A Multi-Objective Optimization Approach to Test Suite Reduction

Jorge Costa

Mestrado Integrado em Engenharia Informática e Computação

Jorge Costa: Rui Maranhão (PhD)

Co-Supervisor: José Carlos Campos (MSc)

June 29, 2015
A Multi-Objective Optimization Approach to Test Suite Reduction

Jorge Costa

Mestrado Integrado em Engenharia Informática e Computação

June 29, 2015
Abstract

During the software development process, the code needs to be changed many times. This is necessary either because requirements may have changed, developers are correcting a fault, or programmers are re-factoring the code to increase its quality. After each software change, the software needs to be tested again to ensure that it is still working correctly. This is called regression testing.

Although regression testing is necessary to ensure software quality, it is a costly step in the software development process. Normally, the greater the number of tests, the more time they take to run - most programmers are not productive during this time. Therefore, the regression testing cost increases with the number of tests.

Running redundant or obsolete tests that do not increase the fault detection capabilities is an unnecessary cost. This is a problem the scientific community is well aware of. Several automatic approaches were proposed to remove unnecessary tests. Most of the approaches are single objective - they find the best test suite subset that maximizes a given variable, like code coverage.

Although single objective reduction is a good fit for some cases, for many others it is not. We may want to give more relevance to minimizing the number of tests than to maintaining good code coverage (e.g. testing suite is too big). Multi-objective test suite reduction is more adaptable to each software project.

Software testing is useful in detecting faults, and could also be able to locate them using automatic fault detection techniques, such as spectrum based fault localization. The diagnosability is dependent on the test suite. To the best of our knowledge, until now there is no multi-objective approach that tries to produce test suites that are good at localizing faults.

According to some authors, without tools that make test reduction easily accessible by programmers, practical adoption will be limited. This document proposes a practical solution for multi-objective test suite reduction that tries to minimize fault localization effort.

On our empirical evaluation, we were able to improve the average fault localization effort (FLE) value by 31%. And we were able to conclude at a 95% confidence level that our approach produces a better average (FLE) between 77% to 100% of the times when compared to the most common existing multi objective approach.
Acknowledgements

First of all, I would like to offer a big thank you to Professor Dr. Rui Maranhão, my supervisor and someone whom I would call a friend. He introduced me to the wonderful world of software engineering research and gave precious advice that allowed this dissertation to be better. In addition to bestowing academic knowledge, he also provided work methodology and motivation to make all this possible.

I would also like to thank my co-supervisor, José Carlos Campos - without our discussions and the references you provided me with, this dissertation would surely not be the same.

Filipa Guerra, Alexandre Perez, and Filipe Oliveira - without your reviewing this document, giving good suggestions and finding even the most hard-to-find typo, this document would not be the same. You are all wonderful friends.

Last but not least, thanks to Faculdade de Engenharia da Universidade do Porto and all the Professors I had the pleasure to meet - for providing me with the knowledge that I have acquired in these last few years.

Jorge Costa
“Testing is the process of comparing the invisible to the ambiguous, so as to avoid the unthinkable happening to the anonymous.”

James Bach
Contents

1 Introduction 1
  1.1 Context ........................................ 3
  1.2 Motivation ...................................... 5
  1.3 Research Questions ............................. 7
  1.4 Results ........................................ 8
  1.5 Document Structure ............................. 9

2 State of The Art in Test Minimization 11
  2.1 Minimization vs Selection and Prioritization ............... 11
  2.2 Classical Approaches ................................ 12
  2.3 Heuristics ........................................ 14
    2.3.1 GE Heuristic .................................. 14
    2.3.2 GRE Heuristic .................................. 15
  2.4 ILP Model ........................................ 15
  2.5 Constraint Satisfaction Approach .......................... 15
  2.6 MOO Coverage and Cost ................................ 17
  2.7 Two Objectives ILP .................................. 17
  2.8 Minimization in Practice .............................. 18
    2.8.1 Scenario at Google .............................. 18
    2.8.2 RZoltar ......................................... 19
  2.9 General Overview .................................... 20

3 A MOO Test Suite Minimization Approach 23
  3.1 Objectives ........................................ 24
    3.1.1 Requirement Coverage (Cov) ...................... 24
    3.1.2 Execution Cost (Cost) .......................... 25
    3.1.3 Coverage Matrix Density (ρ) ..................... 25
  3.2 Problem definition .................................. 27
    3.2.1 MOO Formal Definition .......................... 27
    3.2.2 Test Suite Minimization as a MOO Problem ............ 28
  3.3 Algorithm Inputs ................................... 29
    3.3.1 Coverage Matrix ................................ 29
    3.3.2 Test Execution Cost Vector ...................... 30
  3.4 Algorithm Description ................................ 30
    3.4.1 Decision Vector ................................ 31
    3.4.2 NSGA-II ......................................... 31
  3.5 Objectives evaluation ................................ 32
    3.5.1 Requirement Coverage (Cov) ...................... 33
## CONTENTS

3.5.2 Execution Cost \((Cost)\) .................................................. 34
3.5.3 Minimized Coverage Matrix density \((\rho)\) ...................... 35

### 4 An Effective Minimization Tool 39

4.1 Components ................................................................. 40
   4.1.1 Inputs Collector ...................................................... 41
   4.1.2 Test Execution Logger .............................................. 42
   4.1.3 Minimization System .............................................. 42
   4.1.4 Visual Studio Extension .......................................... 44
   4.1.5 Graphical User Interface ........................................ 44
   4.1.6 Command Line Interface ......................................... 45
   4.1.7 Data Manager ...................................................... 45
   4.1.8 SBFL ................................................................. 45

4.2 Communication ............................................................ 47

4.3 User Interface ............................................................. 50

### 5 Empirical Evaluation 55

5.1 Experimental Subjects .................................................. 55
5.2 Experimental Setup ..................................................... 56
5.3 Experimental Results ................................................... 58
5.4 Threats to Validity ..................................................... 61

### 6 Conclusion and Future Work 63

6.1 Main Contributions ..................................................... 64
6.2 Future Work ............................................................. 65

References 67
# List of Figures

1.1 Ariane 5 disaster ........................................... 2
1.2 V model of software development process .................... 4

2.1 Example coverage matrix for the constraint-based approach as shown by Campos et al. ........................................ 16
2.2 RZoltar Interface [CRPA12] .................................................. 20

3.1 Coverage matrix .................................................. 30
3.2 Flow chart of NSGA-II by Tushar Goel ............................ 32

4.1 Components of the solution as seen on visual studio IDE ............ 41
4.2 JSON representation of the minimization inputs ....................... 47
4.3 JSON representation of the components ............................. 48
4.4 JSON representation of the test cases ............................... 49
4.5 JSON representation of the minimization output ..................... 49
4.6 Visual Studio extension start button and results display .............. 50
4.7 Display of test cases on a given solution and save button .......... 51
4.8 Save solution window ................................................. 52
4.9 Visual Studio test explorer with the chosen minimized solution ........ 53
4.10 General Overview of Visual Studio environment with the proposed minimization approach ........................................ 53

5.1 Improvement of average FLE when comparing our approach with time and coverage 60
5.2 Improvement of minimum FLE when comparing our approach with time and coverage ................................. 61
List of Tables

2.1 Example table representing test cases and the testing requirements they cover . . . 14
2.2 Summary of test suites at Google [YNH11] . . . . . . . . . . . . . . . . . . . . 18

5.1 Subjects used on empirical evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . 56
5.2 FLE value on original test suite and average and minimum FLE of each Pareto front 59
5.3 Confidence for minimum FLE results when comparing our approach with time 62
     and coverage approach . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
LIST OF TABLES
# List of Algorithms

1. Simple number of requirements covered computation ........................................ 33
2. Our approach to compute the number of requirements covered by the minimized test suite .......................................................... 34
3. Minimized test suite execution cost computation ............................................ 35
4. Simple UN-optimized method to compute the $\rho$ of a minimized test suite ...... 36
5. Pre-computation of the number of requirements covered by each test case ........ 36
6. Optimized computation of the $\rho$ value of a test suite ................................ 37
Abbreviations

IDE  Integrated Development Environment
CSP  Constraint satisfaction problems
ILP  Integer Linear Programming
UAT  User acceptance testing
API  Application programming interface
UAT  User acceptance testing
MOO  Multi Objective Optimization
FLE  Fault Localization Effort
JSON JavaScript Object Notation
SBFL  Spectrum Based Fault Localization
Chapter 1

Introduction

Software is present everywhere, nowadays - from the smartphone on our pockets, to the TV in our rooms. It is used to fulfill our entertainment needs and as a professional tool. There are many factors driving the growth of software use for the customer market:

1. It is easier to implement features in software than to implement them in other ways (like in hardware). Moreover, software allows features to be changed more easily and with a lower transition cost. That is what allows a device like a tablet to be used as a manual for aircraft repair and as a toy for kids.

