Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
Electronic Notes in Theoretical Computer Science 266 (2010) 3–15
www.elsevier.com/locate/entcs

Debugging as a Science, that too, when your Program is Changing

Abhik Roychoudhury [1]

School of Computing
National University of Singapore

Abstract

Program debugging is an extremely time-consuming process, and it takes up a large portion of software development time. In practice, debugging is still very much of an art, with the developer painstakingly going through volumes of execution traces to locate the actual cause of an observable error. In this work, we discuss recent advances in debugging which makes it systematic scientific activity in its own right. We explore the delicate connections between debugging and formal methods (such as model checking) in the overall task of validating software. Moreover, since any deployed software undergoes changes in its lifetime, we need debugging methods which can take the software evolution into account. We show how symbolic execution and Satisfiability Modulo Theories (SMT) solvers can be gainfully employed to greatly automate software debugging of evolving programs.

Keywords: Software Debugging, Symbolic Execution.

1 Introduction

Software development is at the heart of our everyday societal infra-structure, and software debugging is key to reliable software development. A study in 2002 by the United State Department of Commerce's National Institute of Standards and Technology (NIST) mentioned that software bugs or errors cost $60 billion annually or about 0.6% of USA's gross domestic product [1]. The same study also mentions that more than a third of these costs, or $22 billion can be saved by building better testing and debugging infra-structure.

The limited short-term memory of human beings causes smart programmers to make dumb mistakes. Moreover, the effect of errors thus introduced show up in unexpected places during software testing. It is truly a difficult problem to locate the error cause from an observed error - the art of *debugging*. In other words, debugging denotes method(s) for detecting causes of unexpected observable

[1] Email: abhik@comp.nus.edu.sg

doi:10.1016/j.entcs.2010.08.045

behavior in computer programs (such as a program crashing, or an unexpected output value being produced).

The social and economic importance of software debugging is enormous. Today, more and more functionalities in our daily life are controlled by software. Due to the overwhelming growth of embedded systems, software-controlled devices are ubiquitous — automotive control, avionics control and consumer electronics being prominent application domains. Many of these software are safety-critical and should be validated extensively. This accentuates the importance of software debugging in our daily lives.

Technically, the task of software debugging is an extremely time-consuming and laborious phase of software development. An introspective survey on this topic in 2002 [2] mentions the following: "Even today, debugging remains very much of an art. Much of the computer science community has largely ignored the debugging problem ... over 50 percent of the problems resulted from the time and space chasm between symptom and root cause or inadequate debugging tools."

Let us first explain in a simple way what the authors mean by "chasm between symptom and root cause". Following is a program fragment to illustrate this issue.

```
1. void setRunningVersion(boolean runningVersion){

2.     if( runningVersion ) {
3.           savedValue = value;
         }
         else{
4.           savedValue = "";
         }

5       this.runningVersion = runningVersion;

6.      if ( savedValue == null )
```

Assume that the bug is in line 4, where the variable `savedValue` should be set to something other than an empty string. Thus, line 4 is the root cause of this particular error. However, this wrong value may be propagated elsewhere (e.g. via line 6 in the above code) and the program may make several decisions based on this wrong value. Thus, the error may be manifested much later in the program execution and in a completely different line in the program. We need automated tools to detect the *root cause* from the *manifested error*!

Organization

The rest of this article is organized as follows. In Section 2 we review the state-of-the-practice in debugging. In particular, we discuss the commercially available debugging tools. In Section 3, we review some advances in debugging methods in the past decade where the bug is localized via trace comparison based dynamic

