DARWIN: An Approach for Debugging Evolving Programs[1]

Abhik Roychoudhury abhik@nus.edu.sg
Kapil Vaswani kapilv@microsoft.com

December 2008

Technical Report
MSR-TR-2008-91

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
http://www.research.microsoft.com

[1]DARWIN is named after Charles Darwin who explained evolution of species in his books *On the Origin of Species* and *The Descent of Man*. Our tool is geared to explain program evolution.

1 Introduction

Programmers do not write programs entirely from scratch. Rather, over time, a program gradually evolves with more features possibly being added. In industrial software development projects, this complexity (of software evolution) is explicitly managed via checking in of program versions. Validation of such evolving programs (say, to address possible bugs introduced via program changes) remains a huge problem in terms of program development. This adds to the cost for software maintenance, which is much larger than the initial software development cost. The cost of maintaining a software and managing its evolution is said to account for more than 90% of the total cost of a software project, prompting certain authors to call it as the "*legacy crisis*" [1].

To tackle the ever-growing problem of software evolution and maintenance, software testing methodologies have long been studied. Regression testing is a well-known concept which is currently employed in any software development project. In its simplest form, it involves re-testing a test-suite as a program moves from one version to another. In the past, lot of research work has been devoted to finding out which tests in a given test-suite do not need to be tested for the new program version (*e.g.*, see [2]). However, even among the tests which are tested in both program versions (old and new) — how do we find the root cause of a failed test case? Such failed test cases expose so-called "*regression bugs*" — the test cases pass in the old version, but fail in the new version. For any large software development project (particularly for commercial software products which regularly add features based on customer needs thereby leading to new program versions), finding root causes of regression bugs is a major headache! In this paper, we employ dynamic analysis techniques to address this issue.

Problem statement The problem we tackle can be summarized as follows. Consider a program P accompanied by a test-suite T, such that P passes all the tests in T. The test-suite could have been constructed automatically or manually based on some coverage criteria (statement coverage, branch coverage etc) of P and/or by following the evolution history of P. We call P as the old version or the stable version since it passes all the tests, that is, the observable output of P for all the tests in T is as expected by the programmer. Suppose P changes to a program P' and certain tests in T now fail, that is, their output does not meet the programmer's expectations. Let $t \in T$ be such a test. Our goal is to produce a bug report $Rep(t, P, P')$ such that Rep contains the explanation of why t fails in P' while passing in P. Of course, our goal is to find a bug report which is as concise as possible, while pinpointing the error cause statements to the programmer. Recall that P is the old stable program version and P' is the new buggy version. Typically any program debugging method highlights a fragment of a given program as bug report. So, we can expect $Rep(t, P, P')$ to be a fragment of the buggy version P' and/or old version P.

Try to use differencing methods? Program differencing methods (*e.g.*, see [3]) have long been used for identifying semantic differences between program versions by comparing their program dependence graphs. Since we are investigating the behavior of a specific test-case in two program versions, we cannot directly use these methods. Interestingly, our conversations with testing/development teams on the field revealed that large software development teams may try to perform differencing of traces (not programs) for finding root causes of regression bugs. Given a test t which passes in program P and fails in program P', one may compare the path traced by t in P vis-a-vis the path traced by t in P'. However, such a method is extremely syntactic. Since P and P' constitute two different programs, structurally comparing two paths of two different programs does not explicitly consider the semantics of the *changes* between P and P'.

Then, why not only look at the changes? Since by changing a stable program P to a new program P' we cause certain test cases to fail, one possibility is to focus on the changes alone. Indeed, this approach has been studied (*e.g.*, see [4]). The first step of such an approach will be to enumerate the changes between P and P'. Now, after the set of changes between P and P' are enumerated, by searching for the "culprit" among them, we can only report back a subset of the changes. However, the actual bug may be in P, but it could be manifested by the change from P to P'. A pointer which is mistakenly set to null in P but never dereferenced is indicative of such a situation. The mistake may only be observed in P' where the pointer is dereferenced. A good bug report should pinpoint the control location where the pointer is mistakenly set to null, not the control location where it is dereferenced.

What about trace comparison methods? In the last decade, trace comparison methods have been successfully used for localizing error causes in programs. Given a buggy program, the trace produced by a failed test-case (whose behavior is unexpected) is compared with the trace produced by a successful test-case (whose behavior is as expected). Techniques have been developed to determine (a) which successful test-case to use (*e.g.*, [5]), and (b) how to compare and report the differences between two program executions (*e.g.*, [6]). While such an approach is promising, directly employing them to our problem (by comparing the traces of two test-cases in the buggy program version) amounts to completely ignoring the program evolution. The successful test-case (in a buggy program) is supposed to capture "bug-free" program behavior. In our problem, we have a stable old program version representing bug-free behavior, which should be analyzed/used within the debugging method.

Basic idea behind our approach Given a stable program version P, a changed version P' and a test-case t which passes (fails) in P (P'), we compare the trace produced by t in P' with the trace produced by *another* test-case t' in P'. We automatically generate a test case t' satisfying the following properties:

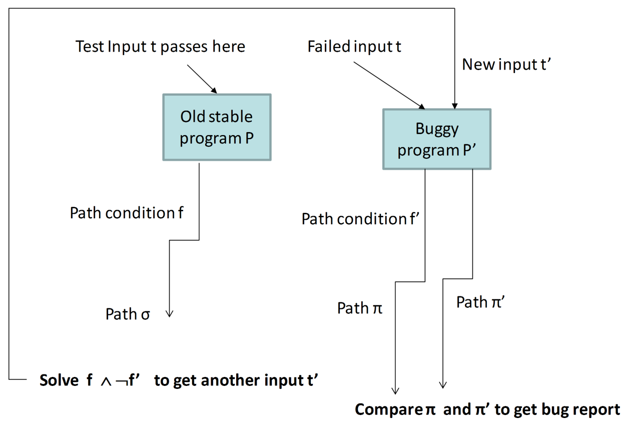


Figure 1: Pictorial description of basic debugging method

- t' and t follow the same program path in P.
- t' and t follow different program paths in P'.

Such a test t' can be found by computing path conditions of t in P and P'. Since t' and t follow the same program path in P – the behavior of t, t' are supposed to be "similar" in P (the stable program version). However, since t, t' follow different program paths in P' — their behaviors "differ" in P' (the buggy new version). By computing and highlighting the differences in their behavior, we highlight the possible causes of the error exposed by test-case t. A pictorial description of the debugging method appears in Figure 1. As we will see in the next section, this is only the core method and needs extensions.

We are motivated by the success of semantic trace comparison methods in locating error causes in adopting this approach. Note that, we cannot compare the traces of the failed test t in the two program versions since it amounts to a syntactic comparison of paths in two different implementations. We cannot compare the execution trace of t in the new program version P' with another execution trace of P' by simply ignoring the old stable program version P. Hence, we consider the evolution of P to P' in defining a test-case $t' \neq t$, and compare the execution trace of t in P' with the execution trace of t' in P'.