2. Customers are always looking for the new thing and have high expectations from the product, which can more easily be met by the ability of software implementations to change and suffer upgrades.

The scenario is not different in the corporate world. Software use is also growing and taking a more relevant position on the value chain of enterprises. This change happens in most industries - from the automotive industry, that has been affected significantly by the industrial software revolution during the last few years [Gri03], to the retailing industry.

With the rise of the sharing economy, there are new innovative and efficient enterprises entirely based on software that are disrupting old industries. One example of that is what Airbnb, a multi-billion dollar company, did in the accommodation industry [Ger13].

Some say that software has taken such an important role in the organizations, that now every company is a software company [Kir11]. This presents challenges to the software industry like never before. If the enterprise is based on software and the software fails, all the enterprise fails with it.

It has been proven in the past that software is not perfect and sometimes fails, causing big disasters. One example of this is what happened to Ariane 5 rocket. Ariane 5 had a control software written in the Ada programming language. On June 4, 1996, it was self-destructed just 37 seconds after blast-off because of a programming error.
Another case that illustrates well the problems that software malfunction can cause is the one in Knight Capital. A software error allowed the computers of the company to sell and buy shares without any human intervention controlling the transactions. In just half an hour, the company lost half a billion dollars. This bug nearly pushed the company into bankruptcy and made it lose seventy-five percent of its market value.

Not only is software growing in use, but also in complexity. That is explained by a more important position that software has in the value chain of companies nowadays, and by the economic pressure to lower software costs. This economic pressure causes companies to outsource the software to many other different companies specialized in a more specific programming task. This outsourcing to many different companies, when each one of them may be using different technologies and programming languages, promotes the creation of more levels of indirection, sometimes leaving the systems more complex.

We know that the more complex the software is, the more likely it is to contain bugs. A computer bug could be more catastrophic than ever before, given the more important role of software nowadays. A software bug can cause money losses and even harm people.

The more time has passed since a computer bug was introduced in the code, the more it costs to find and correct it. So companies need to detect software errors as soon as possible. Given the present economic pressure on them, they also need to detect errors as cheaply as possible. If detecting errors is expensive in today’s market pressure, they are going to risk not running
Introduction

mechanisms that could detect errors early, even knowing that in the future they may face an even higher cost if the system fails.

1.1 Context

As previously referred, it is more important than ever that software works as expected. One common mechanism used to try to ensure that the software is working as expected is software testing. Besides software testing being used to ensure software quality, it can also lower the costs of finding a bug (debugging). This happens because, if a test fails, the most probable location of a bug is in the components that the test exercises. Some tools and methodologies have even been proposed in order to reduce debugging costs even further by taking advantage of tests [JAG09, CRPA12].

Software Testing could be used to ensure many properties of a system:

- Meets the design and development requirements.
- Responds correctly to the tested inputs.
- Has an adequate performance.
- Is secure.
- Is usable.
- Is ready to be deployed.
- Gives an answer to the stakeholders’ needs.
As software testing can ensure many properties, there are different types of tests to ensure each one. Moreover, software testing is present in many parts of the software development cycle. When programmers are implementing each module or designing it, they use unit test cases. To test the architectural design, they use integration tests that allow us to test the integration between the different modules. In order to test the system and high-level technological features, software companies do system testing. In the last level of testing, companies do UAT (User acceptance testing). UAT ensures that the requirements of the stakeholders were attained by the software. All the different types of testing and the corresponding software development phases can be observed in figure 1.2, which represents the V-Model of software development process.

During each change to the software source code, programmers can add new software errors that change some expected behavior of the system or remove some feature that was expected to be offered. Thus, after each software change, the system should be tested again, to ensure that it is working as expected. We call this process regression testing.
1.2 Motivation

Software needs to be changed during its life cycle. Some of the reasons are:

- Requirements have changed;
- Addition of new features;
- Upgrade to new technologies;
- Removing of bugs;
- Change of software design/architecture;
- Change of the software interface;
- Code re-factoring.

As there are many reasons leading to a change in software, changes occur many times during software life-cycle. As mentioned previously, each software modification could cause problems. Kim et al. predict that software modifications cause bugs seventy-eight percent of the time [KWZ08a]. At the beginning of this chapter, it was referred that software problems could cause big troubles - from costing money to the economy, to doing harm to people. This proves that testing software after each modification is really necessary. One possible way of doing regression testing of software is to use all the tests created for it.

Running tests has some costs. Some of the reasons for these costs are:

- Tests take time to run and most programmers are not productive while tests are running - they are waiting for the test results, which is completely understandable.
- Tests may not be fully automatized. Although this is not a good practice, some companies still use tests that need humans to perform some actions. In some cases, fully automatizing a test would be very costly or even impossible.
- Tests may use a lot of computing resources. Although one might may think that now this is a less problematic issue, given that bigger computing resources are available, that is not true - software has also become more complex and heavier in resources. Consequently, testing software also takes much more computing resources than before. Moreover, there is a new reality with companies paying the computing resources they use due to cloud base infrastructure. So now these have become more important criteria. Because of the abundantly common cloud base scenario, each time a test is run, companies pay something for running it.
- Tests may need to interact with pay-per-call external application programming interfaces. With the advent of cloud computing, the pay-per-call model has become greatly common.
Introduction

There are pay-per-call API’s to do almost everything, from data mining and machine learning, to video and music encoding and decoding. If a software system uses one of those API’s, a high-level test case may also need to use it. Thus, developers are paying each time one of those tests is run.

Test suites tend to grow in size, becoming huge[HGS93a]. Many tests on a test suite can be obsolete or redundant[HGS93a, Mic12, YH10a]. As running tests may have a cost, explicit or not, running all the tests may have a big cost or even be completely impractical, since running all of them could take an enormous amount of time[YNH11]. Therefore, it is necessary to identify those redundant tests and discard them from the regression test suite. If this is not done, given the costs that regression testing might have with all tests of the suite, software companies may feel tempted to do less regression testing and not test after each software modification. This is problematic given that, as mentioned, a bug can be harmful during/after the deployment.

It has been proven true, not only in an academic context, but also in practice, that doing testing with all the tests could be impractical. Microsoft, one of the world’s biggest software producers, recommends:

“Periodically review the regression test library to eliminate redundant or unnecessary tests. Do this about every third testing cycle. Duplication is quite common when more than one person is writing test code. An example that causes this problem is the concentration of tests that often develop when a bug or variants of it are particularly persistent and are present across many cycles of testing. Numerous tests might be written and added to the regression test library. These multiple tests are useful for fixing the bug, but when all traces of the bug and its variants are eliminated from the program, select the best of the tests associated with the bug and remove the rest from the library ”[Mic12].

Manually selecting tests for the regression test library is not easy because, as stated by Microsoft, very often tests are done by more than one person. If a given programmer did not create a test, that programmer probably has no idea of what it does and its relevance. Thus, a programmer may feel uneasy about removing a test without being completely sure of its importance. There is a need for tools that assist in the minimization of the test suite to be used on regression tests.

There is awareness among the scientific community about the risk of test suites getting too big, making it impractical to run all the test cases. There are two main ways of optimizing test suites: taking into consideration one objective and finding the minimum subset of the original test suite that still covers the same number of requirements (lines of code, usually) as the original; model the problem as an optimization problem, taking into account several objectives, called multi-objective test suite minimization.

The most commonly used objectives on multi-objective test suite optimization are coverage and time [YH10a]. The aim of automatic test cases is to not only detect faults on the software, but also to help locating them. Although most multi-objective test suite minimization approaches try
Introduction

to maximize fault detection ability, by maximizing code coverage, nothing is done to improve the ability of the minimized test suite help locate faults.

1.3 Research Questions

In addition to all the engineering questions to implement and evaluate a practical and easy-to-use software tool, there are two main research questions in this work.

How to maximize fault localization efficiency during multi-objective test suite minimization?

Most of the multi-objective approaches proposed until now focus on maximizing fault detection ability - normally by maximizing code coverage or historical fault detection - and on minimizing the execution cost - usually by minimizing the time tests take to run.

As refereed, Automatic test cases are not only useful in detecting faults, but also in providing helpful information for locating those faults on the source code. This information could be used manually or automatically. Given several developments, the use of automatic tools for locating faults in software is expected to grow.

In the current multi-objective approaches, test suites are minimized and some tests are discarded, without taking into consideration that a discarded test could be the most relevant one for locating a fault on the source code. In the worst case scenario, this could lead to automatic fault localization tools being useless.

In our work, we researched how to integrate fault localization ability during multi-objective optimization.

In order to ascertain if the research question answers and if our approach would generate solutions that are globally more efficient in locating faults on the source code, we ran automatic fault localization approaches on random faults inserted on real software projects. Then a comparison on the efficiency in localizing faults on those projects between our approaches and the current ones from the literature was made.

How to exchange test minimization information in a "standardized" way?

