Information and Software Technology 58 (2015) 403–418

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Infeasible path generalization in dynamic symbolic execution

Mickaël Delahaye [a], Bernard Botella [a], Arnaud Gotlieb [b,*]

[a] CEA LIST, Software Safety Laboratory, PC 174, 91191 Gif-sur-Yvette Cedex, France
[b] SIMULA Research Laboratory, Certus Software V&V Center, Lysaker, Norway

ARTICLE INFO

Article history:
Received 12 December 2013
Received in revised form 22 July 2014
Accepted 22 July 2014
Available online 1 August 2014

Keywords:
Dynamic symbolic execution
Explanation
Test input generation

ABSTRACT

Context: Automatic code-based test input generation aims at generating a test suite ensuring good code coverage. Dynamic Symbolic Execution (DSE) recently emerged as a strong code-based testing technique to increase coverage by solving path conditions with a combination of symbolic constraint solving and concrete executions.
Objective: When selecting paths in DSE for generating test inputs, some paths are actually detected as being infeasible, meaning that no input can be found to exercize them. But, showing path infeasibility instead of generating test inputs is costly and most effort could be saved in DSE by reusing path infeasibility information.
Method: In this paper, we propose a method that takes opportunity of the detection of a single infeasible path to generalize to a possibly infinite family of infeasible paths. The method first extracts an explanation of path condition, that is, the reason of the path infeasibility. Then, it determines conditions, using data dependency information, that paths must respect to exhibit the same infeasibility. Finally, it constructs an automaton matching the generalized infeasible paths.
Results: We implemented our method in a prototype tool called IPEG (Infeasible Path Explanation and Generalization), for DSE of C programs. First experimental results obtained with IPEG show that our approach can save considerable effort in DSE, when generating test inputs for increasing code coverage.
Conclusion: Infeasible path generalization allows test generation to know of numerous infeasible paths ahead of time, and consequently to save the time needed to show their infeasibility.

1. Introduction

Software testing is an essential part of today's software engineering, as the principal and often the only mean to ensure software reliability. Among the techniques that permit one to improve the quality of a test set, code-based testing, also known as white-box testing, plays an important role. Code-based testing implies the usage of the source code to select test inputs, to measure code coverage, to localize faults and eventually to propose automatically bug repairs. These last years, code-based testing has become more and more appealing with the emergence of new techniques and powerful tools. However, modern effective code-based testing has also a main limitation: high code coverage is often difficult or costly to reach, without compromising the efficiency of the technique. This paper is concerned with this challenge and describes a new cross-cutting technique that contributes to handle this issue.

As said above, the field of code-based testing has seen the emergence of new techniques and powerful tools, most of them being based on Dynamic Symbolic Execution (e.g., PathCrawler [31], DART [12], CUTE [27], SAGE [13] or PEX [28] just to name the pioneering tools). Dynamic Symbolic Execution (DSE) is a software testing and analysis technique which starts by selecting and executing a (feasible) path, by picking up a test input at random. Then, it computes a path condition by symbolically evaluating the instructions along the activated path. By refuting one decision of that path and exploiting a constraint solver, DSE determines a new test input which, by construction, covers another path in the program under test. Said otherwise, DSE tries to uncover test inputs which cover distinct paths that the ones that already covered, in order to increase path coverage. An important observation concerns path selection: when a path is activated by a test input, it is necessarily feasible but when a path is selected by refuting one decision over a (feasible) path, then its feasibility is no more guaranteed. Whenever DSE considers an infeasible path, then the constraint solver tries to prove the unsatisfiability of the path condition. Obviously, this task is not formally required and

* Corresponding author.
E-mail addresses: mickael.delahaye@cea.fr (M. Delahaye), bernard.botella@cea.fr (B. Botella), arnaud@simula.fr (A. Gotlieb).

http://dx.doi.org/10.1016/j.infsof.2014.07.012
0950-5849/

```
/* Let x be the input,
   and res the output        */
[abs := x; i := 2]^a;
[res := 1]^b;
if [abs < 0]^c then
  [abs := −abs]^d;
while [i ≤ abs]^e do
  [res := res × i; i := i+1]^f;
if [x < 1]^g then
  [res := res + 5]^h;
```

Fig. 1. A program with many infeasible paths.

corresponds to a waste of time because the goal of DSE is to find new test inputs, and not to report path infeasibility. Even if detecting all the infeasible paths is impossible,[1] studies have shown that they are ubiquitous in computer programs [33] and that avoiding them when testing programs is highly desirable [22].

