently w.r.t. failing run π_f. Thus, to find the set of paths obtained by evaluating branches in $diff'$ differently w.r.t. failing run π_f, we exploit the relationship $diff' = \langle br\rangle \circ diff$ to simply evaluate br differently for all runs in $paths$. The resultant set of paths is captured in $newpaths$. Thus, our algorithm constructs $newpaths$ by incrementally modifying $paths$ instead of directly constructing it from the failing run π_f.

We now explain the functions used in the `generatePaths` procedure (lines 6,7 of Figure 3). The function $pde(br, \pi)$ called at line 6 returns de, the first statement instance which is not (transitively) dynamically control dependent on br in the execution run π. The function $get\_all(br, de, \pi)$ called at line 7 of Figure 3 retrieves all acyclic paths where: (1) each acyclic path starts from $loc(br)$ (the control location of the branch instance br) and ends at $loc(de)$ (the control location of the statement instance de), (2) br is evaluated differently from π in each acyclic path. We choose to consider acyclic paths to avoid enumerating too many paths. However, this may cause us to miss a feasible successful run since all possible program paths are not constructed by our algorithm.

Subject Pgm.	*Description*	*# Buggy versions*
schedule	priority scheduler	9
schedule2	priority scheduler	10
replace	pattern replacement	32
print_tokens	lexical analyzer	7
print_tokens2	lexical analyzer	10
tcas	altitude separation	41

Table 2: Description of the Siemens suite

Our path generation algorithm requires checking whether an execution run is feasible and successful (line 14 of Figure 3). We have used the automated theorem prover Simplify [4] to check for feasibility. This feasibility check returns the possible inputs under which the execution run is executed. We then check whether the execution run is successful (*i.e.* absence of the fault being localized) by checking the execution run for *any one* of these feasible inputs.[2] *Checking whether the execution run for a specific input is successful, however, requires user intervention*; otherwise the programmer has to precisely characterize properties of successful runs, possibly as assertions.

4. EXPERIMENTS

In order to validate our method experimentally, we developed a prototype implementation of our path generation algorithm. We employed the prototype on the Siemens test suite [7] and used the evaluation framework of [11] to quantitatively measure the quality of generated bug reports. We have chosen the Siemens test suite and the evaluation framework of [11] because previous approaches [3, 11] have also conducted experiments with the same subject programs and evaluation framework. Thus, we can compare our approach with these works. In this section, we first introduce the subject programs (Section 4.1) and the evaluation framework (Section 4.2). Section 4.3 elaborates results from our experiments, and compares them with results reported for existing fault localization approaches [3, 11]. We also report time overheads of our approach.

4.1 Subject programs

The subject programs used in our experiments are 109 buggy C programs from the Siemens test suite [7], as modified by Rothermel and Harrold [13]. We excluded the floating point calculation program `tot_info` in the Siemens suite from our experiments. Our prototype implementation uses the Simplify theorem-prover [4] to check the feasibility of an execution run, and Simplify does not work well with floating-point variables.

Each of the 109 buggy programs has been created from one of six programs by manually injecting one defect. The six programs range in size from 170 to 560 lines, including comments. Table 2 shows descriptions of these programs. The third column in Table 2 shows the number of buggy programs created from each of the six programs. The injected defects include code omissions, relaxing or tightening conditions of branch statements, superfluous code and wrong values for assignment statements; some defects span multiple lines or even functions. Although the benchmarks are simple, and cannot reflect all possible errors in real life, they help us gain valuable experience in debugging.

4.2 Evaluation framework

Renieris and Reiss have proposed an evaluation framework to evaluate the quality of a defect localizer [11]. Every error report

[2]Clearly, for the same execution run, some inputs may lead to successful executions, while others lead to failing executions.

is assigned a score to show the quality of this report. The score indicates the amount of code that an programmer can ignore for debugging, assuming that the programmer can find the error when he reads the erroneous statements, and he performs a breadth-first search for defect localization starting from statements in the error report. Details about the score computation can be found in [11].

4.3 Experimental results

Recently, Renieris and Reiss have proposed the nearest neighbor query method ("NN/Perm") for fault localization in [11]. The NN/Perm method compares code coverage between a failing run and its nearest successful run. Also, Cleve and Zeller have proposed the methods "CT/relevance" and "CT/infected" [3]. Both CT/relevance and CT/infected methods analyze cause transitions in the failing run to generate a bug report. They differ in usage of the bug report for fault localization – CT/relevance uses the knowledge of control/data dependencies, while CT/infected uses the knowledge of which program states are infected. We compare our method with NN/Perm, CT/relevance and CT/infected. We employed the prototype implementation of our method to 107 buggy programs from the Siemens suite. Two out of the 109 programs had to be ruled out because these two programs are syntactically different from, but semantically the same as the original "correct" program.