One of the reasons that explains the lack of practical tools is that, although some algorithms have been created through research and researchers validate them, most of the times they do not create a ready-to-use tool. That happens because creating a usable and functional software product is not the core objective of the researchers.

On the other hand, if a software company wants to create an automatic regression testing tool to integrate into an IDE, the company will have to check the algorithms from the research and implement them from scratch. The algorithms created by the researchers are not trivial and are complex to understand by people that have not studied them.
Introduction

If there was a standard way of communicating the necessary information for the test reduction, the knowledge transfer from the research to the industry would be much easier. This would contribute to the propagation of test reduction systems.

Researchers would create their tests reduction algorithms that receive inputs in a standard way and the algorithms from the research would be able to work with all the tools that implement this "standard", no matter the programming language or the testing system. This would make the validation work of researchers a much easier task. For companies that create software development tools, their task would become much easier, too. When a given enterprise is creating a test reducing system to be part of an IDE, the company would not need to implement the algorithms that do test reduction. The tool could make available all the algorithms from the research that use this standard and let an advanced user choose the algorithm they want to use.

By having a standard way of communicating test reduction information, the software development utility producers could focus on their job - get the information needed for the test reduction (code coverage, test run times, etc.) and create an attractive, easy-to-use tool for the users. The researchers could focus on creating a really efficient test reduction system without having to worry about getting the information needed and creating an interface for the user.

To contribute to the dissemination of test reduction, a first proposal for a standardized way of exchanging test suite minimization information was made. To the best of our knowledge, this has never been done before.

1.4 Results

The objective of automatic software testing is not only to detect faults, but also to provide valuable insights on where they are located - thanks to automatic fault localization techniques, such as spectrum based fault localization [CRPA12]. It has been proven before that test suite reduction can have an impact on the quality of the diagnostic reported by automatic test localization approaches [YJH08].

Multi-objective optimization has no theoretical limits on the number of objectives it can attend to [CS03], and it has been used before for test suite minimization [YNH11, YH10b, ISSM13]. However, until now and to the best of our knowledge, there was no multi-objective test suite minimization approach that takes into consideration an objective related with fault localization ability of the minimized test being able to diagnose faults.

Therefore, the use of a multi-objective test suite minimization approach that not only optimizes for the code coverage and time - like the common current approaches do - but also optimizes for coverage matrix density ($\rho$), is proposed.

It has been proven in the literature that coverage matrix density ($\rho$) is related with test suite diagnosability and it has an ideal value [GSGvG11]. Coverage matrix density was used with success as an objective function of an automatic test generation tool [CAFd13].
Introduction

We were able to use (\(\rho\)) as objective of a test suite minimization approach, creating a novel test suite minimization approach that not only reduces test suite execution cost (time, normally) while maximizing fault detection ability (code coverage, usually), but also optimizes for diagnosability.

In order to test if our approach performs better when used in conjunction with SBFL tools, A spectrum-based fault localization technique using Ochiai, one of best fault score competing methods [AZvG06] was implemented. Three minimization approaches were run on a set of real world software projects (described in section 5.1) with injected faults. The conclusion is that Ochiai was able to produce better rankings on the test suites generated with the proposed approach.

As referred, test suite minimization always involves two tasks: collecting minimization information (normally that includes code coverage per test case, test case execution time, etc.); and generating the minimized test suite solutions. Up until now, there was no public documented format for this information exchange that made it possible for modules created by different people to work together. Therefore, we proposed the use of a common JSON format and were, in fact, able to specify a JSON schema for it.

Large software projects are developed in a specific development environment. In our opinion, if test suite minimization is not integrated in the used environment, it is not going to be used at all. Thus, we suggest the use of extensions and plugins to integrate test minimization approaches on current development environments. We were able to create a Visual Studio extension that implements our noble minimization approach.

Most minimization approaches normally only work for a certain programming language or framework. Therefore, besides creating a Visual Studio extension, we also created a command line interface that allows our approach to work on any software project - independently of the programming language, provided the inputs are passed according to the proposed schema.

1.5 Document Structure

In order to make the reading and comprehension of this document easier, this section presents how the remainder of this document is structured.

Chapter 2 presents the current state of the art techniques for test suite reduction. Chapter 3 presents a new approach to test suite reduction able to attain the objectives described before. Chapter 4 presents an effective test suite minimization tool that implements our novel approach. Chapter 5 presents the validation techniques to apply to the proposed solution our the results obtained. Chapter 6 is a conclusion of this document.
Introduction
Chapter 2

State of The Art in Test Minimization

According to the literature, there are three ways of dealing with massive test suites: minimization, selection and prioritization. In this work, we use minimization which is one of those ways. Before continuing our work, it is important to discuss what are the alternatives proposed on the literature and how they differ from minimization. In the next section, all three techniques and their differences will be presented.

Although many test suite minimization approaches have been proposed, we are going to present only some of those. The chosen approaches may not be the most advanced or the most efficient ones. They were chosen because they are the ones that best represent the possible different ways of solving the test minimization problem.

In order to understand why test suite reduction is really needed, we are going to present a case study from Google. We will then present one of the few automatic easy-to-use test reduction tools that exist. We will conclude this chapter with a general overview of the actual state of test suite minimization techniques.

2.1 Minimization vs Selection and Prioritization

It is not possible to talk about test case minimization without talking about the related concepts test case selection and test case prioritization.

- **Test case minimization** selects a subset of tests that are not redundant between each other and still have a good fault detection ability. The minimization can be permanent (the redundant test are removed forever) or temporary. If it is temporary, it normally has a predefined lifetime that is not small. Test case minimization is not aware of the software changes applied at a given moment.
State of The Art in Test Minimization

- **Test case selection** is modification-aware, contrarily to test case minimization. Test case selection techniques try to identify the subset of tests that exercised the modified code in a given moment. It utilizes techniques such as *static analysis*.

- **Test case prioritization** tries to establish an order of test case execution that allows earlier fault detection. If a given test suite takes much time to run, programmers have to wait a lot to be sure that the software modifications are right. If there are tests that have more probability of detecting a fault than others, a prioritization technique will run those first, in order to be much faster and providing a useful feedback to the programmers.

These three techniques can be applied in scenarios where the test suites are too big and many times they are applied together.

Directly applying a selection or prioritization technique may be impractical because it could be too slow if all the tests are passed as input to these two techniques. Therefore, before applying these techniques, a minimization could be done.

The approaches used to solve each of these tree problems could also be exchanged between those techniques. For example, we could run a test minimization approach and establish a prioritization that runs the test cases that are part of the minimized solution first, and later the others that are not part of the minimized test suite solution, when there are resources available. A variation of this minimization-prioritization approach was used at Google as will be described in section 2.8.1. On the other hand, a prioritization technique that orders all the test cases could be used for minimization. The test suite is minimized to the first $N$ ordered test cases, according to the prioritization technique, that are able to run in an acceptable amount of time.

### 2.2 Classical Approaches

As mentioned, the code changes many times during the software development process. A system’s test suite tends to become big and many tests in the test suite become redundant. Redundant tests may have costs and do not increase the fault detection ability of a system. The aim of test reduction is to identify and remove those tests, temporarily or not.

In order to identify the redundant tests classical test suite minimization approaches, try identify the test suite subset with minimum cardinality that still exercises all the requirements on the test suite. Requirements may be lines of code, methods, classes or components.

So in classical approaches, test minimization is seen as a one objective problem.

In classical test suite minimization approaches, test suite minimization problem is formally specified as:

“Given: A test suite $T$, a set of test requirements $r_1, \ldots, r_n$, that must be satisfied to provide the desired testing coverage of the program, and subsets of $T$, $T_1, \ldots, T_n$, one associated with each of the $r_i$’s such that any one of the test cases $t_j$ belonging to $T_i$ can be used to test $r_i$’s.
Problem: Find a representative set, \( T' \), of test cases from \( T \) that satisfies all \( r_i \)’s.”

The set of requirements normally represents the lines of code of a program, making the minimization process select a subset of tests that still exercise/cover all lines of code of a program. It can also represent a function, for example, making tests still exercise all the functions of the system, or it can represent a high-level requisite of the system.

If we want to find a minimum set of tests from the test suite that still exercise all the requirements, that is equivalent to finding the minimal hitting set. As it has been explained by Shin Yoo et al:

“The testing criterion is satisfied when every testing requirement in \( r_1, \ldots, r_n \) is satisfied. A test requirement, \( r_i \), is satisfied by any test case, \( t_j \), that belongs to the \( T_i \), a subset of \( T \). Therefore, the representative set of test cases is the hitting set of the \( T_i \)’s.

Furthermore, in order to maximize the effect of the minimization, \( T' \) should be the minimal hitting set of the \( T_i \)’s” [YH10a].

The minimal hitting set is known to be an NP-complete problem since 1972. It is one of famous Karp’s 21 NP-complete problems [Kar72].