Contributions The main technical contribution of this paper is to provide an automated and scalable solution to a problem faced by any program development team on the field — locating causes of regression bugs in programs which evolve from one version to another. Given a test case which fails, our DARWIN tool works in two phases. In the first phase, DARWIN collects and suitably compose the path conditions of the failed test case in two program versions to generate an alternate test input. In the second phase, DARWIN compares the trace of the generated test input with the trace of the failed test input to produce a bug report. Trace comparison proceeds by employing string alignment methods (which are widely used in computational biology for aligning DNA sequences) on the traces; the branches which cannot be aligned appear in the bug report. Efficacy of our bug-report is demonstrated on

- programs from the well-known SIR benchmark suite [7], as well as
- a large real-life case study involving *libPNG* — a widely used open-source library for the Portable Network Graphics (PNG) image format.

Thus, we propose a simple, scalable methodology for finding root-causes of regression bugs. As a by-product of our approach, our proposal for using string alignment algorithms for trace comparison based software debugging is also novel, to the best of our knowledge. Moreover, the alternate test inputs we generate (for explaining failed regression tests) can be useful for future evolutions of the program as well.

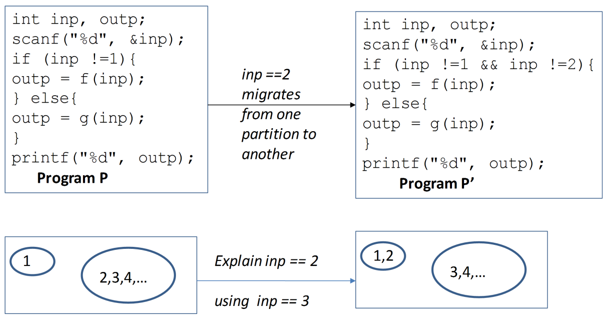
2 Overall Approach

In this section, we first present an overview of our approach via illustrative examples. We then give an outline of the different steps in our method.

To start with, consider a program fragment with a integer input variable `inp` – the program P in Figure 2. This is the old program version. Note that `f`, `g` are functions invoked from P. The code for `f`,`g` is not essential to understanding the example, and hence not given. Suppose the program P is slightly changed to the program P' in Figure 2 thereby introducing a "bug". Program P' is the new program version. As a result of the above "bug", certain test inputs which passed in P may fail in P'. One such test input is `inp == 2` whose behavior is changed from P to P'. Now suppose the programmer faces this failing test input and wants to find out the reason for failure. Our core method works as follows.

- We run program P for test case `inp == 2`, and calculate the resultant *path condition* f, a formula representing set of inputs which exercise the same path as that of `inp == 2` in program P. In our example, the path condition f is $inp \neq 1$.
- We also run program P' for test case `inp == 2`, and calculate the resultant *path condition* f', a formula representing set of inputs which exercise the same path as that of `inp == 2` in program P'. In our example, the path condition f' is $\neg(inp \neq 1 \wedge inp \neq 2)$.
- We solve the formula $f \wedge \neg f'$. Any solution to the formula is a test input which follows the same path as that of the test case `inp == 2` in the old program P, but follows a different path than that of the test case `inp == 2` in the new program P'. In our example $f \wedge \neg f'$ is

$$inp \neq 1 \wedge (inp \neq 1 \wedge inp \neq 2)$$



```
int inp, outp;
scanf("%d", &inp);
if (inp !=1){
outp = f(inp);
} else{
outp = g(inp);
}
printf("%d", outp);
```

```
int inp, outp;
scanf("%d", &inp);
if (inp !=1 && inp !=2){
outp = f(inp);
} else{
outp = g(inp);
}
printf("%d", outp);
```

Figure 2: Two example programs P, P' and their input space partitioning. The input 2 migrates across partitions in changing $P \rightarrow P'$. We choose an input 3 to explain the behavior of the failing input 2 — since $2, 3$ are in the same partition in P, but different partitions in P'.

A solution to this formula is any value of `inp` other than 1,2 — say `inp` == 3.

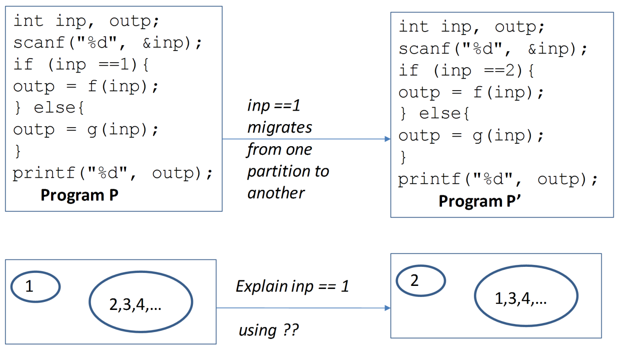
- Finally, we compare the trace of the test case being debugged (`inp == 2`) in program P', with the trace of the test case that was generated by solving path conditions, namely (`inp == 3`). By comparing the trace of `inp == 2` with the trace of `inp == 3` in program P' we find that they differ in the evaluation of the branch `inp !=1 && inp != 2`. Hence this branch is highlighted as the *bug report* — the reason for the test case `inp == 2` failing in program P'.

The above example clarifies the idea behind our method. We are assuming that P and P' have the same input space, and we consider the partitioning of program inputs based on paths —- two inputs are in the same partition iff they follow the same path. Then, as P changes to P' certain inputs migrate from one partition to another. Figure 2 illustrates this paritioning and partition migration. If we consider the change $P \rightarrow P'$ the behavior of the failing input `inp == 2` is explained by comparing its trace with the trace of `inp == 3`, an input in a different partition in the new program P'. Furthermore, `inp == 3` and `inp ==2` lie in the same partition in the old program P.

Sometimes, given two program versions P, P' and a test input t which passes (fails) in P (P') — we may not find a meaningful alternate input by solving $f \wedge \neg f'$. Consider the example programs in Figure 3 and their associated input space partitioning. In this case there are at least two inputs which migrate across partitions in changing the program from P to P'. Suppose we have the task of explaining the behavior of `inp == 1`.

The path condition f of `inp == 1` in P is $inp = 1$ while the path condition f' of `inp == 1` in P' is $inp \neq 2$. So, in this case $f \wedge \neg f'$ is

$$inp = 1 \wedge \neg(inp \neq 2)$$


```
int inp, outp;
scanf("%d", &inp);
if (inp ==1){
outp = f(inp);
} else{
outp = g(inp);
}
printf("%d", outp);
```
Program P
inp ==1 migrates from one partition to another
Program P'
1
2,3,4,...
Explain inp == 1 using ??
2
1,3,4,...

```
int inp, outp;
scanf("%d", &inp);
if (inp ==2){
outp = f(inp);
} else{
outp = g(inp);
}
printf("%d", outp);
```

Figure 3: Two example programs P, P' and their input space partitioning. The input 1 is an example input which migrates partitions. How to find an input to explain its behavior?

which is unsatisfiable! The reason is simple, there is no input which shares the same partition as that of `inp == 1` in the old program!

The solution to the above dilemma lies in conducting our debugging in the old program version. If we find that $f \wedge \neg f'$ is unsatisfiable, we can solve $f' \wedge \neg f$. This yields an input t' which takes a different path than that of the failing input t in the old program version. We can now compare the traces of t and t' in the old program version to find the error root cause.