As a simple motivating example, please consider the program of Fig. 1. On this program, a typical DSE tool might stumble onto a lot of infeasible paths, as shown on the run given in Fig. 2. In step (1), an input is arbitrarily chosen (e.g., $x = 2$), the program is executed, and the activated path is traced ($a\ b\ c^{\mathsf{f}}\ e^{\mathsf{t}}\ f\ e^{\mathsf{f}}\ g^{\mathsf{f}}$), as shown on the figure. In step (2), the tool chooses a new path to cover (indicated with dotted arrows) based on the current path using a depth-first strategy. The tool computes a path condition for this new path (indicated beside the path), and passes it on to a solver. The solver answers negatively (indicated by an X). Indeed, the path condition is inconsistent ($2 \leqslant x$ contradicts $x < 1$). In other words, the generation has met its first infeasible path $a\ b\ c^{\mathsf{f}}\ e^{\mathsf{t}}\ f\ e^{\mathsf{f}}\ g^{\mathsf{t}}\ h$. In (3), the tool tries to activate another path with the same method. This time, the solver gives a solution to the path condition. This solution ($x = 3$) is used as input to a concrete execution of the program to get a full activated path. This path iterates the loop one more time. In (4), the tool tries to activate the statement h, and for the very same reason as step (2), the attempt fails. And so on and so forth, going deeper and deeper in the loop. Hopefully, the tool bounces back either on an arbitrary limitation of the path length or the maximal value of the data type (only for finite data type). After covering a first path with no iteration in the loop, this hypothetical tool indeed finds an input that activates the statement h. During the test input generation, a lot of paths are proved infeasible. Indeed, a manual code review lets us confirm that, if the control flow passes through the "then" branch of the first conditional and through at least one iteration in the loop, it cannot go into the "then" branch of the second conditional. This family can be represented by an automata given of Fig. 3. Every path for which a prefix is recognized by the automata is necessarily infeasible.

Though one can argue another search strategy might perform better on this particular example, such traps exist for every strategy. Moreover, real programs do contain families of similar infeasible paths. Recommended programming practices, such as code reuse, modularity, and assertions, are often source of possibly redundant checks leading to numerous infeasible paths. As we will see later, even well known algorithms possess such families of infeasible paths.

Motivated by such cases, we propose in this paper a technique that allows test input generators to detect early and to skip numerous infeasible paths. This technique takes opportunity of the detection of an infeasible path by the test input generator to generalize to a possibly infinite family of infeasible paths. The method consists first of extracting the "essence" of the infeasibility from the path condition. Then, by combining data dependency information and finite state automaton operations, our approach constructs an infeasible path automaton, a representation of an infeasible path family of the program. Finally, this automaton can be used to detect paths belonging to the family for the cost of matching a regular expression.

To evaluate our approach, we developed a modular tool called IPEG (Infeasible Path Explanation and Generalization) for programs in C. It can be parametrized by any solving procedure. For our experiments, we used three constraint solvers (i.e., Colibri, Yices and Z3) that are currently used in dynamic symbolic execution tools. We evaluated our approach at two levels. First, the unitary evaluation checks the effectiveness of the generalization method to prove path infeasibility against an exhaustive symbolic execution. Second, an integrated evaluation checks how a naive integration of the method in a test input generator affects the performances. These experiments show that our approach can save considerable computation time during test generation.

Paper organization. Section 2 gives essential notations and background notions to understand the infeasible path generalization method. Section 3 presents the method in depth with a number of examples. Notions such as data dependencies and infeasible path automaton are introduced in this section. Section 4 discusses the integration of the proposed method within a dynamic symbolic execution procedure. Section 5 contains the results of our experimental evaluation of method. Section 6 positions our proposed method into state-of-the-art path infeasibility analyses. Finally, Section 7 concludes the paper and draws a couple of perspectives to this work.

2. Background and notations

This section first defines some notions and notations about programs, paths and feasibility. Then, it introduces and reviews the notion of constraint-based explanations.

2.1. Program and path

For the sake of clarity, we will use a simple imperative language for representing programs. Fig. 1 gives a concrete example of the syntax used in the paper. It is important to note that simple statements (assignment or **skip** statement) and tests (that is, conditions on loops and conditional constructs) are labeled.

A *program path* is a sequence of program statements allowed by the flow relation defined on the studied language. In the paper, a path is noted by a sequence of augmented labels on a particular program. An *augmented label* is a label possibly followed by a letter, t or f, to explicit the truth value, respectively true or false, when the label points to a test. If x is an augmented label, $\mathsf{label}(x)$ denotes the simple label (without any letter).

For instance, on the program of Fig. 1, $a\ b^{\mathsf{t}}\ c\ e^{\mathsf{f}}\ g^{\mathsf{t}}\ h$ is a path that does not enter the loop and goes through the "then" branches of the two conditionals. Note however that the sequence $a\ e^{\mathsf{t}}\ i$ does not respect the program's control flow and as such is not to be considered a program path.

2.2. Feasibility and path condition

A program path is said to be *feasible* if there is at least one particular input that activates it. Conversely, a path is *infeasible* if there is no input that activates it.

A *path condition* of a program path given a symbolic input vector is a conjunction of constraints on the symbolic input vector

[1] This problem was proved undecidable in general [30].