As on the classical approaches test minimization could be seen as the minimal hitting set problem. The classical approaches fall on two categories:

1. Use heuristics to get a sub-optimal answer to the problem, normally in polynomial time complexity.
2. Solve the minimal hitting set problem, and get an optimal answer. Always involving exponential time complexity.

On next chapter we are going to present heuristics based approaches. Then on the next two chapters, we present two exact answer approaches. One based in integer linear programming and another one based on constraint solving. Although exact answer approaches run in exponential sometimes they use so advanced solvers that for real-world problems are able to compete with heuristics. As happened with the constraint based approach we are going to present [CA13].

Besides the classical approach described, where test minimization is seen as a one objective problem, test minimization could be seen as a multi-objective problem. On the present document, a multi-objective approach will be proposed.
2.3 Heuristics

On section 2.2, it was said that test minimization problem in one objective could be seen as the minimum hitting set problem. Finding the minimal hitting set is an NP-complete problem.

One way to speed up the test minimization computing times was the use of heuristics. To understand how most heuristics work, it is necessary to understand these two concepts:

- **Essential test case** - An essential test case is a test case that, when removed, eliminates the possibility of exercising all the requisites.

- **Redundant test case** - A Redundant test is a test case that satisfies a subset of requirements satisfied by another test case.

<table>
<thead>
<tr>
<th>Test case</th>
<th>Testing Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>r₁</td>
</tr>
<tr>
<td>t₁</td>
<td>x</td>
</tr>
<tr>
<td>t₂</td>
<td></td>
</tr>
<tr>
<td>t₃</td>
<td></td>
</tr>
<tr>
<td>t₄</td>
<td></td>
</tr>
<tr>
<td>t₅</td>
<td></td>
</tr>
<tr>
<td>t₆</td>
<td></td>
</tr>
<tr>
<td>t₇</td>
<td></td>
</tr>
<tr>
<td>t₈</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: Example table representing test cases and the testing requirements they cover

To make the understanding of this important concepts easier, we make an analysis of the redundant and essential test cases of table 2.1. The essential test cases are \( t₁ \) and \( t₂ \) because requisites \( r₁ \) and \( r₂ \) are only covered by these tests, respectively. The redundant test cases are:

- \( t₄ \) and \( t₅ \) because the sets of requisites covered by both of them are contained in the set of requisites \( t₃ \) covers.

- \( t₆ \) because it only covers \( r₆ \) and \( r₆ \) is part of the requisites covered by both \( t₇ \) and \( t₈ \).

2.3.1 GE Heuristic

The GE Heuristic is a variation of the greedy algorithm. It works in two steps.

1. In the first step, the heuristic identifies the essential test cases. All the essential test cases will be part of the reduced solution.

2. In the second step, while the requirements are not covered by the set of the minimized test cases, the heuristic adds the test case that is not part of the minimized test cases and covers the maximum number of unsatisfied requisites to the solution set.
2.3.2 GRE Heuristic

The GRE heuristic is based on GE, only adding an extra step. At the start, the heuristic identifies the redundant test cases and discards them. After the first step, the GRE heuristic runs exactly the same steps as GE (without the removed tests).

According to the creators of GRE [CL98], the complexity of the worst case scenario of this heuristic is:

\[ O(\min(m,n)(m + n^2k)) \]

Where \( n \) represents the number of test cases, \( m \) represents the number (of) requisites and \( k \) is the number of requisites satisfied by the test case that satisfies more requisites.

There is no proof, until now, that GRE heuristic is better than GE or vice versa [YH10a].

2.4 ILP Model

Black et al proposed an innovative approach for test reduction. The authors modeled the problem as an integer linear programming [BMK04]. Given a set of test cases \( T = \{tc_1,\ldots,tc_n\} \) and a set of requirements \( R = \{r_1,\ldots,r_m\} \). The formulation of test reduction as an integer programming problem for the single-objective case is as follows:

\[
\text{Minimize} : \sum_{i=1}^{n} x_i \tag{2.1}
\]

\[
\text{Subject to : } \sum_{j=1}^{m} x_i a_{ij} \geq 1 \quad i = 1,\ldots,n \tag{2.2}
\]

Where \( x_i \) is a Boolean variable that represents if test case \( i \) is on the minimized set of tests or not, being \( a_{ij} \) equally a Boolean variable representing if the requirement \( j \) is exercised by test case \( i \).

Equation 2.1 tells us to minimize the summation of variables \( x_i \) given that \( x_i \) is one, if test case \( i \) is part of the minimized solution, and if not, is zero. In a non-formal way, what equation 2.1 tells us is to minimize the number of tests that are part of the minimized solution.

What equation 2.2 tells us is that all the requirements have to be exercised at least one time (\( \geq 1 \)).

This formulation guarantees that all the requirements are still covered by test cases and, contrarily to the heuristic approaches, guarantees that the test minimization solution has the minimum number of tests possible.

2.5 Constraint Satisfaction Approach

More recently, Campos et al. has proposed the use of a constraint solver to minimize test suites. This approach presents a collection of minimized solutions that can be further ordered and chosen...
State of The Art in Test Minimization

by the user, based on some criteria like the carnality of the solution. The minimized solutions
given by the constraint-based approach guarantee that all requirements are covered [CA13].

They make use of a coverage matrix. A coverage matrix is a binary matrix with \( N \) columns
and \( M \) rows, where \( N \) is the number of test cases and \( M \) the number of requirements. The contents
of the matrix on position \( a_{ij} \) represent if test case \( i \) covers the requirement \( j \) or not. A value of
one indicates that it covers the requisite and a value of zero indicates that it does not cover the
requisite.

\[
\begin{bmatrix}
  m_1 & m_2 & m_3 \\
  t_1 & 1 & 1 & 0 \\
  t_2 & 1 & 0 & 0 \\
  t_3 & 0 & 1 & 0 \\
  t_4 & 0 & 0 & 1 \\
\end{bmatrix}
\]

Figure 2.1: Example coverage matrix for the constraint-based approach as shown by Campos et

al.

Figure 2.1 shows an example of a coverage matrix. We can see that test case one covers
requirements one and two (represented by \( m_1, m_2 \)). Test case number two only exercises require-
ment one and test case number three exercises requirement two. Requirement three is covered by
test case number four.

The basis of this approach is that, in order to guarantee that all requirements are covered, we
have to grantee that:

- **For requirement one**: Test case one or test case two are run.

- **For requirement two**: Test case one or test case three are run.

- **For requirement three**: Test case four is run.

Given that on this constraint-based approach all the requirements have to be exercised, we can
create a big restriction that is the "logical and" between all the restrictions of each requirement.
They map the coverage matrix into a constraint following the logic mentioned.

After mapping the matrix into a constraint, they create a file describing the constraints and
then pass that file to MINION. MINION is an efficient and scalable constraint solving system
[GJM06].

Thanks to the correct formulation of the problem as a constraint solving one, and thanks to the
efficiency of MINION, according to the results published by the authors [CA13], this approach is
able to compete in terms of performance with the simply GRE heuristic and achieves a better test
reduction rate.


2.6 MOO Coverage and Cost

Treating the test case minimization problem as a multiple objective problem has many advantages. Although, most of the times, the problem is treated as a single-objective, there are also proposed some approaches in the literature that treat test minimization as a more than one objective problem.

One of this works is the one proposed by Yoo et al. [YH07]. Yoo et al. used the concept of Pareto efficient. The basic explanation of the concept for test minimization scenario is that, given a test case minimization solution \( S \), it is Pareto efficient if, for all measures/objectives used, there is no other solution presenting one or more objectives better than \( S \), without also presenting some other objective worse than \( S \). It is not possible to improve one objective of a Pareto efficient solution without negatively affecting another.

The authors were able to show the benefits of having a Pareto efficient solution [YH07]. They formulated the problem in the following way:

“Given: a test suite, \( T \), a vector of \( M \) objective functions, \( f_i, i = 1, ..., M \).

Problem: to find a subset of \( T \), \( T' \), such that \( T' \) is a Pareto optimal set with respect to the set of objective functions, \( f_i, i = 1, ..., M \).” [YH10b]

After formulating the problem, they proposed a two objective approach of finding the Pareto efficient solutions between requirement coverage and time to run tests.

They found the set of solutions where it is not possible to improve requirements coverage - without spending more time running the test cases - and where it is not possible to improve the time to run tests - without reducing requirement coverage.

They used the genetic algorithm NSGA-II described in chapter 3 to solve the problem.

2.7 Two Objectives ILP

This approach is the extension of the approach described in section 2.4, in order to allow it to find solutions taking into account two objectives: requirement coverage and a measure representing fault detection ability. The authors of this approach extended the formulation of the single-objective problem as follows:

\[
\text{Minimize : } \quad a \sum_{i=1}^{n} x_i - (1-a) \sum_{i=1}^{n} e_i x_i \\
\text{Subject to : } \quad \sum_{j=1}^{m} x_i a_{ij} \geq 1 \quad i = 1, ..., n
\]  

(2.3)  

(2.4)

Where, similarly to the formulation from section 2.4, \( x_i \) is a Boolean variable that represents if test case \( i \) is on the minimized set of tests or not and being \( a_{ij} \) equally a Boolean variable representing if the requirement \( j \) is exercised by test case \( i \). The variable \( e_i \) represents the fault detection ability of test case \( i \). The authors do not specify how this measure is computed. The variable \( a \) allows to...
configure how test minimization is done, making this approach one of the few in the literature that allows configuration.

