THE NATIONAL UNIVERSITY
of SINGAPORE

School of Computing
Lower Kent Ridge Road, Singapore 119260

TRC6/05

Automated Path Generation for Software Fault Localization

Tao WANG and Abhik ROYCHOUDHURY

June 2005

Technical Report

Foreword

This technical report contains a research paper, development or tutorial article, which has been submitted for publication in a journal or for consideration by the commissioning organization. The report represents the ideas of its author, and should not be taken as the official views of the School or the University. Any discussion of the content of the report should be sent to the author, at the address shown on the cover.

JAFFAR, Joxan
Dean of School

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 which is close to the failing run in terms of a distance metric capturing control flow. The distance metric takes into account the sequence of statements executed in the two runs, and not just the set of statements executed. We use the distance metric to locate "similar" branch instances which appear in the failing and successful run with different outcomes. The program statements for such 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.

Keywords

Program Understanding, Automated Debugging, Fault Localization

1. INTRODUCTION

Program debugging is an age-old software engineering activity. Many programmers still use a source-level debugger, which originated in the 1970s, 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.

An important challenge in automated debugging is in characterizing the desired correct behaviors. One approach for describing correct program behaviors is to use logical assertions at specific control locations of the program, often given as pre- and post-conditions of program methods. Thus, a program is correct if *all* its execution traces satisfy each assertion whenever the corresponding control location is visited. Given an execution trace showing an observable error, one can check for violation of some of the user-provided assertions to localize the error. However, developing these assertions is usually a difficult task.

Another approach to fault localization, which has been explored in the recent times [5, 9, 12, 15, 16, 17, 24], 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 use 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.

In this paper, we automate the process of constructing one successful run from a program's failing run. We construct a successful run from the failing run by changing the outcomes of some of the conditional branch instances in the failing run. 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 are submitted to the programmer as bug report. *In this paper, whenever we refer to evaluating branch statement instances differently, we mean changing the outcome of the branch condition evaluation.*

Consider the program fragment in Figure 1 where a `0` value is printed in line 10 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)`.

In the preceding, we have illustrated a simple example where 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.

In general, given a program P and a failing execution run π, there exist many runs of P which do not exhibit the error observed

```
v=0;
...
if (x>0) {
u=5;
...
}
else {
u=v;
}
printf(``%d'',u);
```

Figure 1: A simple example.

in π. Certain works [16] have shown empirically that successful executions which are "similar" to the failing execution run can be more useful for fault localization. In this paper, we systematically generate a successful run from the failing run with the help of a distance metric capturing control flow. Given a failing run π of program P, our method automatically generates a successful run π' of P whose control flow is close to π as per our distance metric. Data flow is not considered since we want to generate the successful run efficiently. Our distance metric is sensitive to the *sequence* of statements executed in the two runs, not just the set of statements executed. Given a failing run π and a successful run π' which execute the same set of statements but possibly in different order, our distance metric captures the difference in relative order of statements by reporting sequence of branch instances which are evaluated differently in π, π'. The statements for these branch instances are returned as bug report.

How useful is our path generation method and the resultant bug report? To evaluate our technique, we have employed it to localize bugs of the Siemens benchmark suite [19], 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 [5, 16] which requires the programmer to manually provide/choose a successful run.

Summary of Contributions. 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 take the view that the *distance* between two runs can be summarized by the sequence of comparable branch statement instances which are evaluated differently in the two runs. Based on this view, we automatically generate a feasible successful run[1] which is close to, that is, has *little distance* with, 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 size of our bug report and the time overheads of our fault localization method.

Section Organization. The rest of this paper is organized as follows. The next section introduces related work on fault localization. Our distance metric and path generation algorithm are presented in Sections 3, 4. Sections 5, 6 discuss our experimental setup and experimental results. Section 7 concludes the paper.

2. RELATED WORK

In this section, we discuss work on localizing software errors.

Using assertions for program debugging/testing has long been studied. Assertion processing has been integrated into programming languages and environments [14, 18]. There is a rich body of work on this topic and we do not discuss it here since it is orthogonal to our approach. There has also been intense research on the topic of input/test case generation based on various coverage criteria. The aim of these methods is to expose more program behaviors for the purpose of testing. This is somewhat different from our goal since we generate a program input/execution-run for localizing the error cause in a *specific* failing run.

Recently, there has been a lot of interest in program error localization by comparing successful and failing runs of the buggy program [3, 5, 9, 12, 15, 16, 17, 24]. These techniques often differ in which characteristic of execution runs is used for comparison. The characteristic can be acyclic paths [17], potential invariants [15], executed statements [3, 12], transitions [9], basic block profiles [16] or program states [5, 24]. Jones et al. [12] and Ruthruff et al. [20] 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 [13] discovers abnormal return value of methods from many runs. Unlike our method, these works require that both failing and successful runs are available before fault localization.