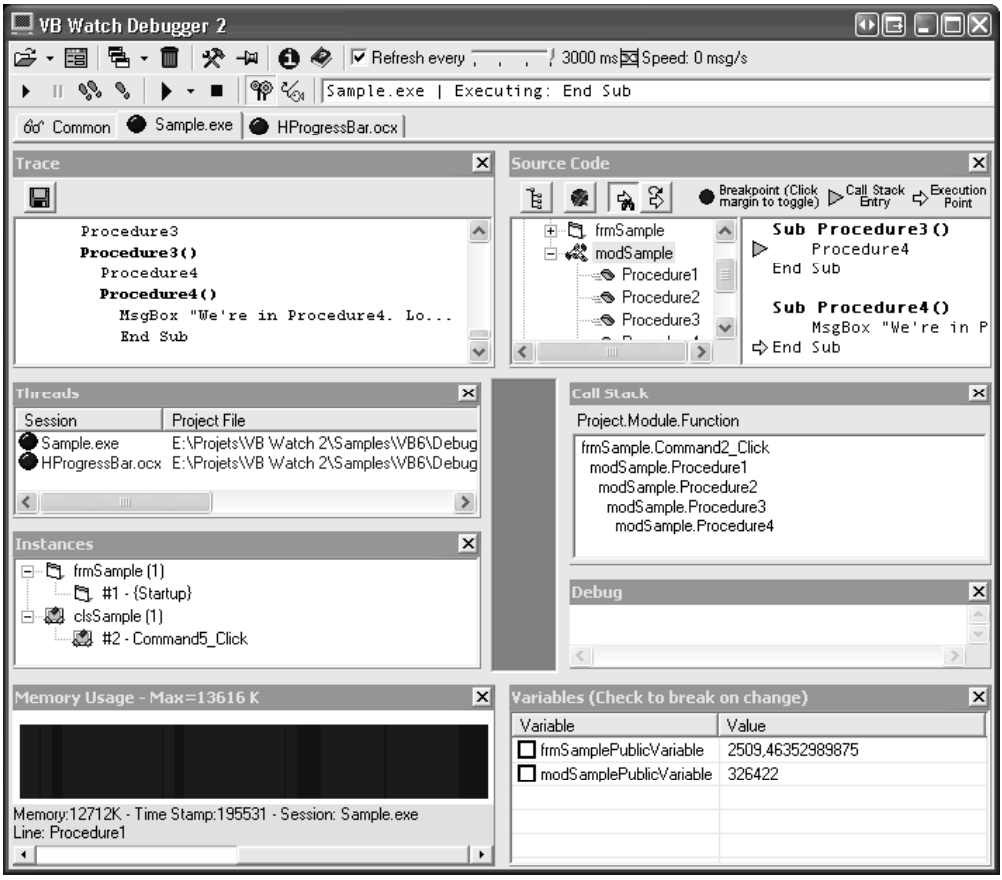

Fig. 1. Snapshots of commercially available debugging tools — VBwatch debugger for VisualBasic.

analysis. In Section 4 we explore the connections between debugging methods and formal verification techniques like model checking. In Section 5 we grapple with the realistic issue of debugging evolving programs — where the program changes from one version to another. Concluding remarks appear in Section 6.

2 Commercially Available Tools

Most of us have grown up writing programs. If we write them, assuming we are writing a substantial piece of code, we also need to debug. Now how do we debug the programs we actually write? Do we do a manual code-review? Do we test them? Do we model check them? In other words, how does a programmer today go about injecting higher reliability in the software he/she is writing? A simple answer to our question will come from the tools that go by the name of "debugging tools". We need to understand them in order to appreciate the need for systematic debugging, or the principle of debugging as a science. Our question here is more about the state-of-the-practice rather than the state-of-the-art. In our day-to-day software development activity, how do we ensure that the programs work "as desired".

Existing tools for checking program executions are extremely manual in nature. They allow a user to step through the execution sequence, set specific control locations as breakpoints and view specific program variables as breakpoints. However the programmer has to guide the entire process. These tools only allow a programmer to trace and check the execution manually. These tools do not analyze the program execution, thereby making the debugging process manual and burdensome.

The typical tools available for this purpose are command line tools such as gdb (for C) or a similar tool jdb (for Java). These command line tools simply allow the programmer to step through the execution of a test case, stopping it at places and observing variable values. The entire process in such command line tools is manual, the programmer must step through the execution, the programmer must stop the execution at desired points (by setting a breakpoint), and the programmer must decide which variables to look at in these breakpoints. Due to the short-term memory of humans, such a manual process quickly goes out of hand for large programs.

