International Journal of Information Management 36 (2016) 963–975

Contents lists available at ScienceDirect

International Journal of Information Management

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

A survey on Test Suite Reduction frameworks and tools

Saif Ur Rehman Khan [a,**], Sai Peck Lee [a,*], Raja Wasim Ahmad [b], Adnan Akhunzada [b], Victor Chang [c]

[a] Department of Software Engineering, Faculty of Computer Science and Information Technology, University of Malaya, 50603 Kuala Lumpur, Malaysia
[b] Department of Computer Science, COMSATS Institute of Information Technology (CIIT), Pakistan
[c] International Business School Suzhou, Xi'an Jiaotong Liverpool University, Suzhou, China

ARTICLE INFO

Article history:
Received 28 May 2016
Accepted 28 May 2016

Keywords:
Regression testing
Test suite optimization
Test Suite Reduction
Frameworks
Fault localization

ABSTRACT

Software testing is a widely accepted practice that ensures the quality of a System under Test (SUT). However, the gradual increase of the test suite size demands high portion of testing budget and time. Test Suite Reduction (TSR) is considered a potential approach to deal with the test suite size problem. Moreover, a complete automation support is highly recommended for software testing to adequately meet the challenges of a resource constrained testing environment. Several TSR frameworks and tools have been proposed to efficiently address the test-suite size problem. The main objective of the paper is to comprehensively review the state-of-the-art TSR frameworks to highlights their strengths and weaknesses. Furthermore, the paper focuses on devising a detailed thematic taxonomy to classify existing literature that helps in understanding the underlying issues and proof of concept. Moreover, the paper investigates critical aspects and related features of TSR frameworks and tools based on a set of defined parameters. We also rigorously elaborated various testing domains and approaches followed by the extant TSR frameworks. The results reveal that majority of TSR frameworks focused on randomized unit testing, and a considerable number of frameworks lacks in supporting multi-objective optimization problems. Moreover, there is no generalized framework, effective for testing applications developed in any programming domain. Conversely, Integer Linear Programming (ILP) based TSR frameworks provide an optimal solution for multi-objective optimization problems and improve execution time by running multiple ILP in parallel. The study concludes with new insights and provides an unbiased view of the state-of-the-art TSR frameworks. Finally, we present potential research issues for further investigation to anticipate efficient TSR frameworks.

1. Introduction

Software testing ensures the quality and reliability of a System Under Test (SUT) by revealing maximum possible defects (Myers, Sandler, & Badgett, 2011). However, software testing is the most expensive quality assurance practice since it consumes up to 50% of the total software development cost (Ramler & Wolfmaier, 2006). Although, exhaustive testing (e.g., running all possible paths in the SUT (Lee & Chung, 2000)) is putative to provide high confidence to development organizations regarding the SUT quality (Kuhn & Okun, 2006). However, exhaustive testing is impractical due to time and budget constraints (Tassey, 2002). Moreover, execution of the entire test suite is also impractical, if high human interventions are required for testing a software system (Haug, Olsen, & Consolini, 2001). In the literature, researchers have reported different testing scenarios (Lin et al., 2012; Rothermel, Untch, Chu, & Harrold, 2001), which concludes that exhaustive testing requires a considerable amount of testing budget. Rothermel et al. (2001) reported that testing a product containing 20,000 lines of code requires seven weeks and several hundred thousand dollars to execute the entire test suite. Moreover, Lin et al. (2012) performed an empirical analysis of regression testing using 57,758 functions and 2320 test cases. They reported that the running time of a single test case ranges from 10 min to 100 min, which is based on the configuration sequence and required test activities. Finally, they estimated that to execute all 2320 test cases would require 36 test-bed days.

To deal with the aforementioned exhaustive testing problems, development organizations are attracted to adopt optimal testing strategies (Nachmanson, Veanes, Schulte, Tillmann, & Grieskamp,

* Corresponding author.
** Principal corresponding author.
E-mail addresses: saif.rehman@siswa.um.edu.my (S.U.R. Khan), saipeck@um.edu.my (S.P. Lee), wasimraja@ciit.net.pk (R.W. Ahmad), a.qureshi@comsats.edu.pk (A. Akhunzada), ic.victor.chang@gmail.com (V. Chang).

http://dx.doi.org/10.1016/j.ijinfomgt.2016.05.025
0268-4012/

2004). Analytics is useful in software design and testing (Chang, 2015b). Similarly, when we design a software, do not forget diverse security concerns (Akhunzada, Gani, Anuar et al., 2015; Akhunzada, Sookhak et al., 2015) to ensure that the software design supports penetration testing (ethical hacking) against hacking (Chang & Ramachandran, 2016).

In the literature, three main techniques have been discussed to support regression testing (Yoo & Harman, 2012): (i) *test suite reduction*: to find a reduced suite by permanently eliminating redundant test cases according to certain criteria, (ii) *test case selection*: to select such previously generated test cases that cover the modified portion of the software, and (iii) *test case prioritization*: to determine the ordering of test cases based on a particular objective such as to increase the rate of Fault Detection Effectiveness (FDE). In our context, the real challenge is to determine a subset of non-redundant test cases, which finds a maximum number of possible defects similar to the original test suite (Khan, Nadeem, & Awais, 2006). A tester can meet this challenge by randomly selecting the generated test cases (Chen, Kuo, Merkel, & Tse, 2010), but such random selection may end up as exclusion of essential test cases (Hao, Zhang, Wu, Mei, & Rothermel, 2012). Consequently, it has a negative impact on the fault detection capability of the reduced suite. A practical approach to solve the test-suite size problem is to find a minimal subset of test cases automatically, while keeping their fault detection capability similar to the original test suite. Test Suite Reduction (TSR) approach focuses on finding a minimal test suite by permanently discarding the redundant test cases from the original test suite (Dandan, Tiantian, Xiaohong, & Peijun, 2013). The optimal TSR problem is formally defined by Harrold, Gupta, and Soffa (1993) as stated below:

> ***Given:*** A test suite, TS, and a set of test requirements R_1, R_2,..., R_n, *that must be satisfied to provide the desired test coverage of the program, and subsets of TS*, T_1, T_2,..., T_m, *where each* T_i *is associated with each of the* R_i*s such that any one of the test cases* tc_j *of* T_i *can be used to test requirement* R_i.
>
> ***Problem:*** Find a Reduced Suite (RS) containing minimal test cases from TS that satisfies all test requirements R_i at least once.