In our example Figure 3, $f' \wedge \neg f$ is

$$inp \neq 2 \wedge \neg(inp = 1)$$

This yields solutions which are different from 1 and 2, say `inp == 3`. We can now compare the trace of `inp == 3` with the trace of `inp == 1` in the old program. This highlights the branch `inp == 1` in the old program, as bug report.

The reader may think the above situation as odd — when a test case fails in a new program, we may return a fragment of the old program as bug report! But, indeed this is our thesis — the bug report returned by our debugging method will help the application programmer comprehend the *change* from the old program to the new program, rather than helping him/her comprehend the new program. Of course, given a branch in the old program as bug report, it can be related to a branch in the new program by dependence preserving program alignment methods (*e.g.* the work of [8] uses such program alignment as the very first step of their test generation method). Note that we do not espouse program change comprehension via a full-scale static alignment of program versions. Only after a bug report is generated via our dynamic analysis, if the bug report refers to the old program — one can relate the bug report to the new program via such program alignment.

In summary, the outline of our method is as follows. Given an old program version P, a new program version P', a test input t which passes in P and fails in P' — our method proceeds as follows.

1. Compute f, the path condition of t in P.
2. Compute f', the path condition of t' in P'.
3. Check whether $f \wedge \neg f'$ is satifiable. If yes, it yields a test input t'. Compare the trace of t' in P' with the trace of t in P'. Return bug report.
4. If $f \wedge \neg f'$ is unsatisfiable, find a solution to $f' \wedge \neg f$. This produces a test input t'. Compare the trace of t' in P with the trace of t in P. Return bug report.

It is noteworthy that $(f \wedge \neg f') \vee (f' \wedge \neg f)$ should be satisfiable, and hence we should get an alternate input from the steps given in the preceding. If $(f \wedge \neg f') \vee (f' \wedge \neg f)$ is unsatisfiable, the formula $(f \Leftrightarrow f')$ is valid — which means that the input t does not migrate from one partition to another of the input space while going from old program version to new program version.

3 Detailed Methodology

In this section, we elaborate on our method. As explained in the preceding, our method has two phases. In the first phase, path conditions are computed and solved to get an alternate program input. In the second phase, comparing the traces of two inputs yields a bug report. We now describe the two phases in further details.

3.1 Generation of alternate input

In the first step of our method, we need to execute the test case under examination t in both the program versions. Here we actually perform a concolic (concrete + symbolic) execution of t on each of the program versions. In other words, during the concrete execution of t along a program path π, we also accumulate a symbolic formula capturing the set of inputs which exercise the path π. This symbolic formula is the path condition of path π, the condition under which path π is executed. It is worth mentioning that our path conditions are calculated on the *program binary*, rather than the source code.

One issue that arises in the accumulation of path conditions is their solvability by constraint solvers. For example, for a program branch `if (x * y > 0)`, we will accumulate the constraint `x*y > 0` into the path condition. This may be problematic if our constraint solver is a linear programming solver. In general, we have to assume that the path condition calculated for a path π is an *under-approximation* of the actual path condition. Usually such an under-approximation is achieved by instantiating some of the variables in the actual path condition. For example, to keep the path condition as a linear formula, we may under-approximate the condition `x * y > 0` by instantiating either `x` or `y` with its value from concrete program execution.

Recall that, we need to solve the formula $f \wedge \neg f'$ for getting an alternate program input, where f (f') is the path condition of the test input t being examined in the old (new) program version. As mentioned earlier, the computed f, f' will be an under-approximation of the actual path conditions in old/new program versions. Let $f_{computed}$ ($f'_{computed}$) be the computed path conditions in the old (new) program versions. Thus

$$f_{computed} \Rightarrow f \qquad f'_{computed} \Rightarrow f'$$

As a result, we have:

$$(f_{computed} \wedge \neg f'_{computed}) \not\Rightarrow (f \wedge \neg f')$$

Thus, we cannot ensure that $f_{computed} \wedge \neg f'_{computed}$ is an under-approximation of $f \wedge \neg f'$. Hence, after solving $f_{computed} \wedge \neg f'_{computed}$ if we find a solution t' we also perform a *validation* on t'. The validation will ensure our required properties, namely: t, t' follow same (different) program paths in old (new) program version. Such a validation can be performed simply by concrete execution of test inputs t, t' in the old and new program versions.

Similarly, if we need to solve the formula $f' \wedge \neg f$ (that is if the formula $f \wedge \neg f'$ is found to be unsatisfiable), we perform a validation of the test input obtained by solving $f' \wedge \neg f$.

3.2 Comparison of traces in buggy program version

In the second phase of our method, we compare the traces of two program inputs. The two test inputs whose traces are generated are (a) the test input under examination t, and (b) the alternate test input t' generated in the first phase.

Comparison of program traces have been widely studied in software debugging, and various distance metrics have been proposed. Usually, these metrics choose an important characteristic, compute this characteristic for the two traces and report their difference as the bug report. Commonly studied characteristics (for purposes of debugging via trace comparison) include:

- set of executed statements in a trace,
- set of executed basic blocks in a trace,
- set of executed acyclic paths in a trace,
- sequence of executed branches in a trace,

and so on. A sequence-based difference metric (which captures sequence of event occurrences in an execution trace) may distinguish execution traces with relatively greater accuracy. In our work, we adopt a difference metric focusing on sequence of executed branches in a trace, but apply it for traces at the assembly code (instruction) level. We note that program instrumentation at the instruction level is well-understood and several mature tools exist (*e.g.*, [9]). After collecting and comparing the traces at the instruction level, we report

```
1.   while (lin[i] != ENDSTR) {
2.      m= ...
3.      if (m >= 0) {
4.         ...
5.         lastm = m;
6.      }
7.      if ((m == -1) || (m == i)){
8.         ...
9.         i = i + 1;
10.      }
11.      else
12.          i = m;
13.      }
14.  ...
15.  }
```

Figure 4: An example program fragment from SIR suite [7].

back the instructions appearing in the "difference" between the two traces at the source-code level for the convenience of the programmer.

We thus represent each trace as a string of instructions executed. In practice, we need not record every instruction executed; storing the branch instruction instances (and their outcomes as captured by the immediate next instruction) suffices. Given test inputs t and t', a comparison of the traces for these two inputs is roughly trying to find branches which are executed with similar history in both the traces, but are evaluated differently. In order to find branches with similar history in both the traces, we employ string alignment algorithms widely employed on DNA/protein sequences in computational biology (*e.g.*, see [10]). These methods produce an alignment between two strings essentially by computing their "minimum edit distance".

To illustrate the workings of our trace comparison method, consider the program fragment in Figure 4. This program is taken from a faulty version of the `replace` program from Software-artifact infrastructure repository (SIR) [7], simplified here for illustration. This piece of code changes all substrings s_1 in string `lin` matching a pattern to another substring s_2. Here variable `i` represents the index to the first un-processed character in string `lin`, variable `m` represents the index to the end of a matched substring s_1 in string `lin`, and variable `lastm` records variable `m` in the last loop iteration. The bug in the code lies in the fact that the branch condition in line 3 should be `if (m >= 0) && (lastm != m)`. At the ith iteration, if variable `m` is not changed at line 2, line 3 is wrongly evaluated to true, and substring s_2 is wrongly returned as output, deemed by programmer as an observable "error".

