EAI/Springer Innovations in Communication and Computing

Manju Khari
Deepti Bala Mishra
Biswaranjan Acharya
Ruben Gonzalez Crespo *Editors*

Optimization of Automated Software Testing Using Meta-Heuristic Techniques

EAI
RESEARCH MEETS INNOVATION

Springer

EAI/Springer Innovations in Communication and Computing

Series Editor
Imrich Chlamtac, European Alliance for Innovation, Ghent, Belgium

The impact of information technologies is creating a new world yet not fully understood. The extent and speed of economic, life style and social changes already perceived in everyday life is hard to estimate without understanding the technological driving forces behind it. This series presents contributed volumes featuring the latest research and development in the various information engineering technologies that play a key role in this process. The range of topics, focusing primarily on communications and computing engineering include, but are not limited to, wireless networks; mobile communication; design and learning; gaming; interaction; e-health and pervasive healthcare; energy management; smart grids; internet of things; cognitive radio networks; computation; cloud computing; ubiquitous connectivity, and in mode general smart living, smart cities, Internet of Things and more. The series publishes a combination of expanded papers selected from hosted and sponsored European Alliance for Innovation (EAI) conferences that present cutting edge, global research as well as provide new perspectives on traditional related engineering fields. This content, complemented with open calls for contribution of book titles and individual chapters, together maintain Springer's and EAI's high standards of academic excellence. The audience for the books consists of researchers, industry professionals, advanced level students as well as practitioners in related fields of activity include information and communication specialists, security experts, economists, urban planners, doctors, and in general representatives in all those walks of life affected ad contributing to the information revolution.
Indexing: This series is indexed in Scopus, Ei Compendex, and zbMATH.

About EAI - EAI is a grassroots member organization initiated through cooperation between businesses, public, private and government organizations to address the global challenges of Europe's future competitiveness and link the European Research community with its counterparts around the globe. EAI reaches out to hundreds of thousands of individual subscribers on all continents and collaborates with an institutional member base including Fortune 500 companies, government organizations, and educational institutions, provide a free research and innovation platform. Through its open free membership model EAI promotes a new research and innovation culture based on collaboration, connectivity and recognition of excellence by community.

Manju Khari • Deepti Bala Mishra
Biswaranjan Acharya • Ruben Gonzalez Crespo
Editors

Optimization of Automated Software Testing Using Meta-Heuristic Techniques

Springer

EAI
RESEARCH MEETS INNOVATION

Editors
Manju Khari
School of Computer & Systems Sciences
Jawaharlal Nehru University
New Delhi, Delhi, India

Deepti Bala Mishra
Department of MCA
GITA Autonomous College
Bhubaneswar, India

Biswaranjan Acharya
Department of Computer Engineering-AI
Marwadi University
Rajkot, Gujarat, India

Ruben Gonzalez Crespo
Computer Science and Technology
Universidad Internacional de La Rioja
La Rioja, Spain

ISSN 2522-8595 ISSN 2522-8609 (electronic)
EAI/Springer Innovations in Communication and Computing
ISBN 978-3-031-07296-3 ISBN 978-3-031-07297-0 (eBook)
https://doi.org/10.1007/978-3-031-07297-0

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

Test automation is now ubiquitous, and almost assumed in large segments of the research. Agile processes and test-driven development are now widely known and used for implementation and deployment. This book presents software testing as a practical engineering activity, essential to producing high-quality software. This book is beneficial for an undergraduate or graduate course on software testing and software engineering, and as a resource for software test engineers and developers. This book has a number of unique features:

1. It includes a landscape of test coverage criteria with a novel and extremely simple structure. At a technical level, software testing is based on satisfying coverage criteria. The book's central observation is that there are few truly different coverage criteria, each of which fits easily into one of four categories: graphs, logical expressions, input space, and syntax structures.
2. It projects a balance of theory and practical application, presenting testing as a collection of objective, quantitative activities that can be measured and repeated. The theoretical concepts are presented when needed to support the practical activities that researchers and test engineers follow.
3. It assumes the reader is learning to be a researcher whose goal is to produce the best possible software with the lowest possible cost. The concepts in this book are well grounded in theory, are practical, and most are currently in use.