If \( a = 1 \), no relevance is given to fault detection ability and we are, in fact, running exactly the single-objective algorithm specified in 2.4 that chooses the solution with minimum number of tests possible. If \( a = 0 \), no relevance is given to number of tests and the algorithm chooses the solution that maximizes the fault detection ability, being also single-objective.

If \( 0 < a < 1 \), we are running a two objectives algorithm, giving more relevance to the number of tests if \( a \) is closer to zero and more relevance to the fault detection ability if \( a \) is closer to one. Equation 2.4 is making a restriction on the possible values of the variables, meaning that all the requirements have to be exercised at least once \((... \geq 1)\). Making this approach force total requirement coverage, something that, in some cases, may not be necessary and be impractical.

2.8 Minimization in Practice

2.8.1 Scenario at Google

Google, the creator of the most used search engine on the planet, is a software-based company. For Google, if software fails, all the company fails with it. Thus, they follow good testing practices, in order to increase the faults detected before deploying.

The scenario for Google has some nuances. The software that Google produces does not have to be shipped to the client. The client does not have to download or install most of the Google projects. The software is made available to Google users without them noticing that there was an upgrade or something new was deployed. This generates a lot of pressure - they want to maintain their dominant market position, so they are always making new releases and improvements on their products.

By following good software testing practices, and given that they have many employees working there and some of the biggest software projects in the world, they end up with a giant test suite.

On table 2.2, extracted from “Faster Fault Finding at Google Using Multi Objective Regression Test Optimisation” [YNH11], it is presented a summary of the Google test suites. There we can see that Google has test suites that take 32 hours to run.

<table>
<thead>
<tr>
<th>Property</th>
<th>Minimum</th>
<th>Average</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test suite size</td>
<td>4</td>
<td>461</td>
<td>1,439</td>
</tr>
<tr>
<td>Test execution time (sec)</td>
<td>115</td>
<td>39,093</td>
<td>116,1321</td>
</tr>
</tbody>
</table>

Table 2.2: Summary of test suites at Google [YNH11]

Google uses massive parallelism in order to run the tests. The tests are distributed to many computing machines but, even so, they have projects with tests that take 32 hours to run and that
is impractical. After submitting a change to the repository, a programmer has to wait much time
until he is sure that the modification submitted is okay.

In order to transfer knowledge from research to the enterprise world, Yoo et al., the creators
of the Pareto efficiency approach described in section 2.6, implemented some test minimization

During this work, they noticed that the corporate world already has long date practices estab-
lished. In order to test minimization techniques be adapted by a big company like Google, the
techniques have to adapt themselves so as to be able to be established on the already implemented
software development process.

The authors did not remove the tests permanently. Google had already implemented a contin-
uous integration system that will eventually run all the tests. The authors used test minimization
techniques to select a minimized test suite that could run faster on a less powerful computing
system (maybe even in the developers computers).

During their work the authors concluded:

“Our experience also highlights the importance of a multi-objective approach: opti-
mizing for coverage and time alone is insufficient; we have, at least, to additionally
prioritize historical fault revelation” [YNH11].

Given that, as referred in the citation, the approach they had proposed before optimized for
time and coverage, they changed the approach they had. They implemented a three objectives
approach, taking into consideration time, coverage and historical fault detection.

The authors were able to achieve \(33\% - 82\%\) smaller test suites with the proposed minimiza-
tion approach. Moreover, this allowed an earlier fault detection on \(86\%\) of the cases.

The results achieved at Google are very promising. It will certainly increase awareness for test
suite minimization on other companies.

The authors had to create a new system for the Google case. Not all companies have the ability
to do that. The system used at Google has 3 specific objectives. These objectives may or may not
be adequate on other scenarios. This proves that, although scientifically we had algorithms and
test minimization solutions, some tuning was needed. If there was an easy-to-tune tool supporting
almost any criteria, maybe the adoption by companies would be easier.

2.8.2 RZoltar

Regression Zoltar (RZoltar) was created by Campos et al.. The tool uses the approach described
in section 2.5 by the same authors.

Regression Zoltar was integrated into GZoltar. GZoltar is a plugin that currently works in
eclipse for the java language. GZoltar is able to assist programmers in the debugging task by
providing innovative visualizations representing the calculations from Spectrum Based Fault Lo-
calization.

Regression Zoltar provides an easy-to-use interface integrated with the Eclipse IDE, as we can
see on figure 2.2.
The authors took a big step proving a practical tool that anyone can use, offering a great performance and good test minimization results, thanks to the constraint-based approach they used.

RZoltar forces all requirements to be covered, and takes only the requirements coverage criteria in consideration during the test minimization step. This solution may fit perfectly well some software projects. But for others, like the one at Google described in section 2.8.1, with really massive test suites, it may not be possible to achieve an acceptable test minimization, maintaining total requirement coverage. RZoltar is not currently able to give an answer to the needs of such cases.

2.9 General Overview

Heuristic approaches are the simpler ones to implement and have a good speed. The results are reasonable, but largely dependent on the specific problem. Heuristic approaches do not guarantee to find the optimal solution nor give an estimation of how far we are from the optimal result.

On the other hand, the integer linear programming based approach is able to get an optimum solution. In an experimental study by Zhong et al. [ZZM08], the integer linear programming was able to perform at almost the same speed as the simple GRE heuristic described in 2.3.2. When compared with more recent and more complex heuristics, integer linear programming was not able to achieve the same speed.

The constraint-based approach is the most recent one and, contrarily to the other approaches, produces more than one solution that can further be ordered by other criteria.

Pareto efficiency between code coverage and cost approach has the big advantage of not forcing total coverage, which on some big software projects is necessary. Although the approach does not try to produce test suites that are good at localizing faults.

The two objectives ILP model has the big advantage of allowing the configuration of the relevance of each objective. Although the proposed method is not general and only uses two objectives, the objectives proposed are requirements coverage and fault detection ability. In their
work, the authors do not specify how fault detection ability is computed. Which is understandable, considering that predicting the ability of a test case to detect faults is a very hard task to do.

Practical adoption in the industry is still limited. This can be explained by the lack of practical tools.

Currently creating a practical tool involves understanding how test minimization approaches work and learning how to implement them, or understanding how to integrate with them. Researching a test minimization approach involves creating a test information collector, in order to be able to validate the research with a real software project. This happens because there is currently no standard or propagated practice for test suite minimization information exchange.
State of The Art in Test Minimization
Chapter 3

A MOO Test Suite Minimization Approach

Yoo et al. proposed a multi-objective approach using Pareto efficiency [YH10c]. The approach used two objectives: time and requirements coverage. Although this approach may give acceptable results in terms of creating a minimized test suite that runs in an acceptable amount of time and has still an acceptable fault detection capability, this approach does not explicitly take into consideration fault location ability of the minimized test suite.

Automated tests are not only useful in detection software faults - they could also be useful in locating those faults on the source code. The location of those faults could be done manually (e.g. inspecting the code exercised by the failing test cases) or automatically, for example, by using spectrum based fault localization as in [JAG09, JH05, CRPA12].

As referred in the state of the art analysis, most of the proposed approaches were single-objective. Also, even though Yoo et al. showed the advantages of using a multi-objective approach and were even able to test a multi-objective approach with success on an industrial scale software project at Google, there is not yet a multi-objective approach that tries to give some relevance to test suites that optimize fault localization ability.

On the scientific community there is much research being done about techniques that automatically locate faults on the source code, using requirement coverage information of each test case of the test suite and the pass or fail information as input. There have even been proposed practical tools for the most common IDE’s used nowadays.

Given this scenario, it is expected that the adoption of automatic fault localization tools will grow. Test suite minimization discards test cases of the original bigger test suite. The test cases discarded could be the ones that are best suited to help automatic test localization tools locate faults in the source code. Therefore, to say the least, test suite minimization could make automatic fault localization tools useless.

Given that on multi-objective approaches it is possible to take many objectives into account for the minimization process, we think it is necessary to explicitly take into consideration an objective
that is related with the efficiency of automatic fault localization approaches, using the minimized test suite.

To the best of our knowledge, until now there is no multi-objective test suite minimization approach that explicitly takes into consideration an objective that is related with automatic fault localization efficiency. The efficiency of using an automatic fault localization approach on a given test suite has been studied before. Gonzalez et al., proposed a metric ($\rho$) [GSGvG11] that, given a test suite and the requirement coverage for each test case, has an ideal value for efficiency in locating faults. When that metric is more distant from its ideal value, the efficiency decreases. We think this metric could be used as an objective of a multi-objective minimization approach.