An execution trace exhibiting the above-mentioned observable error will execute $\langle 1, 2, 3, 4, 5, 7, 8, 9 \rangle$ in the ith loop iteration. An execution trace not exhibiting the error (*i.e.*, a successful execution trace) will execute $\langle 1, 2, 3, 7, 8, 9 \rangle$

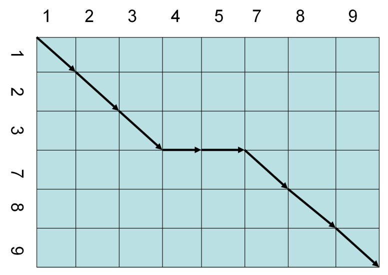


Figure 5: Conceptual view of aligning two execution traces. The traces in this example are from the program fragment in Figure 4.

in the ith loop iteration. Now, let us consider the alignment of these two execution traces — for simplicity we only show the alignment of their ith loop iterations.

Our string alignment method computes the smallest edit distance between the two traces — the minimum cost edits with which one string can be transformed to another. The edit operations are insert/delete/change of one symbol, and the cost of each of these operations need to be suitably defined. Conceptually this is achieved by constructing a two-dimensional grid. The rows (columns) of the grid are the symbols recorded in the first (second) execution trace. Finding the best alignment between the traces now involves finding the lowest cost path from the top-left corner of the grid to the bottom right corner of the grid. In each cell of the grid, we have choice of taking a horizontal, vertical or diagonal path. Horizontal (vertical) path means insertion (deletion) of a symbol in the first execution trace, while a diagonal path means comparing the corresponding symbols in the two traces. If we have to insert/delete a symbol we incur some penalty (say $\alpha > 0$). Moreover, if we compare two symbols of the two traces and record a mismatch we also incur some penalty (say β where typically $\beta > \alpha$). Of course, if we compare two symbols of the two traces and record a match, zero penalty is incurred. A least-cost alignment then corresponds to finding the path with minimum penalty from the top left corner to bottom right corner of the grid.

Note that the string alignment methods from computational biology (which we use in our work) often use complicated cost functions to capture the penalties of inserting/deleting/changing a symbol. However, for our trace comparison we always use the following: (i) cost of inserting a symbol = cost of deleting a symbol = α (a positive constant), and (ii) cost of changing a symbol = β (another positive constant greater than α).

Figure 5 shows the two-dimensional grid for the two traces $\langle 1, 2, 3, 7, 8, 9\rangle$ and $\langle 1, 2, 3, 4, 5, 7, 8, 9\rangle$ taken from the program in Figure 4.[1] A least-cost alignment

[1]We are explaining our example by presenting the traces at the level of statements. However, in our implementation they will be captured at the level of instructions.

found for these two traces (assuming $\alpha = 1$, $\beta = 2$) is shown in the figure via arrows. This corresponds to the following (expected) alignment.

```
1 2 3 _ _ 7 8 9
1 2 3 4 5 7 8 9
```

Having found the *alignment* between two traces, our bug report construction simply records the aligned branches in the two traces which have been evaluated differently. The sequence of these branches are presented to the programmer as *bug report*. In the preceding example, only the branch `3` will appear in the bug-report, thereby highlighting the error root-cause.

We have now explained our trace alignment and bug report construction method. The trace alignment can be computed by dynamic programming methods operating on the above-mentioned two-dimensional grid – where for each cell `[i,j]` of the grid we keep track of the lowest cost path from the top left corner (cell `[0,0]`) to cell `[i,j]`. However, a straightforward application of dynamic programming will involve space proportional to the product of the lengths of two traces being compared. In practice, this can lead to huge blow-up. For this reason, we have integrated existing linear-space string alignment methods [10] into our trace comparison. Given two traces of length `m`, `n` respectively, we will never construct the entire two-dimensional grid. Instead we find the least-cost path from cell `[0,0]` (top left corner) up to the middle row, that is, row `m/2`. In a similar way, we find the least cost path from cells in the middle row to cell `[m,n]` (bottom right corner). This gives us the cell in the middle row through which the least-cost path from cell `[0,0]` to cell `[m,n]` passes. We can now compute the least cost alignment in a divide-and-conquer fashion by finding least cost paths on smaller sub-grids.

3.3 Handling common programming errors

We now explain the suitability of our debugging methodology for different common kind of programming errors — branch errors, assignment errors and code-missing errors.

Branch errors We believe that our methodology is naturally suited for localizing branch errors — errors in branch conditions. This is because our method aligns a trace containing an observable error (say due to a branch condition) with another trace without the observable error in question. So, if the error is in the condition of a branch b, typically b will be evaluated differently (from the erroneous trace) in the trace without the observable error. The examples given in Section 2 illustrate this point.

Assignment errors Another important class of programming errors are called assignment errors — errors in an assignment statement. Since our bug report construction aligns/compares two traces based on their branch evaluations, how will such assignment errors get reflected in our bug report? As our trace alignment/comparison is control flow based, possible assignment errors need to be reflected via changed control flow. Inspired by the work on "co-operative bug isolation" [11, 12], we use an instrumenting compiler to insert extra code that checks various predicates at different program points. In other words, we introduce additional branches at different program point to make assignment errors observable via change in control flow. We introduce branches with branch conditions checking

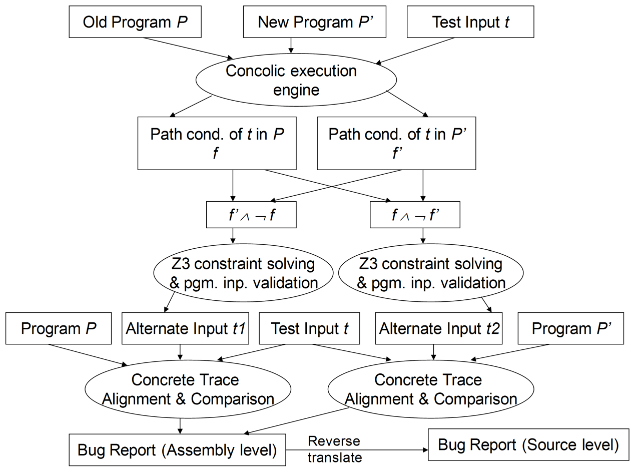
- function return values at each function return site, and
- binary constraints describing relationship of a program variable `x` with other variables of the same type, at each assignment to `x`. Thus, if `x`, `y` are of the same type — we introduce branches to check `x > y`, `x == y` and so on.

Code-missing errors Code-missing errors correspond to chunks of code being left out during change of a program. Such code will be missing in the new program version, but is present in the old program version. Whether the missing code chunk contains assignments (which, if they were present would have affected control flow via instrumented branches) or branches (which directly affect control flow) — the old program version P can be expected to have more paths than the new program version P'. Given a failing test input t, and f (f') being the path condition of t in P (P') — we can thus expect $f' \wedge \neg f$ to yield a solution. This will be a test input t' following the path of t in P', but following a different path than t in P (where the code missing in P' is present, leading to more branches and more paths). Thus, the traces of t' and t in P will be aligned and compared to yield a bug report. No additional change is needed in our methodology to handle code missing errors.