Through this book an effort to in support of the idea of promoting software testing and establishing as to software testing is made possible.

Chapter 1

In Chap. 1, test suite minimization is done with an intention of optimizing the test suite, and software faults detection and localization as well as adjoining activities are focused on. In this chapter, code coverage and mutant algorithms are used to generate the compact test cases on which an algorithm is applied for identifying and locating errors. To optimize the test cases, NSGA-II algorithm is used. Defects4j repository has been used for generating and performing tests.

Chapter 2

Chapter 2 focuses on mutation testing, which is the fault-based software testing approach that is widely applicable for assessing the effectiveness of a test suite. The test suite effectiveness is measured through artificial seeding of faults into the programs under test. Six open-source mutation testing tools and JAVA-based MTT (Jester, Javamut, MuJava, Jumble, Judy, and Javalanche) are analyzed. The results are based on the performance of various JAVA programs and two real-life applications. Benchmark comparison among the MTT is presented in terms of mutants, mutation operator, mutation score, and quality output.

Chapter 3

In Chap. 3, the authors present MBT and state-based test case generation using a state chart diagram. Firstly, the authors review the main concepts and techniques in MBT. Then, in the next step, they review the most common modeling formalisms for state chart diagram, with focus on various state-based coverage criteria. Subsequently, the authors propose methods for a synchronous state-based testing approach to generate test cases.

Chapter 4

In Chap. 4, the Author designed and developed a TCP technique to enhance the fault detection rate of test cases at the early execution of the test suite. The developed algorithm was examined with two benchmark algorithms on four subject programs to evaluate the performance of the algorithm. APFD metrics are used as performance evaluation metrics and the performance of the developed algorithm outperforms both of the benchmark algorithms.

Chapter 5

In Chap. 5, authors analyze the already available and enhanced testing techniques for the improved and good quality product. Some recent research studies have been summed up in this work as software testing is acquiring more significance these days.

Chapter 6

In Chap. 6, authors identify and analyze an existing research paper previously conducted by different researchers on predicting software reliability using a machine learning approach in the context of formulated research questions.

Chapter 7

In Chap. 7, a systematic approach to finding bugs means errors or different other defects in a running application which are ready to tested. It also helps to analyze the actual programs and to lower the cost of finding errors. And different EAs like GA-, PSO-, ACO-, and ABCO-based methods have been already proposed to generate the optimized test cases.

Chapter 8

Chapter 8 represents a use case of optimization of software testing in different wireless sensor network applications. The survey in the paper also shows that the use of a metaheuristic is not limited to WSN, and the use of a metaheuristic in automated software testing is exemplary. In the field of software testing, optimization of test cases and increasing usability are a few tasks that can be optimized with the help of metaheuristic algorithms.

Chapter 9

In Chap. 9, the author develops my CHIP-8 emulator for software testing strategy for playing online games on many platforms. The author lists each instruction explaining what it does and how it carries out it while providing the detailed documentation of our CHIP-8 emulator and thus, providing metaheuristic high-level solutions to fix them.

Chapter 10

Chapter 10 describes defects maintainability prediction of the software. This chapter evaluates the mentioned scenario by using maintainability index and defect data. The maintainability index is computed using the object-oriented metrics of the software.

Chapter 11

The book ends with Chap. 11, which develops a hybrid metaheuristic encryption approach employing software testing for secure data transmission named EncryptoX. The main objective behind doing this project report is to gain skills and knowledge regarding various cryptography and storage techniques used in software testing.