In this chapter we present our multi-objective test reduction approach that uses requirement coverage, test execution cost and $\rho$.

At the beginning of the chapter, we will describe each of the objectives used on our approach and its relevance on a test minimization scenario. Then we will describe both what is a multi-objective optimization problem and a test minimization problem and how they could be related. After that description, we will report the inputs of our algorithm and then briefly describe the algorithm used. Since the objectives used need to be computed thousands of times during the final solution computation, the algorithms used to compute each of the objectives need to be as fast as possible. Thus, after the algorithm description, we will explain our approach for the computation of each objective.

3.1 Objectives

3.1.1 Requirement Coverage ($Cov$)

Requirement coverage is a metric of how much the test suite exercises the software under test. A requirement could be a function, a class, a high level functionality or a line of code. The last one is commonly called "code coverage". We used code coverage on our prototype. In order to get the code coverage information of each test case, a software component was developed.

The requirement coverage metric could be defined as:

$$\text{RequirementCoverage} = \frac{\text{NumberOfExercisedRequirements}}{\text{NumberOfRequirements}} \quad (3.1)$$

The requirement coverage objective is directly related with the capacity of the test suite in detecting faults on the software under test. If a line of code contains a bug that affects the correct function of the software, and that line of code is not executed by the test suite, that bug will not be detected. If, on the contrary, that line of code is executed by the test suite, that bug could be detected.

During the test minimization process, some tests will be discarded. Some requirements that were covered before could, therefore, be uncovered after the minimization process. Most of the times, a requirement is not only covered by a test case, but rather by more than one. This means it
is sometimes possible to minimize the test suite without affecting the requirement coverage. That is what the classical single-objective approach does - find a smaller test suite that still covers the same requirements.

Although ideally we wanted the same value of requirement coverage as the original test suite, sometimes it is not possible to achieve the desired amount of minimization maintaining the same requirement coverage as the original test suite. One example of those cases happened at Google and was detailed in section 2.8.1.

Our approach tries to maximize the value of the requirement coverage metric. It did not force the same requirement coverage as the original test suite. After the minimization, the user can select solutions that have the desired requirement coverage and that are acceptable to his scenario, according to the other objectives. If the user so desires, they could choose the solutions with the same requirement coverage as the original one. However, if it is not possible for their case, when, for example, solutions with the same coverage take a great amount of time to run, the user could choose one where the coverage is reduced.

### 3.1.2 Execution Cost (Cost)

Execution cost is a metric of how much a test suite costs to run. Its value for "cost" could be different from project to project. For example, on a limited hardware, we might give much relevance to the memory consumption of the test suite execution and, on another scenario, the most relevant cost could be the time required to run the test suites or even a combination of a set of values.

On our work, we used the execution time of the test suite as the cost. This could be easily changed to another value, by passing a different cost vector described in more detail in section 3.3.2.

Given a test suite and given a cost for each test case that is part of the test suite, the execution cost of the test suite could be computed as the sum of the costs of each test that is part of the test suite.

On our approach, we try to minimize the execution cost of the minimized test suite.

### 3.1.3 Coverage Matrix Density ($\rho$)

Besides being used to detect faults on the software, software testing is also useful on localizing the region on the source code where those faults are located. This type of localization made by using the testing information could be done in two ways:

1. **Manually** - The programmer inspects the source code regions associated with a failing test case find the bug. For example, if each test case executes one function and one test case fails, there is a significant probability that the bug is in the function called by the failing test case.
2. **Automatically** - Using an automated tool that, having had access to code coverage information of each test case and the pass-or-fail information of the test suite, generates an error score for each component of the program (e.g. line of code).

Although the use of automatic fault detection tools is not yet a common practice among the software industry, fault localization techniques are continuously being researched by the scientific community and some usable practical tools have been created, namely:

- **GZoltar** - A plugin for the eclipse IDE that works with the java programming language. Besides computing a fault score for each software component, this plugin also offers innovative visualizations in order to help the programmer locate the faulty component faster [CRPA12].

- **FLAVS** - An extension for the Visual Studio IDE from Microsoft that computes an error score for each component of a C# computer program. Besides offering some of the formulas for error score computation proposed by the scientific community, the extension also allows the programmer to specify a custom formula [WZZC15].

Since practical tools are being developed and automated fault localization techniques can provide useful information to the programmers, it is expected that the use of automated fault localization tools will increase in the next years.

Automated fault detection tools make use of code coverage and test pass or fail information. During the minimization process, some test cases are discarded, which means the input information of automatic fault localization tools is changed. This change of input data could influence the results generated by the tools.

Until now, most test suite minimization approaches focused on test execution costs and requirement coverage in order to maximize fault detection ability, but the ability of the minimized test suite in helping to locate software faults was not taken into consideration in the big majority of the cases.

The \( \rho \) metric represents the density of the coverage matrix of the minimized test suite. A coverage matrix represents the relation between test cases and the requisites they cover. Our algorithm uses a coverage matrix as input, so coverage matrix is further defined on section 3.3.1.

It has been shown by Gonzalez *et al.* that under certain assumptions, “matrices with \( \rho = 0.5 \) provide ideal efficiency. Increasing or decreasing \( \rho \) will cause a reduced efficiency. The density of a matrix cannot be arbitrarily low or high without compromising its diagnosability (the ability to uniquely distinguish all faults)” [GSGvG11].

Thus, in order to increase the efficiency of the minimized test suite in locating software bugs on our multi-objective approach, we try to minimize the difference of \( \rho \) to 0.5.

Coverage Matrix Density \( \rho \) has been used before as a fitness function to guide an automatic test suite generator (EvoSuite [FA11]). The authors were able to automatically generate test cases that improved the ability of the test suite, with the new generated test cases locating faults on the software [CAFd13].
A MOO Test Suite Minimization Approach

We think that for the test suite minimization if $\rho$ is taken in consideration we are also able to improve the diagnosability of the minimized test suites, similarly to what happened when the test generation.

Details about $\rho$ formula and its computation will be provided on section 3.5.3

### 3.2 Problem definition

We have three objectives in consideration that need to be optimized at the same time. Our objectives could be in conflict. One example of a possible conflict is, requirement coverage clashing with the test execution cost - in most cases, covering more requirements involves executing more tests and executing more tests involves more costs.

For instance, the same situation happens when buying a car. A customer could want to minimize the cost, minimize fuel consumption and maximize speed. There are three objectives that could conflict with each other, like speed and cost.

On both situations the objectives could be conflicting and, therefore, it is not possible to identify a single optimal solution. However, it is possible to find a set of solutions equally good, without further external information. Both our test minimization approach and the hypothetical car problem are examples of multi-objective optimization problems.

As for the car issue, a customer wants to find the set of solutions (possible cars to buy) where:

1. It is not possible to have a cheaper car without decreasing speed or increasing the fuel consumption.
2. It is not possible to have higher speed without increasing the price of the car or increasing the fuel consumption.
3. It is not possible to have lower fuel consumption without increasing the price of the car or decreasing speed.

Multi-objective optimization problems are characterized by a set of variables, a set of objective functions possibly conflicting with each other, and the aim is to find the set of solutions where it is not possible to get an objective better without getting another one worse. Multi-objective optimization is going to be formally defined on the next subsection.

#### 3.2.1 MOO Formal Definition

For a problem with $m$ decision variables $x_1,...,x_m$, and $n$ objectives $y_1,...,y_n$, multi-objective optimization is defined as follows:

Minimize:

$$\hat{Y} = f(\hat{X})$$  

(3.2)
A MOO Test Suite Minimization Approach

Where:

\[ f(\vec{X}) = (f_1(x_1, \ldots, x_m), \ldots, f_n(x_1, \ldots, x_m)) \]  

(3.3)

\[ \vec{X} = [x_1, \ldots, x_m]^T \in P \]  

(3.4)

\[ \vec{Y} = [y_1, \ldots, y_n]^T \in O \]  

(3.5)

\( \vec{X} \) is the decision vector, it contains the values of the decision variables (variables that the problem wants to optimize), \( P \) is a set of all possible decision vectors (all possible decisions that could be made), the cardinality of \( P \) depends on the constraints between the variables and is very problem-specific. \( \vec{Y} \) is a vector representing the values of each objective and \( O \) is the set of all possible objective vectors.

A decision vector \( \vec{X}_1 \) dominates (is better than) another decision vector \( \vec{X}_2 \) if, and only if:

1. The decision vector \( \vec{X}_1 \) is better or equal than \( \vec{X}_2 \) for all objectives:

\[ f_i(\vec{X}_1) \leq f_i(\vec{X}_2) \forall i \in 1, \ldots, n \]  

(3.6)

2. The decision vector \( \vec{X}_1 \) is better than \( \vec{X}_2 \) for at least one objective:

\[ \exists i \in 1, \ldots, n \mid f_i(\vec{X}_1) < f_i(\vec{X}_2) \]  

(3.7)

A decision vector \( \vec{X}_1 \) is Pareto-Optimal if there is no decision vector \( \vec{X}_2 \) that dominates \( \vec{X}_1 \):

\[ \exists \vec{X}_2 \in P \mid \vec{X}_2 \text{ dominates } \vec{X}_1 \]  

(3.8)

The aim of a multi-objective optimization solver is to find the set of all the Pareto-Optimal decision vectors, called Pareto frontier or Pareto front.