Visual Interfaces are not enough

Sometimes these tools also come with a visual interface. In Figure 1, we have shown a snapshot of the Visual Basic debugger VBWatch which monitors a program execution, allowing users to set breakpoints. It collects enough information about the execution so as to be able to print out the call stack (the procedures called), values of global variables, which program objects are being used and so on. Similar tools are also available for other programming languages *e.g.*, Visual Studio tools for Visual C++. The key point is that all these tools simply monitor and collect information about a program execution. They do not analyze the control and data flow / dependencies in an execution for understanding why a test case failed.

3 Fault Localization by Trace Comparison

Simply visualizing a trace and manually examining it is not enough. One idea which has sometimes been pursued with varying degrees of success, is to compare an execution run with other execution runs. Figure 2 depicts the working of fault localization methods. We start with a failing run (the test input which shows an unexpected output), and a pool of successful runs (possibly found via methods like coverage based testing). A successful run corresponds to a test input (and its trace) which does not show unexpected behavior. Using a difference metric (which computes a difference between runs), we choose a successful run from the successful run pool. Once again, we use the difference metric to compute the difference between the chosen successful run and the failing run. This difference is then given as the bug report.

To illustrate the method further, it is best to give an example. Our example is a fragment of the TCAS program from the Siemens benchmark suite [3] which has been extensively used in the software engineering community for research in test-

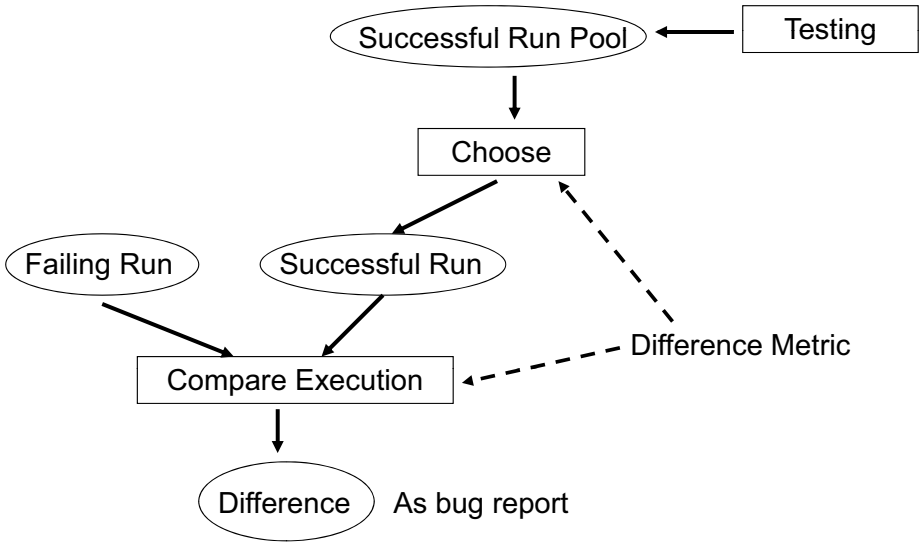

Fig. 2. Fault Localization Methods.

```
1.  if (Climb)
2.      separation = Up;
3.  else
4.      separation = Up + 100;
5.  if (separation > 150)
6.      upward = 1;
7.  else
8.      upward = 0;
9.  if (upward > 0)
10.     ...
11.     printf(‘‘Upward");
12. else
13.     ...
14.     printf(‘‘Downward");
```

Fig. 3. Example program fragment from Siemens benchmark suite

ing/debugging. The TCAS program is an embedded software for altitude control. In Figure 3, we show a fragment of the program. Note that `Climb` and `Up` are input variables of the program. There is a bug in the following program fragment, namely lines 2 and 4 are reversed in order. In other words, line 2 should be `separation = Up + 100` and line 4 should be `separation = Up`.