The work of Renieris and Reiss [16] is related to ours. They have demonstrated through empirical evidence that the successful run which is "closest" to the failing run can be more helpful for fault localization than a randomly selected successful run. However, [16] does not automatically *generate* the closest successful run from a failing run. Instead, they choose the successful run which is closest to the failing run from a large number of available successful runs. Thus, even if a large set of representative program inputs is available, one has to first classify a subset of these inputs as successful for any *given* faulty run.

Moreover, [16] compares two runs (and measures how close they are) by comparing the *set* of basic blocks[2] executed in each run. Thus, they cannot distinguish between runs which execute the same statements but in different order. We consider the *sequence* of statements executed in each run for determining proximity of two runs. Clearly, even if for a faulty run, the programmer has a number of successful runs at his disposal (*i.e.* automated generation of one successful run is unnecessary), our sequence based distance metric can be used for accurately comparing the control flow of two runs. Our distance metric bears some similarities to the notion of proximity between execution runs in [5], where [5] compares program states with similar contexts for fault localization. Detailed experimental results comparing our method with [5] and [16] appear in Section 6.

Software fault localization via model checking has also been studied [4, 8]. 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 [25]. The reduced input usually corresponds to a shorter execution,

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

[2] Actually a sorted sequence of the basic blocks based on execution counts is used; this is of course different from the execution sequence of the basic blocks in the failing run.

which may be easier to debug. Simplifying/reducing the erroneous input can also lead to a successful program input. 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. For programs with integer inputs this approach may be problematic *e.g.* consider the situation where the failing input of a program is `i=2` and the only successful input is `i=3`.

Program slicing [23], especially dynamic slicing [1], can facilitate debugging by ruling out statements which do not affect the observable error via control and/or data flow, because such statements are likely to be irrelevant to the cause of the error. However, the dynamic slice is not always useful for accurate fault localization *e.g.* consider an assignment `x=0` which is wrongly written as `y=0` thereby leading to an erroneous value of `x` later on.

In the past, various interactive approaches for error localization have also been studied (*e.g.* [21]). We do not discuss them here, since our focus is on automated debugging.

3. PROXIMITY BETWEEN RUNS

Our approach for generating a feasible successful run is based on a notion of "proximity" between runs. Given a failing run π of program P, our approach attempts to find a feasible successful run of P which is close to π based on a *distance metric*. The intuition is that, if we can find π', the feasible successful run closest to π, then we can localize the error in π by comparing π and π'. We elaborate the distance metric used for comparing execution runs in this section. In the next section, we present a path generation algorithm which takes in a failing run and produces a "similar" successful run (*i.e.*, similar based on our distance metric).

We consider each execution run of a program to be a sequence of *events* $\langle e_0, e_1, ..., e_{n-1}\rangle$ where e_i refers to the ith event during execution. *Each event e_i represents a control location or a line number in the program*; the program statement corresponding to this line number is denoted as $stmt(e_i)$. To distinguish events from different execution runs, we denote the ith event in an execution run π as e_i^π, that is, the execution run appears as a superscript. We will drop the superscript when it is obvious from the context.

Our distance metric measures the difference between two execution runs π and π' of a program, by comparing behaviors of "corresponding" branch statement instances from π and π'. The branch statement instances with differing outcomes in π, π' are captured in $dist(\pi, \pi')$ – the distance between execution run π and execution run π'. How do we find out the "corresponding" branch statement instances? For this purpose, we have defined a notion of *alignment* to relate statement instances of two execution runs. Our alignment is based on *dynamic control dependence*.

DEFINITION 1 (DYNAMIC CONTROL DEPENDENCE). *Given an execution run π of a program, an event e_i^π is dynamically control dependent on another event e_j^π if e_j^π is the last event before e_i^π in π where $stmt(e_i^\pi)$ is statically control dependent*[3] *on $stmt(e_j^\pi)$.*

Note that any method entry event is dynamically control dependent on the corresponding method invocation event. According to the above definition, any event e_i^π in an execution run π (except the first one which denotes entry to the main method) is dynamically control dependent on exactly one event. We use the notation $dep(e_i^\pi, \pi)$ to denote the event on which e_i^π is dynamically control dependent in the execution run π. We are now ready to present our definition of event alignment.

DEFINITION 2 (ALIGNMENT). *We define alignment between two execution runs π, π' as a binary relation*

$$align_{\pi,\pi'} \subseteq Events_\pi \times Events_{\pi'}$$

where $Events_\pi$ ($Events_{\pi'}$) denotes the set of events appearing in the execution run π (π'). In particular,

$$\forall e \in Events_\pi\ \forall e' \in Events_{\pi'}\ \ (e, e') \in align_{\pi,\pi'} \text{ iff}$$

1. *$stmt(e)$ = $stmt(e')$, and*
2. *either e, e' are the first events appearing in π, π' or $(dep(e,\pi), dep(e',\pi')) \in align_{\pi,\pi'}$.*

Thus, when a branch event e_i^π cannot be aligned with any event from the execution π', this should only affect alignments of those events in π which belong to the chain of events dynamically control dependent on e_i^π. Note that the beginning of the ith iteration of a loop in the execution π is aligned with the beginning of the ith iteration of the same loop in the execution π', in order to properly compare events from different loop iterations.