Quality of Bug Report: Table 3 shows the distribution of scores for four methods. The data for NN/Perm is taken from [11], and the data for CT/relevance and CT/infected are taken from [3]. Our method is shown as APG, an abbreviation for Automated Path Generation. From this table, we can see that our method is comparable with the cause transition methods, and a little better than the nearest neighbor method. Bug reports returned by APG, CT/relevance and CT/infected methods all achieved a score of 80% or better for more than 41% of all the buggy programs, while the NN method achieved at least 80% score for about 26% of the programs. Note that our method achieved these scores with automatic generation of successful run, while the NN, CT/relevance and CT/infected methods all required the programmer to provide/choose a successful run. In other words, by automating the successful run construction, we have not compromised on the bug report quality.

Score	NN/Perm	CT/relevance	CT/infected	APG
100%	0.00	5.43	4.55	0.93
90-99%	16.51	30.23	26.36	34.58
80-89%	9.17	6.20	10.91	8.41
70-79%	11.93	6.20	13.64	8.41
60-69%	13.76	9.30	4.55	5.61
50-59%	19.27	10.08	6.36	4.67
40-49%	3.67	3.88	1.82	7.48
30-39%	6.42	10.08	3.64	4.67
20-29%	1.83	3.10	7.27	9.35
10-19%	0.00	10.85	0.00	1.87
0-9%	17.43	4.65	20.91	14.02

Table 3: Distribution of scores for four methods.

Size of Bug Report: Apart from using the scores to describe the quality of bug reports,the size of a bug report (*i.e.* the number of statements in a bug report) can be of practical importance. If a bug report contains too many statements, the programmer can be overwhelmed with the bug report. In the experiments, we observed that 71% of our 107 bug reports contained at most seven branch statements. This indicates that the bug reports produced by our method are not voluminous and overwhelming.

Time Overheads: In the experiments, our method found the successful run within 10 minutes for 75% of all 107 buggy programs, and for 83% of 47 buggy programs which had high quality bug reports (*i.e.*, score of 80% or above). Most of the time overheads for our method is due to the feasibility check by the external theorem prover Simplify. The feasibility check enables the check to find whether a run is successful (since we cannot even observe the behavior of infeasible runs). Still the overall time overheads are tolerable for most programs in the Siemens suite.

5. DISCUSSION

In this paper, we have investigated the problem of software fault localization, that is, localizing the error cause(s) from an observable program error (as seen in a failing program run). We automatically generate a successful execution close to the failing execution, and then compare the two execution runs to discover the likely defects in the buggy program. Through this comparison, we highlight the sequence of branches in the failing run which are evaluated differently in the successful run. Our approach does not require the user to provide successful executions for debugging as in previous approaches. Using the Siemens benchmark suite, we have conducted experiments to evaluate the quality/volume of our bug reports as well as the time required to construct the bug reports.

Acknowledgments

This work was partially supported by a Public Sector research grant from A*STAR, Singapore.

6. REFERENCES

[1] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In *ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 97–105, 2003.

[2] S. Chaki, A. Groce, and O. Strichman. Explaining abstract counterexamples. In *ACM Symposium on the Foundations of Software Engineering (FSE)*, 2004.

[3] H. Cleve and A. Zeller. Locating causes of program failures. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2005.

[4] D. Detlefs, G. Nelson, and J. Saxe. Simplify: A theorem prover for program checking. Technical report, HP Labs, Palo Alto, CA, 2003. `http://research.compaq.com/SRC/esc/Simplify.html`.

[5] A. Groce. Error explanation with distance metrics. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 108–122, 2004.

[6] A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *SPIN Workshop on Model Checking of Software*, pages 121–135, 2003.

[7] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, 1994.

[8] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 467–477, 2002.

[9] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2003.

[10] B. Pytlik, M. Renieris, S. Krishnamurthi, and S. P. Reiss. Automated fault localization using potential invariants. *CoRR*, cs.SE/0310040, Oct, 2003.

[11] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *Automated Software Engineering (ASE)*, pages 30–39, 2003.

[12] T. W. Reps, T. Ball, M. Das, and J. R. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *ACM Symposium on the Foundations of Software Engineering (FSE)*, 1997.

[13] G. Rothermel and M. J. Harrold. Empirical studies of a safe regression test selection technique. *IEEE Transactions on Software Engineering*, 24, 1998.

[14] J. Ruthruff, E. Creswick, M. Burnett, C. Cook, S. Prabhakararao, M. F. II, and M. Main. End-user software visualizations for fault localization. In *ACM Symposium on Software Visualization*, pages 123–132, 2003.

[15] A. Zeller. Isolating cause-effect chains from computer programs. In *ACM Symposium on the Foundations of Software Engineering (FSE)*, 2002.

[16] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28, 2002.