New Delhi, India — Manju Khari
Bhubaneswar, India — Deepti Bala Mishra
Rajkot, Gujarat, India — Biswaranjan Acharya
La Rioja, Spain — Ruben Gonzalez Crespo

Contents

NGA-II-Based Test Suite Minimization in Software

Renu Dalal, Manju Khari, Tushar Singh Bhal, and Kunal Sharma

1 Introduction

Developing software is one of the major works that is being done in the industry in this era of technology. For developing software one of the major tasks is testing for errors and issues. Running the complete test suite without minimizing is a tedious job as it will induce a big load on the system and the operation under execution. Test case minimization is one of the options to reduce the test suite. For testing purposes, the first step is to create test suites in which some operations are defined or a set of information in which the software has to perform and the results of which describe the ability of the software to perform under that kind of task.

But after creation of a test suite, the next step is to minimize the test suite as it can contain many redundant and faulty tests which have to be removed to improve the efficiency of the testing. The load on the machine gets reduced if proper minimization is done. Test case minimization is used to getting the compacted test case. This aids in testing that modification done in software program has not affected the unmodified part of the software. Identifying and locating errors is one of the major tasks which has to be done to minimize the test suite. But performing them at the same time is a different issue as they are subsequent activities. At first, fault detection in the test suite using failing test cases is done, and then localization is done by using the pass and fail information of the test suit.

R. Dalal (✉)
Department of Information Technology, MSIT, GGSIP University, Delhi, India

M. Khari
School of Computer and Systems Sciences, Jawaharlal Nehru University, Delhi, India

T. S. Bhal · K. Sharma
Department of Computer Science, AIACT & R, GGSIP University, Delhi, India

M. Khari et al. (eds.), *Optimization of Automated Software Testing Using Meta-Heuristic Techniques*, EAI/Springer Innovations in Communication and Computing, https://doi.org/10.1007/978-3-031-07297-0_1

Software testing is one of the major processes in all the SDLC phases present. Testing takes a lot of time and resources present at our end. Research says that in about 50% of the total time given in development, 25% of it should be given only to debugging. Test suite minimization helps in this process by reducing the time and computing power applied on the tasks. As both detecting and localizing faults should be performed one after another if we can combine these processes, this can significantly reduce the work and load applied. The aim of this chapter is to achieve greater efficiency as possible in minimizing and to achieve better CPU utilization, memory utilization, and disk usage.

A software always gets updated, and new functionality is added from time to time due to which new test cases are added in the test suite. After some updates, there are some test cases which are no longer needed and are an overhead over the system, so they have to be removed or should not be considered that is why it is a must to perform minimization over the test suite. It also becomes easy to detect faults in the minimized suite. As mentioned above, the detection and localization tasks are done one after another. Combining them is a difficult process but essential as performing the in concurrently takes more time and usage of other resources. For combining them offline techniques can be used. The reduced test suite obtained after these processes can be used for regression testing. Vidács et al. and many other researchers work on this approach. They proposed an approach for combining both these processes and minimizing the test suite.

For doing all this, multi-objective optimization algorithms come into play. Multi-objective optimization algorithm deals in the domain of optimization problems where multiple objective functions are optimized. The solution obtained is known as non-dominated, Pareto adequate or non-inferior, and Pareto optimal, if none of the objective functions can enhance the value without deteriorating the other objective values. An assessment is done in this study on projects taken from Defects4J repository [1–3]. This assessment is performed by using NSGA-II, coverage, and mutation algorithms. The main reason for undertaking minimization is:

- To reduce time for testing purpose
- Less system requirement
- Redundant test case elimination
- To predict faults easily

2 Background

2.1 What Is Test Suite?

Test suite is a collection of tests which helps testers in executing and performing testing and reporting faults and errors present in the software. A single test case can be added to many test suites. A test suite consists of many test cases which describe

the various conditions which the software has to encounter while being operated. It can also be defined as a collection of scenarios which define the scope of testing for a given execution environment.

Test suites are used for identifying gaps in testing efforts where successful completion of a test case can occur before the next step begins. Test suites are also useful like build verification test, smoke test, end-to-end integration test, and functional verification test.