According to the notion of alignment presented in Definition 2, for any event e in π there exists *at most* one event e' in π' such that $(e, e') \in align_{\pi,\pi'}$. The distance between π and π' (denoted $dist(\pi, \pi')$) captures all branch event occurrences in π which (i) can be aligned to an event in π' and (ii) have different outcomes in π and π'. Formally, the distance metric between two execution runs can be defined as follows.

DEFINITION 3 (DISTANCE METRIC). *Consider two execution runs π, π' of a program. The distance between π, π', denoted $dist(\pi, \pi')$ is a subsequence of the event sequence represented by π. We define*

$$dist(\pi, \pi') = \langle e_{i_1}^\pi, \ldots, e_{i_k}^\pi\rangle$$

such that

1. *each e in $dist(\pi, \pi')$ is a branch event occurrence drawn from execution run π.*
2. *for each e in $dist(\pi, \pi')$, there exists another branch occurrence e' in execution run π' such that $(e, e') \in align_{\pi,\pi'}$ (*i.e. *e and e' can be aligned). Furthermore, the outcome of e in π is different from the outcome of e' in π' (since e, e' can be aligned, they denote occurrences of the same branch statement).*
3. *all events in π satisfying criteria (1) and (2) are included in $dist(\pi, \pi')$.*
4. *the events in $dist(\pi, \pi')$ appear in the same order as in π, that is, for all $1 \leq j < k$, $i_j < i_{j+1}$ (event $e_{i_j}^\pi$ appears before event $e_{i_{j+1}}^\pi$ in π).*

As a special case, if execution runs π and π' have the same control flow, then we define $dist(\pi, \pi') = \langle e_0^\pi\rangle$.

Clearly we can see that in general $dist(\pi, \pi') \neq dist(\pi', \pi)$. The reason for making a special case for π and π' having the same control flow will be explained later in the section when we discuss comparison of distances.

As a simple illustration of our distance metric, consider the program in Figure 2 and the following runs π, π' in the program