Now, consider an execution of the above program fragment with the inputs `Climb = 1` and `Up = 100`. The execution will lead to "Downward" being printed. Clearly, this is unexpected since the developer would expect "Upward" to be printed for these inputs. Thus, the trace for the inputs `Climb = 1`, `Up = 100` is a **failing run** which needs to be debugged.

We now have an example of a failing run, but what is a **successful run**? A successful run is simply one where the program output is as expected. So, if the programmer expects the output to be "Upward" the program should print "Upward", and if the programmer expects the output to be "Downward", the program should print "Downward". Consider the program execution with the inputs `Climb = 0` and `Up = 0`. The output in this case is "Downward" and this matches the developer's expectations. Hence we deem this as a successful run. Usually, the process of determining whether a given run is failed or successful cannot be fully automated. This involves matching the program output with the developer's expectation — so the task of articulating the developer's expectation remains manual.

We have now explained what we mean by failing run and successful run. Our task is to debug a given "failed" run – explain why it failed – that is, why the program output was not as expected. We are trying to do so by comparing it with a successful run (where the program output was as expected) in order gain insights about what went wrong in the failed run. The computed "difference" between the failed run and the chosen successful run is reported to the programmer as the **bug report**. The key questions now are:

- given a failed run, how to choose a successful run?
- given a failed and a successful run, how to compute their difference?

Both the questions have their answer in a evaluation metric for execution runs. A common (and very rough) metric is the set of statements executed in an execution run. If we have a successful run and a failed run we can compute their difference by computing the difference of the set of statements executed. The question now is how to get a successful run? In other words, how do we choose a successful run corresponding to a given failed run σ_f? We will choose a successful run σ_s such that the set of statements executed in σ_s is "close" to the set of statements executed in σ_f.

Thus, given a program P and failed execution run σ_f in P we can do the following:

- Typically the program P will be endowed with a test suite (set of test cases) based on some coverage criteria (covering all statements or all branches in the program). We construct the execution runs for the test cases from the test suite. Let this set of execution runs be $Runs_{all}(P)$.
- From among the execution runs in $Runs_{all}(P)$, we chose those which are successful, that is, runs where the program output is as per the programmer's expectations. Let this set be $SuccRuns_{all}(P)$; clearly $SuccRuns_{all}(P) \subseteq Runs_{all}(P)$.
- We choose an execution run $\sigma_s \in SuccRuns_{all}(P)$ such that the quantity $|stmt(\sigma_f) - stmt(\sigma_s)|$ is minimized. Here $stmt(\sigma)$ is the set of statements in an execution run σ and $|S|$ is the cardinality or the number of elements in a set S. Note that for two sets S_1 and S_2, the quantity $S_1 - S_2$ denotes the set difference, that is, elements appearing in S_1 but not in S_2.

Thus, we choose a successful execution run σ_s, such that there are only few state-

ments appearing in the failed run σ_f, but not in σ_s. The idea here is that if a statement appears only in the failed run but not in the successful run — it is a likely error cause.

In our running example, the inputs `Climb = 1` and `Up = 100` lead to an unexpected output. The set of statements executed in this failed execution run is $\{1, 2, 5, 7, 8, 9, 13, 14\}$. Furthermore, the inputs `Climb = 0` and `Up = 0` lead to an expected output. The set of statements executed in this successful execution run is $\{1, 3, 4, 5, 7, 8, 9, 13, 14\}$. So, the bug report is the difference between these two sets

$$\{1, 2, 5, 7, 8, 9, 13, 14\} - \{1, 3, 4, 5, 7, 8, 9, 13, 14\} = \{2\}$$

Once this line is pointed out to the developer, hopefully he/she will be able to locate the error in line 2.

