Università degli Studi di Salerno
Dipartimento di Informatica

Tesi di Laurea di I livello in
Informatica

Template tesi ISISLab

Relatore
Nome Cognome
Correlatore
Dott. Nome Cognome

Candidato
Nome Cognome

Academic Year 2021-2022

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Contents

1 Introduction **1**

2 Literature **4**

3 Conclusions **6**

Chapter 1

Introduction

La Figure 1.1 è una figura di esempio.

Figure 1.1: Questa è una immagine di esempio

Things to add:

- where different types of coverage are useful
- Add formulas for coverage kinds

Coverage testing

Coverage is one of the metrics employed during testing to asses what portion of the source code is "covered" by the test suite i.e., what portion of the code is executed when the tests run. Coverage is essential to extract information about the general quality of a test suite and helps determining how comprehensively the software is being verified. As a result, coverage can be classified as a white-box testing technique.

Source code coverage can be expressed according to different sub-metrics:

- Statement coverage aims executing every single statement in the code.
- Branch coverage, also known as decision coverage, measures how many decision structure have been fully explored by the test cases.
- Mutation coverage, also known as fault-based testing, aims at purposefully introducing faults in the program in order to check whether or not the test suite is able to identify them. If the fault is correctly detected, the mutant is "killed". One issue of mutation is scalability, since generating and compiling the mutants, before running the test cases, can be time consuming end quickly exhaust testing resources. Additionally, the introduced mutations can be classified as weak or strong: with strong mutation, the artificial fault is propagated to an observable behaviour, while weak mutants are confined to more specific environments.
- Function coverage measures how many functions have been called by the test cases.
- Condition coverage determines the number of boolean conditions/expressions executed in the conditional statements.

To reach statement coverage, it is sufficient to execute a branch in which the statement is control dependent.

A high coverage can sometimes be deceiving, however: in the case of Machine Learning Systems (MLS), where typically the source code is made up of a sequence of library functions and API invocations [2], thus resulting in very high statement and branch coverage with realtively modest test suites. Additionally, the effectiveness of such systems is highly determined by the dataset employed for model training and validation, which cannot be covered by tradition test cases.

Coverage can also be measured at any testing levels; while at the unit test-level we focus mostly on the coverage of statements and branches, at the system-testing level, the coverage targets shift towards more complex elements, such as menu items, business transactions or other operations that require multiple components of the system to work properly.

Automatic test case generation

Test case generation can be seen as a multi-objective problem, given that the goal is to cfover multiple test targets.

Search-based approaches for test case generation use optimization algorithms to attempt to find the best candidate test case with the objective to maximize fault detection. Genetic Algorithms (GAs) are an example of an evolutionary search approach for test case generation; starting from an initial, often randomlly genrated, population of test cases, the algorithm keeps evolving the individuals according to simulated natural evolution theory principles. In this context, a typical fitness function of a GA would measure the distance between the execution trace of the generated test cases and the coverage targets.

Testing in the Internet of Things

Chapter 2

Literature

When formulating the test case search problem as a many-objective optimisation problem, the goal is to minimize all the individual distances from all the test targets in the class under test.

One of the most popular multi-objective algorithms for this problem is the Non-dominated Sorting Genetic Algorithm II (NSGA-II). This algorithm is based on three principles:

- It uses elitism when evolving the population: the most fit individuals are carried over along the offsprings.
- It uses an explicit diversity-preserving mechanism, the Crowding distance.
- It emphasizes the non-dominated solutions, as its name suggests.

First of all, in the context of test cases, domination can be expressed by the following relation:

The NSGA-II algorithm works as follows:

- Perform non-dominated sorting in the combination of parent and offspring populations and classify them by fronts.
- Build new population according to front raking.
- It emphasizes the non-dominated solutions, as its name suggests.

DynaMOSA, Dynamic Many-Objective Sorting Algorithm [1] is an approach that focuses on ..., and has been developed as an evolution of MOSA. This latter solution implements a many-objective GA to tackle test case generation and has three main features:

***Definition* 1:** *A test case x dominates another test case y (also written $x \prec y$) if and only if the values of the objective function vector satisfy the following conditions:*

$$\forall i \in \{1, \ldots, k\} \quad f_i(x) \leq f_i(y)$$
$$\textit{and}$$
$$\exists j \in \{1, \ldots, k\} \textit{ such that } f_j(x) < f_j(y)$$

Figure 2.1: Test case domination

- instead of ranking candidates for selection based on their Pareto optimality, it uses a preference criterion. This criterion selects the test case with the lowest objective score for each uncovered target; these selected individuals are given a higher chance of survival, while other test cases are ranked with the traditional NSGA-II approach.
- The search is focused only on the uncovered coverage targets.
- All tests that satisfy one or more of the uncovered targets will be archived and used as the final test suite once the search ends.

In many-objective optimisation problems, candidate solutions are typically evaluated in terms of Pareto dominance and Pareto optimality.

DynaMOSA has been employed with Java classes.

Traditionally, with evolutionary search-based approaches, the algorithm is applied multiple times, once for each coverage criterion; doing so may Ultimately, however, the effectiveness of the solution depends on the problem

Chapter 3

Conclusions

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Bibliography

[1] F. M. Kifetwe A. Panichella and P. Tonella. "Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2018).

[2] A. Author. "Article title". In: *Journal name* (2099).

[3] A. Author. "Tesi di esempio ISISLab". 2099.

[4] A. Author and A. Author. *Book reference example*. Publisher, 2099.

[5] *Example*. URL: `https://www.isislab.it`.

[6] V. Riccio G. Jahangirova A. Stocco N. Humbatova M. Weiss and P. Tonella. "Testing machine learning based systems: a systematic mapping". In: (2020).