[3] A statement $stmt$ is statically control dependent on a statement $stmt'$ if $stmt'$ evaluates a conditional branch which can control whether $stmt$ will subsequently be executed [7].

```
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.

$$\begin{aligned} \pi &= \langle 1, 3, \quad 5, 6, 7, 10 \rangle \\ \pi' &= \langle 1, 3, 4, 5, 6, 9, 10 \rangle \end{aligned}$$

The distance metric between these two runs is $dist(\pi, \pi') = \langle 3, 6 \rangle$. This is because branches $3, 6$ are aligned in runs π and π' and their outcomes are different in π, π'. If the branches at lines $3, 6$ are evaluated differently we get π' from π.

Why do we define the distance metric $dist(\pi, \pi')$ to contain branch event occurrences of π which evaluate differently in π' ? Recall that we want to generate a successful run for purposes of fault localization. If π is the failing run and π' is a successful run, then $dist(\pi, \pi')$ tells us which branches in the failing run π need to be evaluated differently to produce the successful run π'. Clearly, if we have a choice of successful runs we would like to make minimal changes to the failing run to produce a successful run. Thus, given a failing run π and two successful runs π', π'' we would choose π' over π'' if $dist(\pi, \pi') < dist(\pi, \pi'')$. This requires us to *compare* distances. How we do so is elaborated in the following.

DEFINITION 4 (COMPARISON OF DISTANCES). *Let π, π', π'' be three execution runs of a program. Let*

$$dist(\pi, \pi') = \langle e^{\pi}_{i_1}, e^{\pi}_{i_2}, \ldots, e^{\pi}_{i_n} \rangle \text{ and}$$

$$dist(\pi, \pi'') = \langle e^{\pi}_{j_1}, e^{\pi}_{j_2}, \ldots, e^{\pi}_{j_m} \rangle$$

We define that $dist(\pi, \pi') < dist(\pi, \pi'')$ iff there exists a non-negative integer K such that

1. *$K \le m$ and $K \le n$*

2. *the last K events in $dist(\pi, \pi')$ and $dist(\pi, \pi'')$ are the same, that is, $0 \le x < K \;\; i_{n-x} = j_{m-x}$.*

3. *one of the following two conditions hold*

   - *either $K = n < m$, that is, $dist(\pi, \pi')$ is a suffix of $dist(\pi, \pi'')$,*

   - *or the ($K + 1$)th event from the end in $dist(\pi, \pi')$ appears later in π as compared to the ($K + 1$)th event from the end in $dist(\pi, \pi'')$, that is, $i_{n-K} > j_{m-K}$.*

It should be noted that given a failing run π and two successful runs π', π'' we say that $dist(\pi, \pi') < dist(\pi, \pi'')$ based on a careful combination of the following criteria.

- The branches of π that need to be evaluated differently to get π' appear closer to the end of π (where the error is observed), as compared to the branches of π that need to be evaluated differently to get π'' . This is reflected in the condition $i_{n-K} > j_{m-K}$ of Definition 4.

π	π'	π''	$dist(\pi, \pi')$	$dist(\pi, \pi'')$
1	1	1		⋆
		2		
3	3	3	∘	
	4			
5	5	5		
6	6	6	∘	⋆
7				
	9	9		
10	10	10		

Table 1: Comparison of distance metrics. The first three columns show the event sequences of execution runs π, π' and π''. The last two columns show distance metrics.

- Fewer branches of π need to be evaluated differently to get π' as compared to the number of branches of π that need to be evaluated differently to get π''. This is reflected in the condition $K = n > m$ of Definition 4.

To illustrate our comparison of distances, take the program segment in Figure 2 as an example. Consider the following execution runs π, π', π''.

$$\begin{aligned} \pi &= \langle 1, 3, 5, 6, 7, 10 \rangle \\ \pi' &= \langle 1, 3, 4, 5, 6, 9, 10 \rangle \\ \pi'' &= \langle 1, 2, 3, 5, 6, 9, 10 \rangle \end{aligned}$$

In Table 1, the first three columns show the event sequences (in terms of line number) of execution runs π, π' and π'', respectively. Events along the same horizontal line are aligned. Branches $3, 6$ behave differently in π and π', so $dist(\pi, \pi') = \langle 3, 6 \rangle$, as indicated by the "∘" in Table 1. Similarly, $dist(\pi, \pi'') = \langle 1, 6 \rangle$, as indicated by the "⋆" in Table 1. Comparing $\langle 3, 6 \rangle$ with $\langle 1, 6 \rangle$ we see that $\langle 3, 6 \rangle < \langle 1, 6 \rangle$ since line 3 occurs after line 1 in execution run π. It should be noted that in this simple example all the execution runs have only a single occurrence of a program statement; this is for simplicity of illustration. Of course, if there are multiple occurrences we will distinguish these occurrences through our notion of alignment.

Comparing runs with identical control flow. Using Definitions 3 and 4 we can see that if π is the failing run, π_{same} is a successful run with same control flow as that of π (*i.e.* same sequence of statements executed by a different input) and π_{diff} is a successful run with control flow different from π we will have $dist(\pi, \pi_{diff}) < dist(\pi, \pi_{same})$. As a result, our path generation algorithm which tries to find the successful run with minimal distance to the failing run will avoid successful runs with same control flow as that of the failing run. Indeed this choice is deliberate, since we want to find a successful run with minimal difference in control flow from the failing run, but not with zero difference. This is because we construct our bug report by comparing the control-flow (not data flow) of the generated successful run with the failing run. If the two runs have the same control flow, the bug report is null and hence useless to the programmer.

4. 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

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

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.

Our path generation algorithm tries to generate the closest successful run from the failing run π_f, according to the distance metric in Definition 3. In other words, given a failing run π_f, we seek to generate a successful run π' such that there does not exist any other successful run π'' with $dist(\pi_f, \pi'') < dist(\pi_f, \pi')$ (see Definition 4). The distance between π_f and π' consists of the branches in the failing run π_f which need to be evaluated differently to get π'. These branches are presented to the programmer as "bug report" from which the programmer will try to isolate the bug.

How do we construct the closest successful run from the failing run ? Of course, we will not generate all the possible runs and then find out the closest. Instead, we seek to generate the "closest" program run from the failing run by exploiting our understanding of the distance metric. If this turns out to be a feasible successful run then our search stops; otherwise we try for the next closest run and so on. Our notion of proximity in Definition 4 ensures that a run π' is close to a failing run π_f (*i.e.*, distance between π_f, π' is small) if the branches of π_f which need to be evaluated differently are near the end of π_f (where the error is observed). Furthermore, the number of branches of π_f that need to be evaluated differently should be small. Thus, given a failing run π_f, we will first try to evaluate differently the last branch occurrence (call it b_{last}) in π_f to construct a run π_1. Among all the branch occurrences in π_f, clearly b_{last} is nearest to the end of π_f. If π_1 is a successful and feasible run, we return π_1 as the closest successful run. Otherwise we successively construct other runs by evaluating b_{last} as well as other branch occurrences of π_f 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_f differently and carry out the above steps again. This process goes on until a feasible successful run is obtained.

Example. Let us take the program segment in Figure 2 as an example. Assume that the failing run $\pi_f = \langle 1,3,5,6,7,10\rangle$. The branch occurrences appearing in this run are at lines $1,3,5,6$. Note that the execution run π_f does not contain multiple occurrences of any program statement; so we do not need to worry about distinguishing between 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_f, thereby constructing new execution runs.

Table 2 shows the order in which the branches of failing run π_f 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. According to the distance metric, we should now choose the branch which is the closest to line 6. However, the algorithm cannot choose line 5 at this time, although line 5 is the closest. This is because 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 2 (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 2. 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 the closest successful run. In Table 2 we have shown the order of the generated execution runs for a given failing run and the distances of these runs from the failing run. However, our algorithm will *not* generate the distances and then find out the execution run(s) for each distance. This would be inefficient since all execution runs will have to be generated from scratch by modifying the failing run π_f. Let us consider the first two execution runs tried out in Table 2. They are

$$\begin{array}{ll} \pi_1 = \langle 1,3,5,6,9,10\rangle & dist(\pi_f,\pi_1) = \langle 6\rangle \\ \pi_2 = \langle 1,3,4,5,6,9,10\rangle & dist(\pi_f,\pi_2) = \langle 3,6\rangle \end{array}$$

Recall that the failing run is $\pi_f = \langle 1,3,5,6,7,10\rangle$ 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. Indeed, our algorithm generates the execution runs in this incremental fashion. Thus, run π_1 is constructed by modifying failing run π_f at line 6 (the last branch occurrence of π_f). 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_f 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 state-

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

generatePaths ($paths$, $last$, $distance$)
Input: $paths$, a set of execution runs
    $last$, a branch event
    $distance$, distance between $\pi_f$ and runs in $paths$
Output: a feasible successful run, or Null

begin
1.   $br$= branch event just prior to $last$ in $\pi_f$;
2.   while ($br$ is defined)
3.       if ( no event in $distance$ 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.              $distance'$= $\langle br \rangle \circ distance$;
20.              $\pi_r$ = generatePaths($newpaths$, $br$,
                                          $distance'$);
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 event just prior to $br$ in $\pi_f$;
31.  return Null;
end
```

