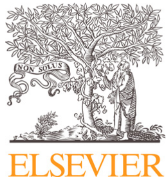
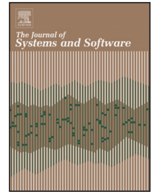
The Journal of Systems and Software 000 (2016) 1–17



Contents lists available at ScienceDirect

The Journal of Systems and Software

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



An empirical study to quantify the characteristics of Java programs that may influence symbolic execution from a unit testing perspective

Marcelo M. Eler[a,*], Andre T. Endo[b], Vinicius H.S. Durelli[c,d]

[a] Escola de Artes, Ciências e Humanidades Universidade de São Paulo (USP), São Paulo, Brazil
[b] Universidade Tecnológica Federal do Paraná (UTFPR), Cornélio Procópio, Brazil
[c] University of Groningen, Groningen, The Netherlands
[d] Faculdade Campo Limpo Paulista, So Paulo, Brazil

ARTICLE INFO

Article history:
Received 23 January 2015
Revised 12 September 2015
Accepted 7 March 2016
Available online xxx

Keywords:
Software testing
Symbolic execution
Test data generation

ABSTRACT

In software testing, a program is executed in hopes of revealing faults. Over the years, specific testing criteria have been proposed to help testers to devise test cases that cover the most relevant faulty scenarios. Symbolic execution has been used as an effective way of automatically generating test data that meet those criteria. Although this technique has been used for over three decades, several challenges remain and there is a lack of research on how often they appear in real-world applications. In this paper, we analyzed two samples of open source Java projects in order to understand the characteristics that may hinder the generation of unit test data using symbolic execution. The first sample, named SF100, is a third party corpus of classes obtained from 100 projects hosted by SourceForge. The second sample, called R47, is a set of 47 well-known and mature projects we selected from different repositories. Both samples are compared with respect to four dimensions that influence symbolic execution: path explosion, constraint complexity, dependency, and exception-dependent paths. The results provide valuable insight into how researchers and practitioners can tailor symbolic execution techniques and tools to better suit the needs of different Java applications.

© 2016 Elsevier Inc. All rights reserved.

1. Introduction

Software testing is a key quality assurance activity and also one of the most costly activities of the whole software development process. During this activity, testers run the program under test with the goal of uncovering faults (Myers et al., 2004). Ideally, the program under test should be run against all possible inputs and scenarios. However, this is impractical or even infeasible due to the large size of the input domain of most programs.

Given such a limitation, instead of using the entire input domain, testers resort to testing techniques in order to decide what inputs are more likely to uncover different types of faults. Testing techniques help testers to define test scenarios and input data based on a small but significant subset of the input domain. Each testing technique has specific criteria to cover a particular aspect of the program and each criterion defines different test requirements that should be met by test cases.

Functional and structural testing are two of the most commonly used testing techniques. The main goal of the former is to exercise all functions of the program under test, whereas the purpose of the latter is to cover certain structures such as paths, instructions, and branches. Structural testing criteria yield a large number of test requirements, which implies that manually generating test data to satisfy these test requirements is time consuming, error prone, and unwieldy. Consequently, several approaches have been proposed to automate the generation of test data that satisfy structural testing criteria (Galler and Aichernig, 2014).

Symbolic execution and constraint solving have been used as effective techniques to generate test data for structural testing (Ramamoorthy et al., 1976; King, 1976; Cadar and Sen, 2013; Eler et al., 2014a). In general, symbolic execution lies in representing the program elements (usually local variables and attributes) as functions of symbolic input values (King, 1976; Cadar and Sen, 2013). Then, each execution path in the program is executed based only on the symbolic input data. The symbolic execution of a path is a set of constraints that should be satisfied so that the path can be executed. The resultant set of constraints associated to each execution path is a path constraint. Finally, each path constraint is sent to a constraint solver, which in turn generates, if possible, concrete input values (i.e., test data) that satisfy the constraints.

The idea of generating test data using symbolic execution dates back from more than three decades ago. However, while the idea

* Corresponding author. Tel.: +55 11964746506.
E-mail addresses: marceloeler@usp.br, marceloeler@gmail.com (M.M. Eler), andreendo@utfpr.edu.br (A.T. Endo), durelli@icmc.usp.br (V.H.S. Durelli).

http://dx.doi.org/10.1016/j.jss.2016.03.020
0164-1212/© 2016 Elsevier Inc. All rights reserved.

is appealing in principle, symbolic execution raises a number of research challenges. Some of these challenges have not been completely overcome yet. *Path explosion* is one of them: symbolically executing a large number of paths entails high computational overhead (Anand et al., 2013). Moreover, long paths tend to yield large path constraints, which can hurt performance (Cadar and Sen, 2013). *Constraint complexity* is also an issue given that data types and the complexity of arithmetic expressions may affect the efficiency and precision of constraint solvers (Pasareanu and Visser, 2009; Cadar et al., 2011). Approaches also have to deal with *dependency*. Several constraints are related to method calls, whose values may not depend on symbolic input values (Anand et al., 2013). In addition, there are many *exception-dependent paths* that can only be executed when a given exception is thrown. In such cases, constraints to raise a given exception may not be explicitly declared in the code, hampering the coverage of specific exception paths.