### 3.2.2 Test Suite Minimization as a MOO Problem

Now that we have defined the general multi-objective optimization problem, we will specify the modelation of our test suite minimization approach as a multi-objective optimization problem.

Initially, we have a bigger test suite and the aim is to find subsets of that test suite that are Pareto Optimum, based on our triplet of objectives \( \langle Cov, Cost, \rho \rangle \).

Knowing the tests that are part of a given subset of the original test suite, one could evaluate each of the objectives. The evaluation of the objectives is offered in subsection 3.5. It is impossible to encode the tests that are part of a minimized test suite as a vector of decision variables, the chosen encoding for the decision vector described in 3.4.1. In this way, we have a set of decision variables, representing a subset of tests of the original test suite, and we have a set of objective
functions that, given the subset of tests, can generate a value for the objective they compute. Thus, we are, in fact, in the presence of a multi-objective optimization problem, more formally defined as:

Given: a test suite, $T$ find all $T' \in \mathcal{P}(T)$ such that $T'$ is Pareto Optimum according to the triple of objectives $(Cov, Cost, \rho)$.

In other words, we want to find all the subsets of $T$ where:

1. It is not possible to have a better requirement coverage (code coverage), without having a higher execution cost (spending more time) or having a $\rho$ more distant from 0.5 (lowering the fault localization efficiency).
2. It is not possible to have a lower test execution cost, without having a lower requirement coverage or $\rho$ more distant from 0.5.
3. It is not possible to have $\rho$ closer to 0.5, without having a lower requirement coverage or a higher execution cost.

One theoretical algorithm for solving this problem is to try every possible subset of $T$, evaluate its triplet of objectives, and then return only the ones that are Pareto optimum. But given that $|\mathcal{P}(T)| = 2^{|T|}$ for a test suite of 1000 test cases - which is not that big in industrial projects - we would be talking about $2^{1000}$ possible subsets, which is not possible to evaluate in light of present computation power. A more efficient algorithm from the scientific literature had to be chosen.

### 3.3 Algorithm Inputs

Before going in further detail about how our minimization system works, it is important to formally define what are the inputs of the minimization algorithm.

#### 3.3.1 Coverage Matrix

As on the approach proposed by [CA13], we make use of a coverage matrix. As described on chapter 2.5, a coverage matrix is a binary matrix with $N$ columns and $M$ rows, where $N$ is the number of test cases, and $M$ the number of requirements. The contents of the matrix on position $a_{ij}$ represent if test case $i$ covers the requirement $j$ or not. A value of one indicates that it covers the requirement and a value of zero indicates that it does not cover the requirement.

Figure 3.1 shows an example of a coverage matrix. We can see that test case one covers requirements one and two (represented by $m_1, m_2$). Test case number two only exercises requirement one, and test case number three exercises requirement two. Requirement three is covered by test case number four.

A requirement could be a function, making the coverage matrix represent the test cases and the functions they call, a class or a human-given information (like the high level software feature), making the matrix represent test cases and the features they test. On our prototype, a requirement
A MOO Test Suite Minimization Approach

A coverage matrix is a boolean matrix. Computationally, it could be represented as a vector of BitSets, binary flags or another equivalent bit container. Each position of the vector contains a binary container representing the requirements covered by that test case. This possibility of representation is very efficient in space, making the matrix occupy just $N_{\text{Tests}} \times N_{\text{Requirements}}$ bits in memory. For a very extensive software, with one hundred thousand requirements and one thousand test cases, the space occupied by the matrix is $100000 \times 10000 \approx 11.92$ Megabytes, which is a really small value for memory utilization, considering a hypothetical extensive software like the one referred.

### 3.3.2 Test Execution Cost Vector

The test execution cost vector is a $N$ dimension vector, where $N$ is equal to the number of test cases. Each position of the vector contains a value representing the cost of running the test case associated with that position.

Our algorithm tries to minimize the total cost of the suite. The cost could be memory consumption of the test, CPU utilization, or any other value or combination values. The only requisite of the algorithm is that the cost of each test is independent, no matter what other tests might be run. So if we run test case one and test case two, the cost of that execution is the sum of the cost of test case one with test case two. If all the tests are run, the cost needs to be the summatory of all the value in the vector. In our work, we used the execution time of a test in seconds.

### 3.4 Algorithm Description

In this section, we will detail the approach used to solve our multi-objective optimization problem of minimizing a test suite, taking into account the triplet of objectives $(\text{Cov}, \text{Cost}, \rho)$.

To solve this problem, we chose the NSGA-II algorithm. NSGA-II is a generic algorithm. Genetic algorithms use a set of variables, called genes, and simulate a biological process with a view to optimize the "genes" of the population.

In order to be able to run the NSGA-II algorithm, or any other multi-objective optimization solver, we first have to specify our decision vector. To begin with, we detail our decision vector.

![Coverage matrix](image)
and how the population is represented for the genetic algorithm, and then we briefly present how
NSGA-II works.

### 3.4.1 Decision Vector

As referred on section 3.2, for it to be possible to solve a test minimization problem as a multi-
objective problem, we need to model a subset of the test suite as a set of decision variables that
we want to optimize. This set of decision variables is also going to be the "gene" of the individual
used by NSGA-II.

So as to have our decision variables, we used the idea that, for each test case of the original
test suite, we have to take a decision, either that test case is part of the minimized test suite or it is
not. So, for each test case, we can have a Boolean decision variable that represents if the test case
associated with it is part of the minimized test suite or not.

\[
\text{DecisionVector} = [\text{TestCase}_1 \ \text{TestCase}_2 \ \ldots \ \text{TestCase}_N]
\]  

(3.9)

We can see in 3.9 an example decision vector for an original test suite of N test cases. If
test case one is included in the minimized test suite, the variable \text{TestCase}_1 will take the value of
true; if it is not, the variable will take the value of false. This is the same for the \text{TestCase}_2 until
\text{TestCase}_N.

For a test suite T with N test cases, the decision vector is composed of N variables with each
one being able to take two values, true or false. We have \(2^N\) possible distinct decision vectors. The

cardinality of the power set of T is also \(2^N\), so for each possible subset of the original test suite,
we have an unique decision vector that specifically identifies it.

### 3.4.2 NSGA-II

NSGA-II is a genetic algorithm used to solve multi-objective optimization problems, proposed
by Deb et al. [DPAM02]. Generically speaking, genetic algorithms are normally well-suited for
multi-objective optimization because genetic algorithms are based on biological processes, and
biological processes are normally multi-objective. NSGA-II has been used before for test suite
minimization and has obtained good results [YH10c, AA14].

In our approach, we also used NSGA-II. In this section, we are going to present briefly how
NSGA-II works.

NSGA-II works according to the following steps:

1. Initialize a population of size N. The algorithm allows a dynamic size - for example, one
may want the population to have 50 individuals. It randomly initiates the individuals - in
our case it creates N random possible decision vectors, that represent N possible minimized
test suites.

2. For each individual of the population, the algorithm generates the value of the objectives. In
our case, the algorithm calls the evaluation functions that will be described in section 3.5.
3. Sort/Rank the population based on non-domination relation. Domination relationships were defined in section 3.2.1. For each solution (minimized test suite), the algorithm assigns a fitness function that represents how much that solution is not dominated.

4. Using selection, crossover and mutation, a new child population (normally of the same size as the parent one) is created.

5. Parent and child population are mixed, and best N individuals are selected, based on non-domination, ranking and crowding distance. Crowding distance is essential to maintain a diversified Pareto front.

6. If the stop condition is met, the algorithm stops and the returned Pareto front is the last population. If the stop condition is not met, the process is repeated. Normally, a fixed number of repetitions of the algorithm is used as stop condition.

To better understand NSGA-II, in figure 3.2 we present a flow chart of the algorithm created by Tushar Goel.

![Flow chart of NSGA-II by Tushar Goel](image)

**Figure 3.2: Flow chart of NSGA-II by Tushar Goel**

### 3.5 Objectives evaluation

During the execution of the NSGA-II algorithm, for each new individual of the population, we have to compute the value of its objective function. Given that, during the runtime of the algorithm, thousands of individuals were created, the evaluation functions are called thousands of times. Thus, in order for the minimization algorithm to have an acceptable computation, it is necessary that the evaluation functions are efficient.
Accordingly, during the implementation of the evaluation functions, we took special care and tried to find efficient solutions.