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

ments. For example, for the program in Figure 2 if the failing run is $\langle 1, 2, 3, 4, 5, 10 \rangle$, 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 *any* of these two runs is feasible and successful before proceeding to construct any other execution runs. This is part of our attempt to generate the closest successful run according to the distance metric of Definition 3. We now explain our path generation algorithm in details.

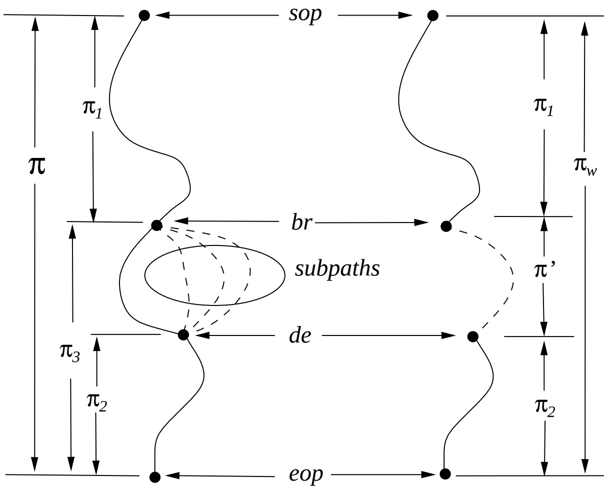

Figure 4: Explanation of algorithm in Figure 3.

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 event 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 *distance*. The `generatePaths` procedure constructs new execution runs from the runs captured in $paths$ by evaluating branch events before the event $last$ differently. All runs in $paths$ have been constructed by evaluating differently events in the distance metric $distance$ w.r.t. the failing run π_f. These runs have the same distance w.r.t. the failing run. Let us re-visit the example in Table 2 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 $distance$ for all invocations of `generatePaths` procedure except the first (where $distance$ is 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 event prior to the event $last$ in failing run π_f and assigns it to *br* (at line 30 of the algorithm). If there are no more branch events, *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 event in $distance$ is dynamically control dependent on br. If it is so, the branches in br as well as the branches in $distance$ 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 2; 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 event in $distance$ is dynamically control dependent on *br*, the algorithm generates new runs by evaluating differently the *br* event over and above the branches captured by $distance$. Thus, $distance$ is updated to $distance'$ by adding br to $distance$. Recall that the path-set $paths$ captures the set of paths obtained by evaluating branches in $distance$ differently w.r.t. failing run π_f. Thus, to find the set of paths obtained by evaluating branches in $distance'$ differently w.r.t. failing run π_f, we exploit the relationship $distance' = \langle br \rangle \circ distance$ 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 event 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

- each acyclic path starts from $loc(br)$ (the control location of the branch event br) and ends at $loc(de)$ (the control location of the event de), and
- 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 the closest successful run since all possible program paths are not considered.