A test suite can divide into three types:

- *Static Suite*: In this suite, creation of a collection of named scenarios will remain static once defined. In this type sequence of execution is always guaranteed. The order of execution is defined in this type of suite.
- *Filter-Based Suite*: In this suite, custom sets of fields are defined using filter parameters. This type of suite is used for targeted testing of specified portions.
- *Requirement-Based Suite*: Here test suite is created based on the user requirements. This is mostly used in agile environments.

2.2 Minimization of Test Suite

A test suite contains a huge amount of test cases, and executing them is an annoying task. Many researches have been done to minimize this annoying task. A test suite is minimized by removing the redundant test cases and removing those cases which are no longer needed, or the functionality has been removed in the updated version of the software [4].

2.3 Partitioning

Partitioning can be divided into classes into equivalent partitions. This idea is based on the concept of equivalence partition in set theory. These partitions are undistinguishable. Majorly two types of partitions are accessible: statement partitioning and mutation partitioning. Statement partitioning is based on the code coverage. It means portioning the code segment on the basis of their code coverage information into diverse classes. Mutant partitioning is another type of partitioning method in which the mutants are partitioned on the basis of their kill information by test cases. Mutants can be partitioned into various partitions, for measuring this a factor d-score is introduced by the researchers, it provides the information of number of mutants differentiate by the considered test-cases [5].

2.4 Optimization Algorithms

Optimization algorithms are used for optimizing test cases. These algorithms are used to reduce the test suite size and to produce results which can be used further in multi-objective tasks. There are many optimization algorithms present. It can broadly be classified into two categories, that is, single objective and multi-objective [6]. Multi-objective includes NSGA-II, NSGA-III, MO-PSO, MO-BAT, etc.

3 Defects4J

3.1 About Defects4J Repository

Defects4J is a repository which contains a collection of reproducible bugs which is used for advancing software research. Defects4J consists of many projects, and there are many versions of each project which can be used to generate test suites of various types. It contains 835 bugs from many open-source projects like Chart, Cli, Closure, Math, Time, Csv, etc. Test suites are generated by using some generator functions and then providing a version of the project. It also provides the support for integrating any methods of applying algorithms outside the repository scope. It comes with the support of applying basic coverage and mutation algorithms [7]. Defects4J comes with basic functionality like performing checkout and compiling and performing testing on a test case.

4 Code Coverage

The code coverage is a metric that illustrates the extent of the source program code that has been tested. It is a part of white box testing. It is used to determine the quantitative measure of the code. It generates the result of the test suite's code coverage. There are many reasons why we use code coverage, and some of them are to: (1) Offers Quantitative measurements. (2) Describes the extent to which the code is tested. (3) Calculate test implementation efficiency [8–10]. There are many techniques in which code coverage can be performed such as:

- Decision coverage
- Statement coverage
- Toggle coverage
- Branch coverage
- FSM coverage

In this chapter statement and branch coverage are used.

4.1 Statement Coverage

Statement coverage involves execution of all the executable statements in the source code at least once. It is white box testing technique. It is used to calculate the number of statements which can be executed on the given requirements. Here as a part of white box testing, the aim is to understand the working of the code at internal levels. Its main goal is to include all the finite routes, lines, and statements present in the source code. Maximization of statement coverage intends to discover the minimized test case and can enhance its value. The statement coverage metric is defined as Eq. 1:

$$SC = \frac{\left|\{s \in M \left| s \text{ covered by } T\right.\}\right|}{\left|M\right|} \tag{1}$$

Here, |*M*| means total number of statements present in the source program.

4.2 Branch Coverage

The outcome of the code module is tested in branch coverage. Branch coverage's main goal is to make sure every possible branch is tested. It tells us about the independent code segments present in the codes [11]. The branch coverage ensures that every section of each control structure may be examined at least once.