Note here that the choice of successful execution run is crucial. Consider an execution of the program in Figure 3 with the inputs `Climb = 1` and `Up = 200`. When executed with these inputs, the program will print "Upward", which is what the developer expects. So, the execution run for these inputs is deemed as a successful run. What would have happened if we chose this execution run to compare with our failed execution run (the one resulting from the inputs `Climb = 1` and `Up = 100`). The set of statements executed for the inputs `Climb = 1` and `Up = 200` is $\{1, 2, 5, 6, 9, 10, 11\}$. So, in this case the bug report (the difference in executed statements between the failed run and the chosen successful run) would have been

$$\{1, 2, 5, 7, 8, 9, 13, 14\} - \{1, 2, 5, 6, 9, 10, 11\} = \{7, 8, 13, 14\}$$

The bug report in this case consists of more statements. Moreover, the statements do not pinpoint to the actual error cause in the program, they are only manifestations of the error cause. This simple example should demonstrate to the reader that the choice of successful run is crucial for the usefulness of the bug report generated by fault localization methods. Thus, we need systematic methods to choose a successful run corresponding to a failed execution run. Research results on this problem have been reported in [4]. More discussion on lightweight software debugging methods appear in [5].

What is needed

To establish debugging as a scientific activity, it should proceed by *analysis* — of the program and/or the failed execution trace. Even if we need to compare the failed execution trace with other traces, we should be able to derive the program inputs for these other traces via formal analysis, rather than by heuristics. The key challenge is, of course, how to build debugging methods with sound formal foundations which scale to real-life industrial software.

4 Debugging vs Model Checking

When we plan to build debugging methods with formal foundations, one obvious way is to base it on existing formal methods. Model checking and theorem proving present themselves as natural candidate methods on which a debugging technique

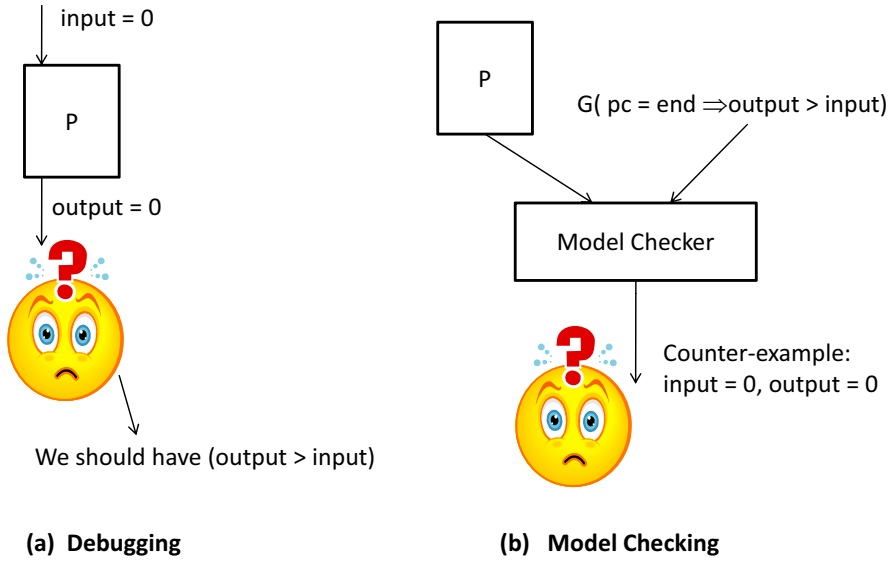

Fig. 4. Model Checking and Debugging as rather different software processes.

may be based on. Indeed, many of the existing literature loosely refer to model checking as a method for debugging! In doing so, they simply mean that model checking can be employed on software for finding latent bugs. However, from the end-user perspective, model checking and debugging are *very* different software processes! In the following, we illustrate these differences.

Figure 4 illustrates the fundamental differences between debugging and model checking. On the left-hand side of the figure, we show how debugging works. Given a program, we run test inputs against the program. If the output of the program is "as expected", we call the test a successful test, otherwise it is a failed test. In this case, suppose with `input == 0`, we obtain `output == 0` and this is not "as expected". The process of debugging is supposed to find out why we obtained an unexpected output for this particular test input. Thus, this is a situation where we have a *witness* showing the program error, but we do not know what the error is! Debugging is supposed to find out where exactly in the program the error is, which is causing the unexpected observable output.