In order to improve the performance of our algorithm, we have exploited the following property: if a path is infeasible, all extensions of the path are also infeasible. In particular, line 11 of the algorithm checks the feasibility of a subpath π' o π_2. If it is infeasible, all execution runs with π' o π_2 as suffix are also infeasible, and there is no need to check them. Similarly, when some event in $distance$ is dynamically control dependent on br, line 26 of the algorithm checks the feasibility of a subpath π_3 and prunes any execution runs with π_3 as suffix if π_3 is infeasible. Figure 4 shows the relation between various paths (*e.g.* π', π_2, π_3) used in Figure 3, the algorithm's pseudo-code.

Recall that we want the path generation algorithm in Figure 3 to construct execution runs monotonically w.r.t. the distance metric in Definition 3, so that it can return a successful run which is close to the failing run. This is stated in Theorem 1 whose proof is omitted due to lack of space.

THEOREM 1. *[Proximity of Successful Run] Consider the failing execution run π_f for a program. If a run π' is constructed before another run π'' by the* `generatePaths` *method in Figure 3, then $dist(\pi_f, \pi') < dist(\pi_f, \pi'')$ (as per Definition 4) or $dist(\pi_f, \pi') = dist(\pi_f, \pi'')$.*

The above theorem does not claim that we generate the closest successful run from the failing run. *This reflects the reality, where we can, but choose not to generate the closest successful run for reasons of efficiency.* Our method does not generate certain program paths and one of these can be the closest successful run.

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 [6] 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 possible inputs. Clearly, for the same execution run, some inputs may lead to successful executions, while others lead to failing executions. Our implementation chooses any one of the feasible inputs and checks whether the corresponding execution run is successful.

5. EXPERIMENTAL SETUP

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 [11] and used the evaluation framework of [16] to quantitatively measure the quality of generated bug reports. We have chosen the Siemens test suite and the evaluation framework of [16] because previous approaches [5, 16] 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 5.1) and the evaluation framework (Section 5.2). We then present important details about the prototype implementation of our approach (Section 5.3).

5.1 Subject programs

The subject programs used in our experiments are 109 buggy C programs from the Siemens test suite [11], as modified by Rothermel and Harrold [19]. Each buggy program 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 3 shows descriptions of these programs. The third column in Table 3 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.

We slightly changed two of the subject programs in our experiments. In particular, we rewrote the `schedule` and `schedule2` programs which read a floating point number and round it to an integer value to directly read an integer. We also excluded the floating point calculation program `tot_info` in the Siemens suite from our experiments. This is because our prototype implementation uses the Simplify theorem-prover [6] to check the feasibility of an execution run, and Simplify does not work well with floating-point variables.

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 3: Description of the Siemens suite

5.2 Evaluation framework

Renieris and Reiss have proposed an evaluation framework to evaluate the quality of a defect localizer [16]. Each error report is assigned a score to show the quality of this report. The score indicates the amount of code that an ideal programmer can ignore for debugging. Clearly, higher score indicates better quality bug report. We now discuss the score computation mechanism.

To compute the score of a bug report, [16] requires a correct version of the buggy program, where the defect has been fixed. Erroneous statements refer to the difference between the two programs (*i.e.* the statements which have been fixed in the correct version). The score computation works on the program dependence graph (PDG) [10] for the buggy program. Nodes in a PDG represent statements in the program, and edges represent data or control dependencies between statements. We have used the Codesurfer [2] to construct the PDG. Erroneous statements are marked as "defect" in the PDG; statements included in the bug report are marked as "blamed" in the PDG. Let $DS(n)$ be the set of nodes that can reach or be reached from blamed nodes by traversing at most n directed edges in the PDG. For example, $DS(0)$ is the set of blamed nodes, and $DS(1)$ include blamed nodes and all nodes which have directed edges to or from blamed nodes. We define that DS_* is the $DS(n)$ with the smallest n, which contains at least one erroneous statement. The score is then computed by the formula:

$$score = 1 - \frac{|DS_*|}{|PDG|} \quad (1)$$

As a special case, we define the score to be zero for an empty report, since an empty report is useless to the programmer. This framework assumes that the programmer can find the error when he/she reads the erroneous statements, and he/she performs a breadth-first search for defect localization starting from statements in the error report. Thus, DS_* reflects the amount of code in the program that the programmer has to examine for defect localization using the bug report, and the score indicates the amount of code which can be ignored.

Note that the edges in the PDG can also be considered as undirected; this should increase DS_* and hence decrease the score. When we ran experiments, we found that this change hardly had any effect on the scores for bug reports generated by our approach.