Maximization of branch coverage helps in finding the minimized test case, and it maximizes the value of branch coverage. The metric of branch coverage is represented in Eq. 2.

$$\text{Branchcover} = \frac{\left|\{b \in P \left| b \text{ covered by } T\right.\}\right|}{\left|P\right|} \tag{2}$$

Here, |*P*| represents the total number of branches in the code.

5 Proposed Approach

5.1 Workflow of Approach

Test suite minimization is required in software for the same the proposed approached in represented in the Fig. 1

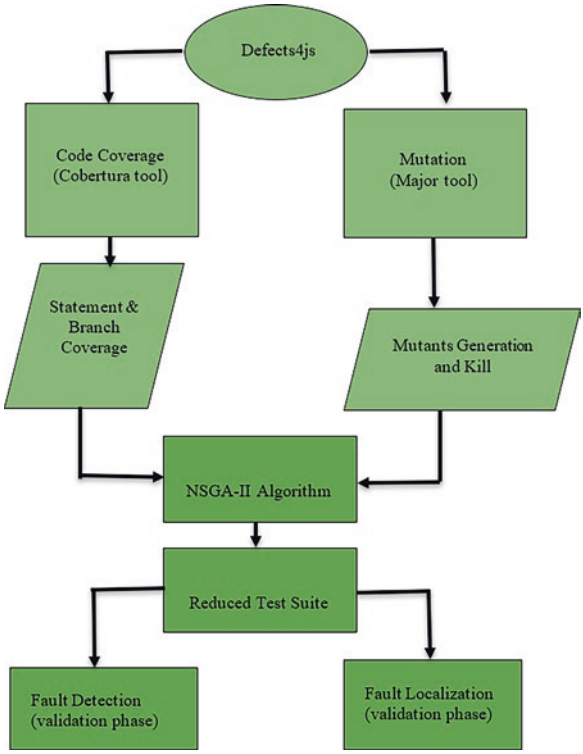

Fig. 1 Workflow of proposed approach

5.2 Optimization NSGA-II Algorithm

NSGA II is the multi-objective algorithm which comes in the class of optimization. It stands for elitist non-dominated sorting genetic algorithm. This algorithm is both elitism preserving and diversity preserving. Elitist means it keeps the best solution for the next iteration from the current one. Non-dominated searching means if there are two individuals A and B, A dominates to B, if and only if there is no objective of A worse than that objective of B and there is at least one objective of A better than that objective of B. The objective of non-dominated sorting is to find out which individual belongs to which front. Mathematically domination is:

A(x_1, y_1) dominates B(x_2, y_2) when : (x_1<=x_2 and y_1<=y_2) and (x_1<x_2 or y_1<y_2)

One of the fronts may not fit properly in the size of the parent population as before for this *crowding distance* is used. To keep a good spread in NSGA-II and avoid local maxima or minima, crowding distance decides which individuals are added to the new population. Individuals with higher crowding distance are picked first. After this new offspring is created which has the same size as the parent. This process happens in three phases tournament selection, crossover, and finally mutation. All this happens for some iterations, and then the result is taken.

Pseudo Code of the Algorithm

```
Fast sNon-Dominated Sort:


for every p ∈P
    S_p = Φ
    n = 0
    for every q∈P
        if q<p then
            S_p = S_p ∪{q}
        else if q<p then
            n = n +1
    if n = 0 then
        p_rank = 1
        F_1 = F_1∪{q}
i = 1
while F_i ≠Φ
    Q = Φ
    for every q∈F_i
        for every q∈S_p
            n_q = n_q -1
            if n_q = 0
               q_rank = i+1
               Q = Q∪{q}
    i = i + 1
    F_i = Q

Crowding_Distance_Assignment


l = | I |
for every i, set I[i] dist = 0
for every objective m
    I = sort(I, m)
    I[1]dist = I[1]dist = ∞
    for i = 2 to (l-1)
              I[i]dist = I[i]dist + (I[i+1].m - I[i-1].m)/
(fmax_m - fmin_m)

Final_Step


R_t = P_t∪Q__t
F = Fast_Non_Dominated_Sort(Rt)
```