If not handled properly, the aforementioned issues can jeopardize the test generation process. Although there has been some research on analyzing the characteristics of programs that have an impact on symbolic execution (Qu and Robinson, 2011; Xiao et al., 2013), no large scale study has been conducted with real-world applications.

In this paper, we set out to investigate the nature and the frequency of the aforementioned issues related to symbolic execution. Specifically, we analyzed the following factors from a unit testing perspective: *(i)* the distributions of loops ad nested loops, which cause *path explosion; (ii)* data types, path constraint size, nonlinear expressions, and contradictory constraints, which contribute to *constraint complexity; (iii)* method calls, which represent *dependencies*; and *(iv)* constraints with exception declarations, which indicate *exception-dependent paths*. Towards this end, we developed a tool to perform symbolic execution on real-world Java programs and collect metrics related to these factors. Two benchmarks composed of real-world open source Java programs were adopted: *(i)* SF100 is a corpus of classes extracted from 100 open source programs described in Fraser and Arcuri (2012), and *(ii)* R47 is a set of well-known and established projects we selected from different repositories (e.g., ASF and GitHub), some of the programs in this set are described by Durelli et al. (2016). In total, we analyzed 219,248 methods with at least one branch, 34,493 methods from SF100 and 184,755 methods from R47.

Preliminary results of this research considering only SF100 were published in Eler et al. (2014b). The extensions of this work are essentially twofold. First, we included a sizable corpus of classes (viz., R47) to the analysis, extending the results we obtained from SF100. We then contrasted the results from both benchmarks. Second, other factors that may influence *constraint complexity* were considered. Nonlinear expressions in path constraints were analyzed and we observed how the search for contradictory constraints helps to eliminate unsolvable path constraints.

By analyzing the two benchmarks, we concluded that: *(i)* methods with potential to cause *path explosion* due to the presence of loops represent around 25% of the analyzed methods, while methods with at least one nested loop represent nearly 6.5% of the methods; *(ii) constraint complexity* is influenced by the occurrence of complex types (e.g., objects) present in 65% to 73% of methods, while floating-point types and nonlinear expressions are rare; *(iii) dependency* is also a major issue since 40% of the methods require an external library (i.e., outside of the project scope); and *(iv) exception-dependent* paths also pose a challenge to test data generation using symbolic execution since around one third of the methods deal with exceptions.

These results provide valuable insight into how to use and evaluate the adequacy of modern-day approaches for symbolic execution in software testing. Researchers and practitioners can draw from the key results of this investigation to tailor symbolic execution techniques and tools to better suit the needs of different Java applications.

The remainder of this paper is organized as follows. Section 2 covers background and issues related to test data generation employing symbolic execution. Section 3 describes the empirical study and outlines the data extraction procedure. Section 4 presents the analysis of the results for the two benchmarks. Section 5 discusses the results. Section 6 synthesizes the lessons learned and recommendations for researchers and practitioners. Section 7 summarizes related work. Section 8 presents concluding remarks and outlines future work.

2. Background

Since the seminal work of King in 1976 (King, 1976), symbolic execution has been used for more than three decades to generate test data for achieving high coverage during structural testing (Ramamoorthy et al., 1976; Cadar and Sen, 2013). The general process is about the same for most of the approaches: the program under test is symbolically executed using symbolic input values and path constraints associated with each execution path are identified. Each path constraint is then sent to a constraint solver that, if possible, finds solutions to satisfy all constraints.

A path constraint is a logical expression connecting all constraints that should be satisfied in order to execute a particular path. Typically, path constraints are made up of combinations of four elements: *(i)* variables, which can be local, instance, or class variables; *(ii)* method calls, which can be either calls to methods of the same class, methods of other classes, or methods of other projects or libraries; *(iii)* constants; and *(iv)* exception declarations, which are indications that exceptions must be thrown to execute certain paths.

Recently, symbolic execution has been receiving renewed attention due to better constraint solvers and the introduction of hybrid approaches (e.g., concolic testing (Sen, 2007)). Nevertheless, symbolic execution still poses several challenges when applied to software testing (Pasareanu and Visser, 2009; Godefroid, 2012; Cadar and Sen, 2013; Anand et al., 2013; Xiao et al., 2013): *(i)* path explosion, *(ii)* complexity of constraints, *(iii)* dependency, and *(iv)* paths triggered by exceptions. The next subsections elaborate on the nature of these four issues.

2.1. Path explosion

Path explosion is one of the key challenges (Anand et al., 2013; Xiao et al., 2013). A workaround to this issue is to use algorithms that lead to paths that cover all branches by going through loops only once. Yet, the number of constraints to be solved in the resulting paths tends to be large. This can overwhelm the constraint solver and hurt performance (Cadar and Sen, 2013). In addition, some branches may be covered only when paths including more than one loop iteration are taken into account. In such cases, methods with several nested loops tend to generate a huge number of paths. As a result, performance is negatively affected because the constraint solver has to go over a multitude of path constraints up to the point that a solvable path constraint is identified.

2.2. Constraint complexity

The complexity of a constraint may be related to the data types of their elements or the complexity of the arithmetic expressions. Symbolic execution approaches can better handle constraints with primitive types (e.g., fixed-point data types) than constraints sequences containing complex types (e.g., arrays and objects) (Pasareanu and Visser, 2009). While the initial approaches only

Download English Version:

https://daneshyari.com/en/article/6885455

Download Persian Version:

https://daneshyari.com/article/6885455

Daneshyari.com