5.3 Implementation

We have built an experimental prototype for evaluation. The prototype is implemented in C and includes two coupled components: an *instrumentation component*, and an *path generation component*. The instrumentation component uses the GNU debugger (GDB) to trace the execution path of the failing run. The path generation component implements our path generation algorithm in Figure 3 and uses an external theorem-prover Simplify to check the feasibility of an execution. We now discuss a few important implementation issues.

Trade-off. In order to improve the efficiency of our prototype, we have made some trade-offs in the implementation. These trade-offs have reduced the time overheads of our path generation algorithm, but they may cause our prototype implementation to miss the successful run which is the closest to the failing run according to the distance metric.

In our initial implementation, we decreased, but did not increase the number of iterations when generating a successful run from a failing run. That is, if a branch event exits a loop, we did not consider evaluating it differently. However, experiments showed that there were still too many paths to construct and check. Therefore we decided to not evaluate *any* loop branch events differently. Thus, the successful execution run returned by our implementation always has exactly the same number of loop iterations as in the failing run. This can dramatically reduce the number of constructed execution runs. Also, when we evaluate a branch event differently, there can be many possible paths because of nested branch statements. Our prototype stops when the number of runs constructed by evaluating differently the same set of branch events has reached a threshold.

Feasibility check. One important problem in the implementation is how to check the feasibility of an execution run or a subpath, as required by the path generation algorithm in Figure 3. Our prototype traverses the execution run from the beginning till the end to generate a constraint ϕ over program input and variables. We then invoked the Simplify theorem-prover [6] to prove the validity of the formula $\neg\phi$. If Simplify succeeds, we infer that the path is infeasible; otherwise we deem the path as feasible.

The use of the Simplify theorem prover requires us to consider the power of its decision procedures for inferencing. Simplify is sound but incomplete, that is, any formula it proves as valid is valid but it may fail to prove the validity of a valid formula. Thus, if ϕ_π is the constraint for a path π, $\neg\phi_\pi$ is valid (i.e. the path is actually infeasible) and Simplify fails to prove the validity of $\neg\phi_\pi$, we will actually treat π as feasible when it is not. However, this unlikely situation did not occur in our experiments with the Siemens suite.

Failing Run. One issue in our experiments is the choice of the failing input (and hence the failing run) for each buggy program. We first chose a failing input which can observe the error for each buggy program from the Siemens test suite. We then used Zeller and Hildebrandt's approach [25] to further simplify the input for producing a failing run. This was particularly useful for the buggy versions of text-processing programs in the Siemens test suite (*e.g.*, `replace`, `print_tokens`).

6. EXPERIMENTAL RESULTS

We employed the prototype implementation of our method to 109 buggy programs from the Siemens suite. Two out of the 109 programs had to be ruled out because there is no input whose execution run observes the error. In fact, these two buggy programs are syntactically different from, but semantically the same as the original correct program. *All reported distributions for our APG (abbreviation for Automated Path Generation) method are in percentage of 107 buggy programs, unless mentioned otherwise*. The results from our experiments are elaborated in this section. We compare our method with existing fault localization approaches [5, 16] in terms of quality of bug reports produced. We also report our time overheads of our approach and discuss some experience gained from our experiments.

Renieris and Reiss have proposed the nearest neighbor query method ("NN/Perm") in [16]. The NN/Perm method compares detailed code coverage between a failing run and the nearest successful run assuming that a set of successful runs is available. Recently, Cleve and Zeller have proposed the cause transition methods "CT/relevance" and "CT/infected" [5] for software fault localization. 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.

Quality of Bug Report. Table 4 shows the distribution of scores for four methods. The data for NN/Perm is taken from [16], and the data for CT/relevance and CT/infected are taken from [5]. Our method is shown as APG, an abbreviation for Automated Path Generation. From this table, we can see that our method's scores are comparable with those of the cause transition methods, and a little better than the scores for 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/Perm 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/Perm, 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 4: 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. We present the distribution of sizes of bug reports in Figure 5. We can see that 71% of our 107 bug reports contain at most seven branch statements. This indicates that the bug reports produced by our method are not voluminous and overwhelming.

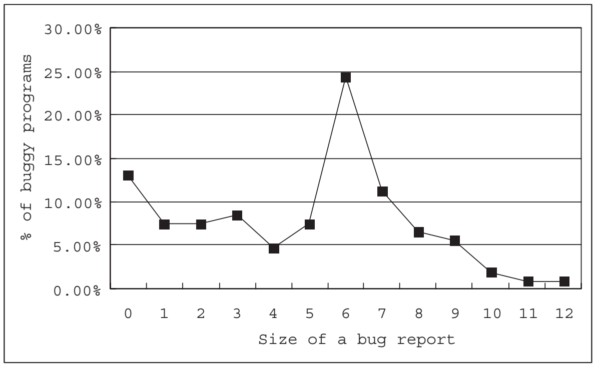

Figure 5: Size of bug reports produced by our method.