On the right-hand side of Figure 4, we show how model checking works. Here the model checker takes in the buggy program and a temporal property, say in Linear-time Temporal Logic (LTL). The temporal property captures certain "requirements" which the program should meet. We have provided the LTL property `G (pc == end ⇒ output > input)`. Here `pc` is the program counter and `end` is the control location at the end of the program. Thus, the property is saying that at the end of the program, we should have `output > input`. If indeed the program is buggy and does not satisfy this property, the model checker produces a counter-example trace which witnesses a violation of the property being verified. The trace corresponding to `input == 0` which leads to `output == 0` is one such trace. So, in this case we

are producing the buggy trace — which we assume to have in program debugging. On the other hand, program debugging finds out which unspecified (or implicit) program requirement is violated — this is the error cause. In the case of model checking, this program requirement is made explicit by the user providing it as a temporal property.

Several papers on software model checking often refer to the technique as a tool for finding program bugs, and hence model checking is also often called as a debugging method. However, model checking is a static checking method which checks code. On the other hand, conventional debugging is a dynamic checking method which analyzes an execution trace and explains an observable error. Moreover, as pointed out in the preceding, debugging takes in an "erroneous" execution trace, where the programmer is trying to find out what the "error" is. Model checking on the other hand takes in a formal definition of what is an "erroneous" execution trace (any trace violating the given LTL property say), and searches for an erroneous execution trace.

5 Debugging of Evolving Programs

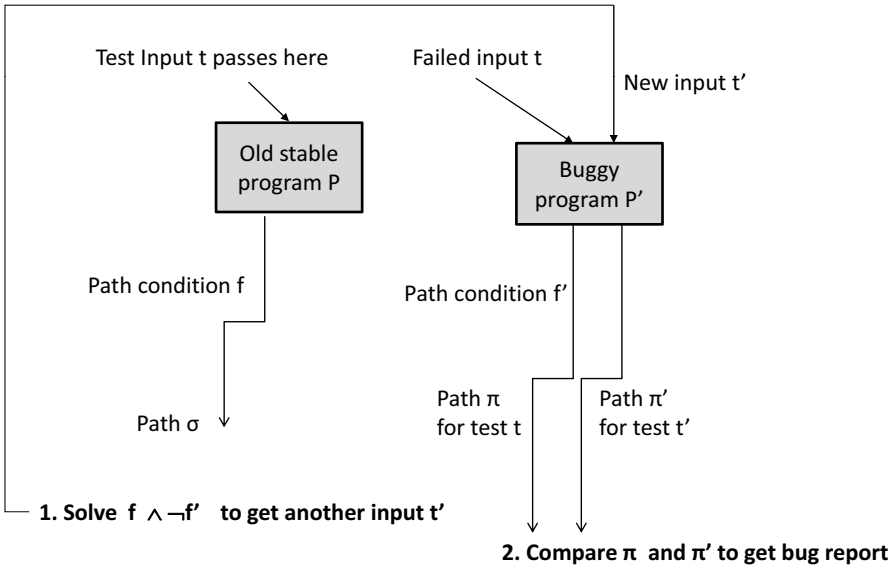

Fig. 5. Pictorial description of debugging method for evolving programs [6]

We now move on to the task of debugging various versions of a program. In doing so, we recognize the widely accepted reality in any large-scale software development — a complex piece of software is never written from scratch. Usually a program evolves from one version to another. When we change a program version to produce a new version, we may introduce "bugs". Thus, the problem formulation involves two program versions — an old stable program P, and a new buggy program P'. We are debugging the behavior of a test input t which forms a legal input of both the program versions. Moreover, test input t passes in P and fails in P'. In other