The optimal TSR problem is known to be NP-complete problem and is equivalent to set cover problem (Michael & David, 1979). Researchers have recommended complete automation support for various activities of test suite development cycle, such as test generation, execution, and evaluation, to minimize high cost of regression testing. Bertolino (2007) has recommended 100% automated testing, which facilitates the tester to meet the challenges of a resource constrained testing environment. Motivated by this, researchers have proposed various frameworks that results various tools to provide automation facility during TSR process (Andrews, Haldar, Lei, & Li, 2006; Burger & Zeller, 2011; Campos, Riboira, Perez, & Abreu, 2012; Chae, Woo, Kim, Bae, & Kim, 2011; Dadeau, Ledru, & Du Bousquet, 2007; Horgan & London, 1992; Hsu & Orso, 2009; Jaygarl, Lu, & Chang, 2010; Kauffman & Kapfhammer, 2012; Li, Sahin, Clause, & Halfond, 2013; Pacheco & Ernst, 2007; Sampath, Sprenkle, Gibson, Pollock, & Greenwald, 2007; Sampath, Bryce, Jain, & Manchester, 2011; Wang, Ali, & Gotlieb, 2015; Woo, Chae, & Jang, 2007; Xie, Marinov, & Notkin, 2004; Xie, Zhao, Marinov, & Notkin, 2006; Zhang, Gu, Chen, Qi, & Chen, 2010; Zhang, Zhou, Hao, Zhang, & Mei, 2009). Consequently, automated TSR helps in accelerating the software delivery process compared to manual test filtering (Hsu & Orso, 2009). Automation has been achieved for Cloud storage for big data, which provide a supporting use case (Chang & Wills, 2016). Cloud service APIs Plays a vital role to integrate enterprise applications as they offer a set of protocols, which helps in connecting applications to various cloud services. APIs are useful for different disciplines such as business intelligence (Chang, 2014a) and social networks (Chang, 2015a, 2014b).

Prior works (Elberzhager, Rosbach, Münch, & Eschbach, 2012; Yoo & Harman, 2012) lack in considering and analyzing TSR frameworks that is necessary to understand the body of knowledge in the area of TSR. To the best of our knowledge, this is the first effort that studies the TSR frameworks and tools comprehensively. The main objective of this survey is to provide an up-to-date view and in-depth analysis of state-of-the-art that is necessary to understand the body of knowledge. The survey analyzes, synthesizes, and categorizes current state-of-the-art TSR frameworks and tools. Furthermore, the survey focuses on identifying future research opportunities in this field of study. The main contributions of the paper are listed below:

- An extensive review on automated support for TSR.
- Presenting a thematic taxonomy to categorize the existing literature based on various testing domains, approaches, and their corresponding parameters.
- Providing a detailed comparative analysis of TSR frameworks based on the devised taxonometric parameters.
- Highlighting the strengths and limitations of the TSR tools and frameworks related to a particular testing domain.
- Synthesizing current state-of-the-art based on the underlying common philosophies.
- Highlighting several potential research issues in this field of study.

The rest of the paper is organized as follows: Section 2 presents a thematic taxonomy of TSR frameworks. Section 3 discusses current state-of-the-art TSR frameworks/tools based on various testing domains and approaches. Furthermore, it discusses strengths and weaknesses of existing TSR frameworks. Section 4 provides a comparison of current TSR frameworks based on the devised thematic taxonomy. Section 5 discusses potential research issues followed by concluding remarks in Section 6.

2. Taxonomy of Test Suite Reduction (TSR) frameworks/tools

This section presents a taxonomy for the thematic classification of TSR frameworks and tools based on defined parameters as depicted clearly in Fig. 1. The defined parameters include: (i) approach type, (ii) testing paradigm, (iii) optimization type, (iv) coverage source, (v) execution platform, (vi) computational mode, (vii) license type, (viii) evaluation, (ix) customizability, and (x) support.

The first identified parameter is the *approach type*, which represents the principal category of TSR approaches focused by a TSR framework. There are four main attributes for approach type: (i) coverage-based, (ii) search-based, (iii) Integer Linear Programming (ILP) based, and (iv) similarity-based. The coverage-based TSR approaches greedily select such test cases that cover maximum portions (e.g., statements or branches) of a program under test. In contrast, search-based approaches employ various search algorithms, such as Genetic Algorithm (Deb, Pratap, Agarwal, & Meyarivan, 2002), to find diverse test cases from the initial population. Conversely, ILP-based approaches determine minimal global solution for TSR problem based on the defined objectives and constraints (Black, Melachrinoudis, & Kaeli, 2004). Furthermore, ILP-based approaches achieve Pareto-optimal solution for multi-objective TSR problem (Baller, Lity, Lochau, & Schaefer, 2014). Notice that Pareto-optimal solution is a *non-dominated* solution, which cannot be further improved (Yoo & Harman, 2007). In contrast, similarity-based approaches focus on finding most different