Time Overheads. Figure 6 shows the time overheads of our method. We cannot compare the time overheads with [5, 16] since these works did not report time overheads. The left bars (in lighter shade) present the distribution of time overheads for all 107 programs, and right bars (in darker shade) show the distribution of time overheads for programs whose bug reports can achieve scores 80% or better (47 of the 107 buggy programs achieve a bug report score of 80% or better). Our method found the successful run within 10 minutes for 75% of all 107 buggy programs, and for 83% of 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 following 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.

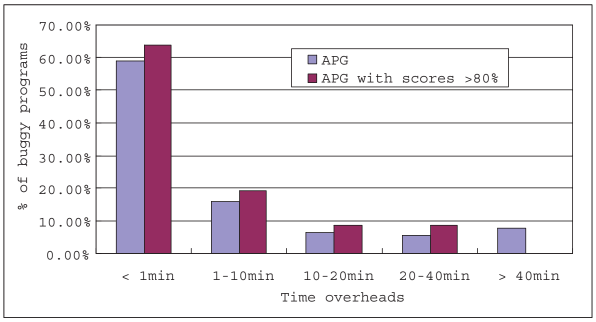

Figure 6: Time overheads of our method.

Remaining Manual Intervention. Our path generation algorithm generates execution runs close to the failing run and checks whether they are feasible and successful. As mentioned earlier, the check for feasibility is done automatically by the Simplify theorem prover. Checking whether a run is successful is however done manually in our experiments. The time for this manual check is not included in the time overheads reported in Figure 6. In our experiments, the first feasible run constructed by the path generation algorithm was a successful one for most buggy programs. In the worst case, we had to manually examine 5 feasible failing runs for success before the algorithm found a feasible successful run. Moreover, manual intervention in checking whether a run is successful is in some sense unavoidable. Otherwise the programmer has to precisely characterize the properties of a successful run, possibly as assertions; this eases our 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 a pool of available successful runs (such as [16]) will first require a classification of available program runs as failing or successful. This will typically require even more checks for each buggy program (one check classifies a given program run as failing/successful) than the 1 – 5 manual checks required in our experiments.

Experience. After generating the bug reports, we have studied these reports to understand what kind of errors can be easily localized by our approach. We found that many buggy programs of the Siemens suite have errors in branch statements. The erroneous branch statements are often correlated with later branch statements. If those later branch statements are evaluated differently during path enumeration, the erroneous branch statements also have to be evaluated differently, and are thus included into our bug report.

Our approach can also help effectively locate erroneous assignment statements, if branch statements which guard these assignments are included in the bug report. In addition, our method may effectively locate errors due to code omission from the program text; these errors cannot be localized by conventional methods like dynamic or relevant slicing (see [22]). In our method, the successful run may be constructed by evaluating differently the branch statement which the missing code is control dependent on. In the successful run, the missing code is not intended to be executed and hence we can locate the error.

Our method cannot effectively locate errors due to wrong initialization of global variables. These statements are not guarded by any branch statements. In such cases, our method often constructs a successful run by evaluating differently irrelevant branch statements. For example, several buggy programs of the `tcas` program in the Siemens benchmark suite contain such errors. For such buggy programs, our method obtains a low score (less than 30%).

7. 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/size of our bug reports as well as the time required to construct the bug reports.

We are currently extending our approach to report not only control locations, but also relevant program variables in these control locations ("searching in space" in Zeller's terminology [5]). One potential solution is to perform post-mortem analysis at the branch events which are evaluated differently in the successful execution. We can use dynamic slicing [1] to compute a set of relevant variables V at these branch events w.r.t. the variables whose values are observably erroneous. We can then employ the Delta Debugging technique of [24] to simplify the variable set V. Such information may help the programmer to better understand the wrong program execution.

8. REFERENCES

[1] H. Agrawal and J. Horgan. Dynamic program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1990.

[2] P. Anderson and T. Teitelbaum. Software inspection using codesurfer. In *the 1st Workshop on Inspection in Software Engineering*, 2001.

[3] 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.

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

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

[6] 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`.

[7] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

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

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

[10] S. Horwitz and T. Reps. The use of program dependence graphs in software engineering. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 392–411, 1992.

[11] 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)*, pages 191–200, 1994.

[12] 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.

[13] 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.

[14] B. Meyer. *Object-oriented Software Construction*. Prentice-Hall, 1988.

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

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

[17] 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 SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 432–449, 1997.

[18] D. Rosenblum. Towards a method of programming with assertions. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, 1992.

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

[20] 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.

[21] E. Shapiro. *Algorithmic program debugging*. PhD thesis, MIT Press, 1982. ACM Distinguished Dissertation.

[22] T. Wang and A. Roychoudhury. Using compressed bytecode traces for slicing Java programs. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 512–521, 2004.

[23] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[24] A. Zeller. Isolating cause-effect chains from computer programs. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 1–10, 2002.

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