Combined Constraint-Based Analysis for Efficient Software Regression Detection in Evolving Programs

Anh D. Le, Tho T. Quan, Nguyen T. Huynh, Phung H. Nguyen, and Nhat-Van Le

Faculty of Computer Science and Engineering, Hochiminh City University of Technology, Hochiminh City, Vietnam
tintinkool@gmail.com, {qttho,htnguyen,phung}@cse.hcmut.edu.vn, ccnhatvan@yahoo.com

Abstract. Software regression is a bug that makes software stop functioning normally after a certain event. In this paper, we investigate detecting regression when software evolves to a new version. In this context, regression bugs occur in parts of software that already passed testing process in the old version. Hence, such kind of bugs is difficult to be discovered if normal strategy like white-box testing is applied. Moreover, since both old and new versions must be taken into account during the testing process, the computational cost is usually high in this con-text.

Concolic testing in an emerging extension of white-box testing that can reduce significantly the number of execution paths needed to be analyzed. However, the typical concolic testing is not really efficient when dealing with software regression. Thus, we propose a new approach based on combined constraint to solve this problem, known as CTGE (Efficient Constraint-based Test-cases Generation) approach. The soundness of our theoretical contribution is formally proved and supported by some initial experiments conducted in education environment.

Keywords: Constraint-based Test-case Generation, Regression Bugs, Evolving Programs Debugging.

1 Introduction

A software regression is a bug which makes a feature stop functioning as intended after a certain event. In practice, this kind of bug involves in many contexts like testing documentation [8] or testing of component-based software [9].

In this paper, we investigate a situation where regression bugs regularly occur. That is, when a program is evolved into a new version to meet new requirements or just to refine the code, chances are that the new evolved program may accidentally violate the original requirements.

So far, software practitioners have still been commonly using testing techniques to detect program bugs. Traditionally, a set of test-cases will be generated for testing. Basically, a test-case is a set of inputs, execution conditions and desired outputs which can be tested by the system when functioning accordingly using some test
procedures and test scripts. However, the generation of test-case is usually costly and thus requiring a systematic method. White-box testing (or structural testing) technique, based on flow-control analysis, is typically applied in this case [6]. In this method, the tested program is analyzed in terms of control flow and data flow. Thus, the test-cases are generated accordingly to (1) exercise independent paths within a module or unit; (2) exercise logical decisions on both their true and false side; (3) execute loops at their boundaries and within their operational bounds; and (4) exercise internal data structures to ensure their validity [10].

The main advantage of white-box testing is that it does not only detect the bugs, but also help locate the piece of code that causes the problem. However, the primary disadvantage of this method is the suffering of high execution cost. If we have a program which has 10 independent if...then...else... statements, there are totally 210 execution paths needed to be explored. When dealing with evolving programs, this problem is still even more crucial, since we must do the combined analysis on both old and new versions.

There are many attempts which have been made to efficiently explore program paths for test-case generation purpose, in which the infeasible combined paths are eliminated [3]. Nowadays, the most prominent testing method based on the white-box approach technique is concolic testing approach. In this approach, apart from concrete execution of the tested program based on certain input, symbolic execution [4] is also involved to resolve the path constraints by means of theorem provers. Hence, only the feasible paths are considered for generating appropriate test-case, thus significantly reducing the numbers of test-case needed for path coverage. This technique is then adopted and exploited remarkably in various testing tools like PathCrawler [16], jCUTE [13] and SAGE [5]. However, DASH [1] is perhaps the most efficient concolic-based technique which uses abstraction to deal with real complex practical programs.

Unfortunately, the typical white-box testing and its concolic-based variations are not able to fully detect this regression in evolving programs, even when the flow analysis is performed on both original and evolved versions. To make it clearer, in the following discussion in Section 2, we will give some motivating examples on this issue. To overcome this problem, we suggest an approach of using constraints combined from path conditions of both original and evolved program versions. We then formally prove the soundness of this approach, under the context of well-conditioned programs. To make this approach practical, we also propose an algorithm, known as CTGE (Efficient Constraint-based Test-case Generation) that reduces the cost of test-case generation from exponential complexity to linear one.

The rest of the paper is organized as follows. Section 2 presents a motivating example which shows that when a program evolves, neither test-cases generated merely from the old version nor the new version are sufficient to detect regression bug. Section 3 discusses our proposed approach on generating test-case by combining execution paths from the previous version and the evolved version of a program into constraints. In Section 4, we introduce the ultimate CTGE algorithm, an improvement of the test-case generation algorithm to reduce its complexity significantly. Section 5 gives some experiments. Finally, Section 6 concludes the paper.