4 Implementation

We now describe our implementation setup for our debugging methodology. We call the integrated toolkit implementing our methodology as DARWIN — since it explains /debugs software evolution. The overall architecture of our DARWIN toolkit is summarized in Figure 6.

Within the DARWIN toolkit, we use a concolic (concrete + symbolic) execution engine for computing the path condition of a given program execution. Our concolic execution engine is essentially a tool for directed test generation [13] — starting with a random test, it performs concolic (concrete + symbolic) execution of the test. Thus, during concrete execution of the random test, it also performs symbolic execution to compute the path condition. In fact, our concolic execution engine systematically explores various program paths thereby exposing test cases which fail. In this work, we do not use the path exploration in our concolic execution tool — the tool is only used to compute the path condition for a given execution. Our concolic execution engine works on Windows x86 binaries, so our path conditions are computed at the level of binaries, rather



Old Program P
New Program P'
Test Input t
Concolic execution engine
Path cond. of t in P f
Path cond. of t in P' f'
f' ∧ ¬ f
f ∧ ¬ f'
Z3 constraint solving & pgm. inp. validation
Z3 constraint solving & pgm. inp. validation
Program P
Alternate Input t1
Test Input t
Alternate Input t2
Program P'
Concrete Trace Alignment & Comparison
Concrete Trace Alignment & Comparison
Bug Report (Assembly level)
Reverse translate
Bug Report (Source level)

Figure 6: Architecture of our DARWIN toolkit. It takes an old program version P, a new program version P' and a test input t which passes in P but fails in P'. The output is a bug report explaining the behavior of test t. The entire flow is *automated*.

than source code. In particular, the variables appearing in the path condition correspond to the different bytes of the program input.

Given program versions P, P' and a test input t which passes (fails) in P (P') — we compute the path condition f (f') of t in P (P'). The formula $f \wedge \neg f'$ is then solved by the publicly available Z3 constraint solver [14]. Z3 is an automated satisfiability checker for typed first-order logic with several builtin theories for bitvectors, arrays etc. The Z3 checker serves as a decision procedure for quantifier-free formula. Indeed this is the case for us, since our formulas do not have universal quantification and any variable is implicitly existentially quantified. Since f, f' are path conditions, they are conjunctions of primitive constraints, that is say

$$f' = (\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_m)$$

where ψ_i are primitive constraints. Then

$$f \wedge \neg f' = (f \wedge \neg\psi_1) \vee (f \wedge \neg\psi_2) \vee \ldots \vee (f \wedge \neg\psi_m)$$

Each $f \wedge \neg\psi_i$ is a conjunction. A solution to any $f \wedge \neg\psi_i$ is a solution for $f \wedge \neg f'$. We solve each $(f \wedge \neg\psi_i)$ separately using Z3, and Z3 emits a solution if the formula is satisfiable. Thus we obtain at most m solutions to the formula $f \wedge \neg f'$. Each of these solutions are test inputs which now undergo validation — we check whether it follows same (different) path as that of t in P (P'). Recall that the validation is required since our concolic execution engine may perform under-approximations while computing path conditions.

At this stage, we elaborate a low-level issue which makes a significant difference in our experiments. While solving $f \wedge \neg\psi_i$, we do not submit the entire formula $f \wedge \neg\psi_i$ to the Z3 solver. Recall that f, being a path condition, is a conjunction of primitive constraints. We first find the variables Var_i appearing in ψ_i. We then perform a least fixed-point computation to find those variables appearing in f which are (directly/indirectly) "affected" by Var_i — they appear with Var_i, or some variable affected by Var_i in a primitive constraint. Of course all variables in Var_i are affected by Var_i by default. Now, only the primitive constraints of f which contain variables "affected" by Var_i need to be considered while solving $f \wedge \neg\psi_i$. As an example suppose f is

$$x > y \wedge y > 10 \wedge z > w \wedge w > 0$$

and ψ_i is $x > 100$. While solving $f \wedge \neg\psi_i$ we can then only solve for

$$x > y \wedge y > 10 \wedge \neg(x > 100)$$

to get the new solutions for x, y from Z3. The solutions for the other variables (in this case z, w) are unchanged — these are obtained from the failing test input which generated the path condition f in the first place. The optimization mentioned here substantially cuts down the size of formulae submitted to Z3, and the solution space that needs to be explored by Z3. More importantly, it makes *minimal changes* to the failed test input to generate the alternate test inputs.

Given the solutions of $f \wedge \neg f'$ we validate them, that is, we check whether the traces for these inputs follow the same (different) path as the failing input in the old (new) program version. Let the set of validated solutions of $f \wedge \neg f'$ (which give us alternate test inputs to work with) be $Inputs^{f,f'}_{validate}$. We then generate the execution traces of each of these test inputs in the buggy program version P'. Each such trace is aligned and compared with the execution trace of the failing test t in P'. This yields a bug report which is used to localize the root cause of the error in program version P' which is manifested by failing test t. The bug report contains a sequence of branches. We then choose an input $t' \in Inputs^{f,f'}_{validate}$ whose bug report contains the least number of branches, that is, the execution trace of t' differs in the evaluation of least number of branches w.r.t execution trace of t. The bug report corresponding to t' is returned.

In case we find $f \wedge \neg f'$ to be unsatisfiable or none of the solutions of $f \wedge \neg f'$ can be validated, we solve $f' \wedge \neg f$ in a similar fashion. Again, this can yield many solutions which we then validate. For the validated solutions, we align/compare their traces in the old program version P with the trace of t in P. This yields several bug reports, from among which we again choose one which contains the least number of branches.

By following the steps mentioned in the preceding (solving either $f \wedge \neg f'$ or $f' \wedge \neg f$), we obtain a sequence of branches at the assembly level as bug report. Using standard compiler level debug information, these are reverse translated back to the source code level — thereby allowing use of the bug report by the application programmer. It is worthwhile to mention that given the old/new

Benchmark	**Versions**	**Description**	**LOC**
`replace`	v1-v10	Pattern matching and substitution	564
`tcas`	v1-v10	Collision avoidance system	174
`libPNG`	v1.07	An open-source library for reading/writing PNG images	32904

Table 1: Description of programs used in our experiments.

program versions and the test input whose behavior needs to be explained, *our bug report construction is fully automated.*

5 Experimental Results

We evaluated our fault localization technique on two aspects, namely, the accuracy of localization and the overall time taken to generate the bug report. We present the results of our evaluation.

5.1 Benchmarks and setup

We employed our fault localization tool DARWIN on some of the C benchmarks from the Software-artifact Infrastructure Repository (SIR). We could not use several SIR benchmarks because of the limitations of our compiler (Microsoft Visual Studio C/C++ compiler 15.00) and the limitations of the concolic execution engine we used. In particular, certain SIR benchmarks such as flex, grep do not compile for Windows and our concolic execution engine works on Windows x86 binaries. Hence we could not involve these programs in our experiments.