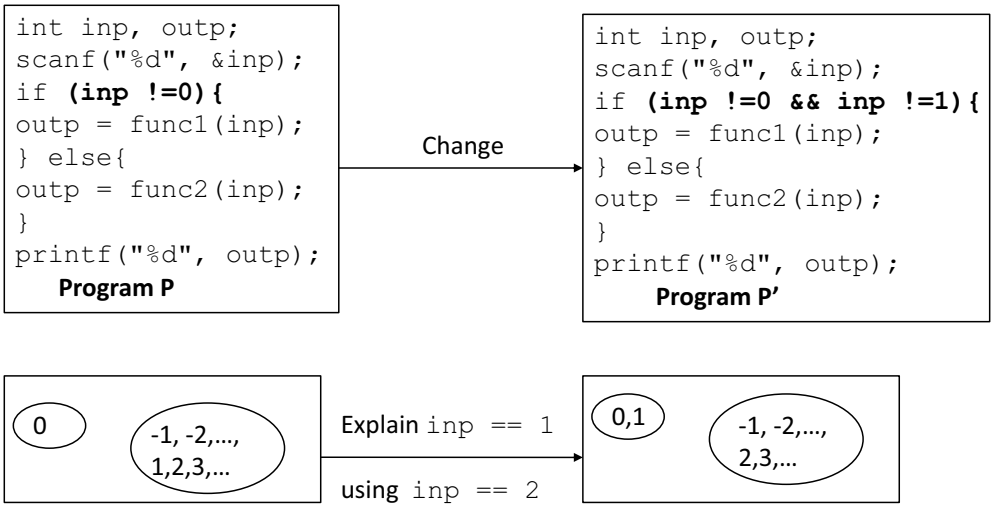

Fig. 6. Two example programs P, P' and their input space partitioning. The behavior of the input 1 changes during the change $P \rightarrow P'$. We choose an input 2 to explain the behavior of the failing input 1 — since $1, 2$ are in the same partition in P, but different partitions in P'.

words, the behavior of test input t is as expected in program P, while it shows unexpected behavior in program P'. The unexpected behavior can be in the form of an unexpected output, or a program crash.

Given such a debugging task, we can solve it as follows. We perform concrete as well as symbolic execution of test input t on programs P and P'. By performing symbolic execution, we can gather the **path condition** — a formula capturing the set of inputs which follow the same program path as that of t. Let the path conditions of t in programs P, P' be f and f' respectively.

Recall that, a solution to the formula f denotes an input which follows the same path as t in program P. Similarly, a solution to the formula f' denotes an input which follows the same paths as t in program P'. We now find a solution to $f \wedge \neg f'$. Let t' be such a solution. What can we say about t'? We know the following facts:

- t' follows the same path as that of t in the old stable program P.
- t' and t follow different program paths in the new buggy program P'.

Assuming the old stable program P to be "correct", we can therefore infer that the behavior of t and t' are intended to be "similar" — to the extent that they follow the same program path in P. However, the behavior of t and t' differ in the buggy program P'. Thus, by comparing the traces of t and t' in P' we can hope to localize the error cause.

A pictorial description of the debugging method appears in Figure 5. To show the working of the method, we present an example (similar to [6]). Consider a program fragment with an integer input variable `inp` – the program P in Figure 6. This is the old program version. Note that `func1`, `func2` are functions invoked from P. The code for `func1`, `func2` is not essential to understanding the example, and

hence is not given. Suppose the program P is slightly changed to the program P' in Figure 6, 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 == 1` 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. The debugging method works as follows.

(i) We run program P for test input `inp == 1`, and calculate the resultant *path condition* f, a formula representing set of inputs which exercise the same path as that of `inp == 1` in program P. In our example, path condition f is $inp \neq 0$.

(ii) We also run program P' for test input `inp == 1`, and calculate the resultant *path condition* f', a formula representing set of inputs which exercise the same path as that of `inp == 1` in program P'. In our example, path condition f' is $\neg(inp \neq 0 \wedge inp \neq 1)$.

(iii) 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 input `inp == 1` in the old program P, but follows a different path than that of the test input `inp == 1` in the new program P'. In our example $f \wedge \neg f'$ is

$$inp \neq 0 \wedge (inp \neq 0 \wedge inp \neq 1)$$

A solution to this formula is any value of `inp` other than 0,1 (say `inp` == 2).