Moreover, the concolic execution engine we use only works for programs whose input is a stream of bytes (typically as a file). Hence it cannot work with programs taking structured input, and is most suited for media applications (as evidenced by our experiments). The restrictions imposed by our concolic execution engine are only limitations of a tool we use inside our DARWIN toolkit. These are *not* inherent restrictions of DARWIN's architecture.

For each benchmark in SIR, there are several accompanying buggy program versions. These defects have been manually injected by the benchmark writers — relaxing/tightening of branch conditions, wrong values in assignment statements and code missing errors. The SIR benchmarks used in our experiments are described in Figure 1.

Since each SIR benchmark has a golden program, several buggy versions and a test-suite, we can employ our debugging method as follows on each buggy version. We take the golden program as the original stable program P and the buggy version as the "new" program P'. Furthermore, we find a failing test-case from the test-suite accompanying the benchmark — a test which produces different output in P and P'. We then use our method to explain this failing test case, and see whether the bug-report produced by our method pinpoints the line containing the actual error (the change between P and P').

To get a feel for the use of our method in debugging large programs with real faults (unlike seeded faults in SIR), we also tried our hand on the `libPNG` open source library[2], a library for reading and writing PNG images. We used a previous version of the library (1.07) as the buggy version. This version contains a known security vulnerability, which was subsequently identified and fixed in later releases. An PNG image that exploits this vulnerability is also available online. As the reference implementation or stable version, we used the version in which the vulnerability was fixed (1.2.21). Assuming this vulnerability was a regression bug, we used our tool to see if the vulnerability could be accurately localized.

For our experiments, we generated alternate inputs by composing path conditions and aligned/compared the execution traces of these alternate inputs with the execution trace of the failing test input. Recall from Section 3 that the trace alignment is parameterized by two penalty functions α, β giving the penalty for (i) a missing/additional symbol and (ii) a symbol mismatch respectively. Our penalty functions are in fact constants, with $\alpha = 1$ and $\beta = 2$.

5.2 Accuracy of fault localization

We measured the accuracy of fault localization by classifying the bug reports we generated into one of the following four classes.

- *Class 1*: Localized bug to the same line as the actual root cause.
- *Class 1a*: Localized bug to a line which is syntactically identical to the actual root cause.
- *Class 2*: Localized bug to the same function as the actual root cause.
- *Class 3*: Localized bug to either a caller or callee of the function containing the actual root cause.
- *Class 4*: Could not localize bug to either class 1, 1*a*, 2, or 3.

Class 1 corresponds to the case where the bug report goes spot on, and locates the line containing error root-cause. Class 1a is a special case of class 1, which we highlight for explaining the issues in bug report comprehension. Class 1a considers the fact that there might be code that is repeated, such as a branch test being repeated several times in a program. When we compute the path condition, such a branch test will be logically represented only once in the path condition. Thus, even if we can localize the error to an erroneous branch — this branch may appear several times in different lines of the program. Class 1a captures this situation.

Class 2 corresponds to the case where we localize the error to the same function, but not to the erroneous line. Class 3 corresponds to the case where we cannot localize the error to the same function, but to a caller or callee. Finally class 4 corresponds to the case where none of the above-mentioned classes apply.

Benchmark	**Version**	**Class of localization**	**# inputs generated**
replace	v1	1	21
	v2	2	24
	v3	3	3
	v4	3	5
	v5	1	13
	v6	1	24
	v7	1	4
	v8	3	5
	v9	1	7
	v10	1	5
tcas	v1	1	1
	v2	4	0
	v3	1	3
	v4	1	3
	v5	1	4
	v6	2	1
	v7	3	1
	v8	3	1
	v9	4	0
	v10	3	1
libPNG	v1.07	1a	4

Table 2: The accuracy of fault localization for programs in our benchmark suite.

Benchmark	replace	tcas	libPNG
Concolic execution	18.07	14.58	24.13
Constraint solving	12.54	2.99	2.35
Trace new inputs	160.34	12.69	81.31
Branch alignment	7.83	0.52	1.65
Total	198.79	30.79	109.44

Table 3: Time taken to localize faults. The times are reported in seconds and represent average across all version of the benchmark.

It is worth mentioning that our criterion for judging the accuracy of bug reports is *considerably more stringent* than the bug report scoring criteria [15, 16, 5] which has been used in literature. Such bug report scoring criteria capture the portion of the program that needs to be examined by following dependence chains from the statements in the bug report until we hit the error root-cause. On the other hand, we have categorized our bug reports with a closer look at the *effort* put in by the programmer in comprehending the bug report. We do not require/expect the programmer to manually perform any semantic analysis of the bug report (such as following program dependency chains and identifying the error root cause when it is hit) — any semantic analysis is integrated into the bug report construction itself. In fact, classes 1 and 1a in our categorization involve close to zero manual effort by the programmer in using the bug report — a syntactic search on the lines in the bug report suffices. Classes 2,3 involve more programmer effort in using the bug report — the functions containing the lines in the bug report are highlighted which the programmer looks into.

Overall accuracy Table 2 shows the results of our evaluation. For each buggy version of a golden program, the table shows the category to which the bug reports belongs, and the number of alternate inputs generated. We find that our approach was able to accurately localize a large fraction of the bugs in the benchmark programs. There are only two cases marked as category 4 bug report in Table 2. In both of these cases, our tool did not produce any bug report at all, since no alternate inputs were found! While impossible in principle, this scenario arises in practice because of the under-approximations made by our concolic execution engine in computing path conditions.

In old or new program? Recall that our debugging method tries to return fragment of the new program version, failing which it tries to return a fragment of the old program version. Out of 21 buggy versions across `tcas/replace/libPNG` on which we ran experiments, the breakdown is as follows.

- In 17 programs, a fragment of the new program version was returned as bug report.
- In 2 programs, a fragment of the old program version was returned as bug report.
- In 2 programs, no bug report could be constructed due to the absence of alternate inputs (resulting from underapproximations inside concolic execution engine).

Working of the tool We now illustrate the working of our approach in more detail using a concrete example. In Figure 7, we show a buggy code snippet from the `replace` benchmark version `v1`. The bug is an off-by-one error in the array reference at line 9. When passed as the first parameter, the string

[2]`http://www.libpng.org/`

```
1 void dodash(char delim, char* src, int * i,
2 char * dest, int *j, int maxset) {
3   int k;
4   bool junk;
5   char escjunk;

6   while ((src[*i] != delim) && (src[*i] != ENDSTR)) {
7     /* BUG: Off-by-one error */
8     /* if (src[*i - 1] == ESCAPE) */
9     if (src[*i] == ESCAPE) {
10     escjunk = esc(src, i);
11     junk = addstr(escjunk, dest, j, maxset);
12    } else
          ...
     }
```

Figure 7: Buggy code from `replace` benchmark version 1

`%[0-9][^9-B][@t][^a-c]` exercises this bug and causes a failure. Our DARWIN toolkit is able to generate a new input string `%[0-9][^9-B][00][^a-c]`. Note that in this string, the character at the 13th position has been changed from an ESCAPE character (`@`) to `0` (which also necessitates changes in few other characters of the input since they are "related" to the ESCAPE character due to correlation of branch outcomes in the `replace` program). This change in input causes the program to trace a different path in the buggy version (in which the condition at line 9 is evaluated to `false`) while following the same path in the bug-free program version. An alignment of the traces obtained using these input strings clearly shows the root cause.

Amount of code examined post-mortem We also evaluated the amount of code to be examined for localizing the bug using our bug reports. We computed the number of lines of code required to localize the bug by ranking the inputs we generated based on the number of branches that align with the trace of the buggy input — stepping through the traces in order according to this rank and stopping when we hit the error root cause. We find that the average number of lines to be examined in both `tcas` and `replace` benchmarks is about 10% of the program size. The number of lines to be examined in `libPNG` is very small — less than 0.01% of the program size. We now elaborate on our debugging experience with the `libPNG` case study.

5.3 Experience with `libPNG`

Unlike the SIR benchmarks which contain seeded faults, the `libPNG` case study involves localizing an *actual real-life error* using our method. We took a buggy version of `libPNG` (with a documented test-case showing the bug) as our new

```
if (!(png_ptr->mode & PNG_HAVE_PLTE))
{
   png_warning(png_ptr, "Missing PLTE before tRNS");
}
else if (length > (png_uint_32)png_ptr->num_palette)
{
   png_warning(png_ptr, "Incorrect tRNS chunk length");
   png_crc_finish(png_ptr, length);
   return;
}
```

Figure 8: Buggy code fragment from `libPNG`

program version. Another version of `libPNG` where the error was fixed by developers serves as the stable program version.

The bug we localized is a remotely exploitable stack-based buffer overrun error in `libPNG`. Under certain situations, the `libPNG` code misses a length check on PNG data prior to filling a buffer on the stack using the PNG data. Since the length check is missed, the buffer may overrrun possibly crashing web-browsers such as Mozilla. What is worse, such a bug may remotely exploited by emailing a bad PNG file to another user who uses a graphical e-mail client for decoding PNGs with a vulnerable `libPNG`! In Figure 8, we show a code fragment of `libPNG` showing the error in question. If the first condition `!(png_ptr->mode & PNG_HAVE_PLTE)` is true, the length check is missed, leading to a buffer overrun error. A fix to the error is to convert the `else if` in Figure 8 to an `if`. In other words, whenever the length check succeeds, the control should return.

We now explain some of the complexities we face in localizing such a bug using approaches other than ours. Suppose we have the buggy `libPNG` program and a bad PNG image which causes a crash due to the above error. If we want to localize the error by an analysis of the erroneous execution trace starting from the observable error — it is very hard to even define the observable error since the program crashes. Even if the buffer being overrrun is somehow defined as the observable error, tracking program dependencies from the observable error can be problemmatic for the following reason. The `libPNG` library is used by a client which inputs an image, performs computation and outputs to a buffer (the one that is overrun due to error inside `libPNG`). In this case, we are debugging the sumtotal of the client along with the `libPNG` library. Since almost all statements in the client program and many statements in `libPNG` involve manipulation of the buffer being overrun itself — a program dependence or slicing based approach, however refined, seems to highlight almost the entire client program as well as large parts of the `libPNG` library.

If we want to employ statistical bug isolation methods (which instrument predicates and correlate failed executions with predicate outcomes), the key is to instrument the "right predicate". In this case, the predicates in question (such as `!(png_ptr->mode & PNG_HAVE_PLTE)`) contain pointers and fields. Hence they would be hard to guess using current statistical debugging methods which usually consider predicates involving branch conditions, return values and scalar variables.

If we want to compare the trace of the bad PNG image (which exposes the error) with the trace of a good PNG image (which does not show the error) for debugging — how do we get the good PNG image? Even if we have a pool of PNG images from which we choose one (for comparison with the trace of the bad PNG image) – making the "right" choice becomes critical to the utility of the bug report. More importantly, such a method is extremely sensitive to the pool of PNG images available at hand.

In our debugging method, given the bad PNG image — we *construct* an alternate PNG image via semantic analysis of the execution traces of the bad PNG image in the two program versions. This image is a "minimal modification" of the bad PNG image — our analysis only minimally changes the bad PNG image to get an image which follows the same (different) program path in correct (buggy) program version. By aligning/comparing the traces of the two images — our bug report localizes the error to the branch condition `!(png_ptr->mode & PNG_HAVE_PLTE)`. Indeed, this is the first branch where the traces of the two images do not align. We then examine all the occurrences of this condition within the same function (there were actually 8 of them since the same condition is tested several times). By examining these 8 lines (the `libPNG` library has more than 32,000 lines of code), we zoom into the error root-cause.

5.4 Analysis time

In our approach, the time required to analyze a fault and generate a bug report can be attributed to the time taken for executing both the buggy program and the reference program concolically, solving constraint, executing the new input and aligning branches. Table 3 gives the breakdown of the time (in seconds) spent in each of these tasks for the benchmarks in our test suite. For the SIR benchmarks, we report the average time across all versions of the benchmark.

Interestingly, we find that a large fraction of the time is spent in tracing the new inputs. We expected the concolic execution or the constraint solving to be the bottleneck but those tasks do not seem to dominate the debugging cycle. For `libPNG`, concolic execution accounts for only 25% of the total time.

On the whole, analyzing a buggy input and generating a bug report can take between 30 seconds and 4 minutes. We believe that these time spans are reasonable given the amount of time developers typically spend in debugging.

6 Related Work

Developing validation methods for evolving programs is an important problem, since any large software system moves from one version to another. One of the established efforts in this direction are the works on regression testing which focus on which tests need to be executed for a changed program. Even though regression testing in general refers to any testing process intended to detect software regressions (where a program functionality stops working after some change), often regression testing amounts to re-testing (of tests from existing test-suite). In the past, there have been several work directions which go beyond re-testing all of the tests of an existing test-suite. One stream of work has espoused test selection [2, 17] — selecting a subset of tests from existing test-suite (before program modification) for running on the modified program. Another stream of works propose test prioritization [18, 19] — ordering test cases in existing test suite to better meet testing objectives of the changed program. Finally, most recently [8] has proposed test-suite augmentation — developing new tests (over and above the test-suite existing prior to program change) to stress the effect of the program changes. We note that our technique is complementary to regression testing since regression testing seeks to detect or uncover software regressions, whereas we seek to explain (already detected) software regressions.

Program differencing methods [3, 20, 21] try to identify differences between two program versions. Indeed, this is the first step towards detecting errors introduced due to program changes — identifying the changes themeselves! The works on change impact analysis are often built on such program differencing methods (*e.g.*, see [21]) — where the analysis identifies not only the changes (by comparing two program versions), but also which tests are affected by which changes. A recent work [22] uses symbolic execution to accurately capture behavioral differences between program versions. Overall, the works on program differencing try to identify (via static analysis) possible software regressions, rather than finding the root-cause of a given software regression.

Dynamic analysis based change detection have also been studied (*e.g.*, [23], which analyzes via regression testing the change in dependencies between parts of a program). These works focus on qualitative code measures and the possible impact of program changes. Instead we focus on the specific issue of root-causing a bug that *has* surfaced due to program changes.