(iv) Finally, we compare the trace of the test input being debugged (`inp == 1`) in program P', with the trace of the test input that was generated by solving path conditions (here `inp == 2`). By comparing the trace of `inp == 1` with the trace of `inp == 2` in program P' we find that they differ in the evaluation of the branch `inp !=0 && inp !=1`. Hence this branch is highlighted as the *bug report* — the reason for the test input `inp == 1` failing in program P'.

We note that for solving the formula $f \wedge \neg f'$, we take the help of state-of-the-art Satisfiability Modulo Theory (SMT) solvers (such as Z3 [7] and STP [8] which have built-in theories for bitvectors and arrays). SMT solvers can be used to decide the satisfiability of quantifier-free first order logic formula. All path conditions are formulae of this kind - universal quantification is not present, and any variable appearing in the formula is implicitly existentially quantified. We leverage on the immense progress in the theory and practice of SMT solvers in the past two decades in our program debugging method.

Details of debugging methods for evolving programs appears in [6]. This approach can not only debug program versions, but also two completely different implementations of the same protocol. Thus, our solution can also be used to debug errors in the situation where P, P' are two completely different implementations (of the same protocol specification), rather than being two versions of the same program. This feature of our method is shown in [6], which reports experiments from debugging different web-servers (all of which implement the well-known HTTP protocol).

6 Concluding Remarks

In this article, we have taken a fresh look at software debugging, a task which is crucial for reliable software development. We have reviewed the state-of-the-practice in software debugging, first by studying the kind of commercial tools available in the marketplace. We find that these tools are focused on execution visualization with little focus on execution analysis. We have then discussed lightweight execution trace analysis methods which often go by the name of fault localization. The success of such methods depend on the quality of available test data. Finally, we present the problem of debugging evolving programs, a common situation in any software development where a program evolves from one version to another. This is a huge problem in any large-scale software development — for example, consider the evolution of Microsoft Windows operating system from one version (say Vista) to another (say Win7).

Our advocated method for debugging evolving programs is built on symbolic execution and Satisfiability Modulo Theory (SMT) formula solving. With the recent progress in the scalability of SMT solvers, analysis methods based on symbolic execution and formula solving have become feasible in practice. Such methods are based on formal foundations, which most program debugging methods lack. Even though symbolic execution based test generation has been studied recently [9], the utility of symbolic execution for program debugging has not been tapped (apart from the most recent work [6]). The software engineering research community needs to focus on these directions, building debugging methods with formal foundations which scale up to large-scale industrial software. Symbolic dynamic analysis and SMT formula solving are likely to be key ingredients of such debugging methods.

Acknowledgments

This work was partially supported by a Defense Innovative Research Programme (DIRP) grant from Defense Science and Technology Agency (DSTA), Singapore.

References

[1] National Institute of Standards & Technology. The economic impacts of inadequate infrastructure for software testing, 2002. http://www.nist.gov/director/prog-ofc/report02-3.pdf.

[2] B. Hailpern and P. Santhanam. Software debugging, testing and verification. *IBM Systems Journal*, 41(1), 2002.

[3] M. Hutchins et al. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, 1994.

[4] L. Guo, A. Roychoudhury, and T. Wang. Accurately choosing execution runs for software fault localization. In *International Conference on Compiler Construction (CC)*, 2006.

[5] A. Roychoudhury. *Embedded Systems and Software Validation*. Morgan Kaufmann, 2009.

[6] D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani. DARWIN: An approach for Debugging Evolving Programs. In *Joint meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2009.

[7] L. de Moura and N. Bjorner. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2008.

[8] V. Ganesh and D. Dill. A decision procedure for bit-vectors and arrays. In *International Conference on Computer Aided Verification (CAV)*, 2007.

[9] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *International Conference on Programming Language Design and Implementation (PLDI)*, 2005.