In the area of computer security, deviation detection of various protocol implementations (such as HTTP) have been studied [24, 25]. This problem involves finding corner test cases in which two implementations of the same protocol might "deviate" in program output, either by pseudorandom test generation or by semantic analysis. This problem setting is somewhat different from ours since two implementations of the same security protocol may be widely different (*e.g.*, one implementation is optimized for resource usage, while the other is not), rather than being two program versions. Moreover, finding deviating program inputs bears similarities with uncovering software regressions. Again, our work is focused on explaining already uncovered software regressions. In other words, given a deviating input (deviating in behavior in two program versions) — we will typically find an input which behaves *similarly* in both program versions, which can allow us to explain the behavior deviation for the deviating input.

Turning now to works on software debugging, the last decade has seen a spurt of research activity in this area. Some of the works are based on static analysis to locate common bug patterns in code (*e.g.*, [26]), while others espouse a combination of static and dynamic analysis to find test cases which expose errors (*e.g.*, [27]). Another section of works address the problem of software fault localization (typically via dynamic analysis) — given a program and an observable error for a given failing program input, these works try to find the root cause of the observable error. Our work solves this problem of fault localization, albeit for evolving programs.

The works on software fault localization proceed by either (a) dynamic dependence analysis of the failing program execution (*e.g.*, [28, 29, 30]), or (b) comparison of the failing program execution with the set of all "correct" executions (*e.g.*, see [31]), or (c) comparsion of the failing program execution with one chosen program execution which does not manifest the observable error in question (*e.g.*, [6, 15, 5]). Our work bears some resemblance to works which proceed by comparing the failing program execution with one chosen program execution. The first phase of our approach tries to construct an alternate input with whose trace we compare the failing program execution, and the second phase of our approach involves a trace alignment/comparison. However, the *main novelty* in our approach lies in its ability to consider two different versions (of an evolving program) in the debugging methodology. As a side remark, note that our trace comparison works on binary level traces (rather than source level) and relies on string alignment techniques (rather than dynamic program dependencies).

As mentioned earlier, the work of [4] studies the debugging of evolving programs by (a) enumerating the changes between two versions, and (b) searching among the changes for a bug report. However, this is restricted to only reporting the changes as error causes since no program analysis is involved. Errors present in the old version which get manifested due to changes cannot be explained using such an approach.

In summary, existing works on program analysis based software debugging have not studied the debugging of evolving programs. In particular, the possibility of exploiting previous program versions (which were thoroughly tested) for finding the root-cause of an observable error has not been studied. This indeed is the key observation behind our approach. Moreover, existing works on evolving software testing/analysis primarily focus on finding test cases which show differences in behavior of different program versions. These works do not prescribe any method for explaining/debugging a failed test case — an issue that we study in this paper.

7 Discussion

In this paper, we have presented a debugging methodology and tool for evolving programs. Our DARWIN toolkit takes in two program versions and explains the behavior of a test case which passes in the old program version, while failing in the new program version. Our experience with the libPNG case study demonstrate the scalability of our method to large programs, and the method's ability to localize real-life faults. We believe that a debugging approach such as ours can truly be useful for developers, who are often faced with hard-to-locate regression bugs when a large software system changes from one version to another.

In future, we plan to link up the DARWIN toolkit with regression testing approaches, leading to an integrated testing and debugging tool for evolving programs Yet another direction of work will be to study test-suite augmentation of evolving programs — when a program changes from one version to another should any additional tests be tested? Construction of such additional tests may be inspired by our alternate input generation method — given a test in a test suite, we can compose and solve its path conditions in different program versions to get new tests.

References

[1] R. Seacord, D. Plakosh, and G. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices.* Addison-Wesley, 2003.

[2] Y.F. Chen, D.S. Rosenblum, and K.P. Vo. Testtube: a system for selective regression testing. In *ICSE*, 1994.

[3] S. Horowitz. Identifying the semantic and textual differences between two versions of a program. In *PLDI*, 1990.

[4] A. Zeller. Yesterday, my program worked. Today, it does not. Why? In *ESEC/FSE*, 1999.

[5] L. Guo, A. Roychoudhury, and T. Wang. Accurately choosing execution runs for software fault localization. In *CC*, 2006.

[6] A. Zeller. Isolating cause-effect chains from computer programs. In *FSE*, 2002.

[7] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 2005. `http://www.cse.unl.edu/~galileo/sir`.

[8] R. Santelices, P.K. Chittimalli, T. Apiwattanapong, A. Orso, and M.J. Harrold. Test-suite augmentation for evolving software. In *ASE*, 2008.

[9] PIN. Dynamic binary instrumentation tool. `http://rogue.colorado.edu/pin/`.

[10] K-M. Chao, R.C. Hardison, and W. Miller. Recent developments in linear space alignment methods: A survey. *Journal of Computational Biology*, 1, 1994.

[11] B. Liblit, M. Naik, A. Zheng, A. Aiken, and M. Jordan. Scalable statistical bug isolation. In *PLDI*, 2005.

[12] B. Liblit. *Cooperative Bug Isolation.* PhD thesis, UC Berkeley, 2005.

[13] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI*, 2005.

[14] L. de Moura and N. Bjorner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[15] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *ASE*, 2003.

[16] H. Cleve and A. Zeller. Locating causes of program failures. In *ICSE*, 2005.

[17] G. Rothermel and M. J. Harrold. A safe efficient regression test selection technique. *TOSEM*, 6, 1997.

[18] S. Elbaum, A. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. In *ISSTA*, 2000.

[19] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in a development environment. In *ISSTA*, 2002.

[20] T. Apiwattanapong, A. Orso, and M.J. Harrold. A differencing algorithm for object-oriented programs. In *ASE*, 2004.

[21] X. Ren, F. Shah, F. Tip, B. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of java programs. In *OOPSLA*, 2004.

[22] S. Person, M.B. Dwyer, S. Elbaum, and C. Pasareanu. Differential symbolic execution. In *FSE*, 2008.

[23] O. Giroux and M.P. Robillard. Detecting increases in feature coupling using regression tests. In *FSE*, 2006.

[24] P. Oehlert. Violating assumptions with fuzzing. *IEEE Security and Privacy Magazine*, 3, 2005.

[25] D. Brumley, J. Caballero, Z. Liang, J. Newsome, and D. Song. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *USENIX Security Conference*, 2007.

[26] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *OOPSLA onward session*, 2004.

[27] C. Csallner and Y. Smaragdakis. DSD-Crasher: a hybrid analysis tool for bug finding. In *ISSTA*, 2006.

[28] M. Sridharan, S.J. Fink, and R. Bodik. Thin slicing. In *PLDI*, 2007.

[29] X. Zhang, N. Gupta, and R. Gupta. Pruning dynamic slices with confidence. In *PLDI*, 2006.

[30] X. Zhang, S. Tallam, N. Gupta, and R. Gupta. Towards locating execution omission errors. In *PLDI*, 2007.

[31] T. Ball, M. Naik, and S.K. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In *POPL*, 2003.