### 3.5.1 Requirement Coverage \((Cov)\)

The requirement coverage metric is defined as:

\[
RequirementCoverage = \frac{\text{NumberOfExercisedRequirements}}{\text{NumberOfRequirements}}
\]  

The number of requirements is independent of the minimized test suite. It is a constant during the minimization of a given software program. The number of exercised requirements is dependent on the minimized test suite. One simple possible way of computing this value would be algorithm

```
input : CoverageMatrix, MinimizedTestSuiteVector, NumberOfRequirements, NumberOfTests
output: Number of requirements covered by minimized test suite
requirementsCovered = 0;
  /* foreach requirement */
  for requirementN = 0 to NumberOfRequirements do
    /* foreach test case */
    for testN = 0 to NumberOfTests do
      /* if testN is on minimized test suite and covers RequirementN */
      if MinimizedTestSuiteVector[testN] ∧ CoverageMatrix[testN][requirementN] then
        requirementsCovered++;
        break;
      end
    end
  end
return requirementsCovered;
```

**Algorithm 1:** Simple number of requirements covered computation

Although the approach referred on algorithm 1 is very simple and intuitive to understand, it involves iterating over each requirement and, inside each requirement, iterating over each test case. This approach does not take advantage of the fact that both coverage matrix and minimized test suite vector are just vector(s) of bits. Computers are very good in making bit operations - their...
processors are optimized for that, so we proposed algorithm 2.

```plaintext
input : CoverageMatrix, MinimizedTestSuiteVector, NumberOfTests
output: Number of requirements covered by minimized test suite

requirementsCoverageByMinimizedTests = coverageMatrixLineOfZeros;

/* foreach test case */
for testN = 0 to NumberOfTests do
    if MinimizedTestSuiteVector[testN] then
        /* merge the coverage testN with the coverage of the other minimized tests */
        requirementsCoverageByMinimizedTests |= CoverageMatrix[testN];
        break;
    end
end

return GetCardinality(requirementsCoverageByMinimizedTests);
```

**Algorithm 2:** Our approach to compute the number of requirements covered by the minimized test suite

The basis of algorithm 2 is that, if we make a logical operation between line number X and line number Y of the coverage matrix, we get a binary flag that represents the requirements covered by test case number X or test case number Y. What we are doing in algorithm 2 is computing a binary flag that represents the requirements covered by any test case that is part of the minimized test suite, and then returning the cardinality of the bits set to true. That is the number of requirements covered by the minimized test suite.

Processors are reality fast in making logical operations like that or giving a good performance to this approach. To get cardinality of the bits set, we used a method found on the Bit Twiddling Hacks page by Sean Eron Anderson, of Stanford university. This method does not involve iterating of a set of bits bit by bit, and showed itself very efficient on many tests.

### 3.5.2 Execution Cost (Cost)

Since we have a vector with the execution cost of each test case as input, computing the total execution cost of the minimized test suite is a simple process. We have only to add up the execution cost of each test case that is part of the minimized test suite, as we can see on algorithm 3.
input : MinimizedTestSuiteVector, ExecutionCostVector, NumberOfTests

output: Execution cost of the minimized test suite

executionCost = 0;

/* foreach test case */

for testN = 0 to NumberOfTests do
  /* if testN is on minimized test suite */
  if MinimizedTestSuiteVector[testN] then
    executionCost += ExecutionCostVector[testN];
    break;
  end
end

return executionCost;

Algorithm 3: Minimized test suite execution cost computation

3.5.3 Minimized Coverage Matrix density ($\rho$)

As referred on subsection 3.1.3, $\rho$ is a metric that represents the density of the coverage matrix of a test suite. When $\rho = 0.5$, under certain assumptions, the test suite provides ideal fault localization efficiency, when using spectrum based fault localization. On our approach, we try to minimize the difference of $\rho$ to 0.5:

$$\text{Min}|0.5 - \rho|$$

(3.12)

Given a test suite $T$, with $N$ test cases and $M$ requirements, a Boolean coverage matrix $C$, and interpreting true as one and false as zero, $\rho$ could be computed as:

$$\rho = \frac{\sum_{i=1}^{N} \sum_{j=1}^{M} C_{ij}}{N \times M}$$

(3.13)

The coverage matrix of a minimized test suite is different from the coverage matrix of the original test suite. We are removing test cases - as a test case is represented as a line in the coverage matrix, one approach for computing the $\rho$ would be to ignore lines of the matrix associated with discarded test cases. This approach is described on algorithm 4.

Although algorithm 4 is simple to understand and does not involve the creation of a new coverage matrix for each minimized test suite - which would be very computationally expensive - that algorithm involves iteration over each test case and, for each test, iterating over each requirement. Given that $\rho$ evaluation is used inside NSGA-II, this evaluation is called many thousands of times. Any improvement on $\rho$ will provide a notable improvement in the overall minimization process.

No matter what the minimized test suite is, if it covers a given test case, the requirements covered by that test case are the same as the requirements covered by the same test case on the
A MOO Test Suite Minimization Approach

Algorithm 4: Simple UN-optimized method to compute the $\rho$ of a minimized test suite

original test suite. Therefore, in order to avoid iterating over each requirement, we could pre-compute the requirements covered by each test case. Besides algorithm 4 not taking advantage of the fact that coverage matrix is vector of boolean flags, to compute the number of trues on a given matrix line, it is not needed to iterate over each line element. We could use efficient methods from the literature that count the bits equal to true without iterating over each bit. Based on this, we propose algorithm 5, which pre-computes for each test case the number of requirements it covers, and algorithm 6, which computes the $\rho$ of a minimized test suite in an efficient way.

Algorithm 5: Pre-computation of the number of requirements covered by each test case

The $GetCardinality$ function used on algorithm 5, as in the requirement coverage evaluation, is a function to get the number of bits equal to true, found on the Bit Twiddling Hacks page by Sean Eron Anderson, of Stanford University. This method does not involve iterating over each bit.
Using the pre-computed vector of requirements per test case, algorithm 6 computes the $\rho$ value of a minimized test suite.

```
input : requirementsPerTest, NumberOfTests, NumberOfRequirements, MinimizedTestSuiteVector
output: $\rho$ value

summatory = 0;
numberOfMinizedTests = 0;
requirementsPerTest[NumberOfTests];
/* foreach test case */
for testN = 0 to NumberOfTests do
    /* if testN is on minimized test suite */
    if MinimizedTestSuiteVector[testN] then
        summatory += requirementsPerTest[testN];
        numberOfMinizedTests++;
    end
end
$\rho$ = $\frac{\text{summatory}}{\text{numberOfMinizedTests} \times \text{NumberOfRequirements}}$;
return $\rho$;
```

**Algorithm 6:** Optimized computation of the $\rho$ value of a test suite
Chapter 4

An Effective Minimization Tool

Although there is much research being done, a study indicates that only 8% of the published research presents techniques with possible application on the software industry.[YH10a]. There is a need for test suite minimization techniques in the industry. There are more cases identical to the one at Google described in section 2.8.1. The industry has a need for test suite minimization solutions, but they are not yet being widely applied.

One of the main reasons for this is the lack of practical tools.

"Without readily available tools that implement regression testing techniques, practical adoption will remain limited. One potential difficulty of providing tool support is the fact that, unlike unit testing for which there exists a series of frameworks based on the xUnit architecture, there is not a common framework for the regression testing process in general."[YH10a]

Regression Zoltar, referred in detail in section 2.8.2, took a big step in proving to be a usable tool that can, in fact, be applied to some software projects. Regression Zoltar forces total requirement coverage, which is okay in some cases. But in cases with the dimension of the one reported at Google, it may not be possible to achieve a good enough test reduction, maintaining total requirement coverage.

Yoo et al. proposed a multi-objective approach using Pareto efficiency [YH10c]. The approach proposed used two objectives: time and requirements’ coverage. When implementing it at Google, they concluded:“Our experience also highlights the importance of a multi-objective approach...”[YNH11]

In this chapter, we will describe an effective multi-objective test suite minimization tool. Our tool consists of an integrated Visual Studio extension, with a graphical user interface that is able to minimize the test suites of projects, using Visual Studio in an user-friendly way. The tool observes our multi-objective approach described in chapter 3, to a Pareto front using the triplet of objectives \((\text{Cov}, \text{Cost}, \rho)\). Then, the tool presents to the users the solutions on the Pareto fronts, allowing them to choose the best one according to their needs.

Visual Studio is not one of the most used IDE’s in the world. It is used in big industrial projects, where test minimization is probably a must, as Microsoft recommends, and, to the best
of our knowledge, there is no public automatic test suite minimization system for Visual Studio. Our tool will be the first one.

Besides the Visual Studio extension, we also implemented a command line version of our approach that is able to run with any software project, provided the input described on section 3.3.1 is passed using the correct JSON format.

In the next sections, we will specify the independent components that are part of our solution. We proposed a "standard" way for the communication of test suite minimization information. We were able to test our "standardized" communication in the modules we implemented. We are going to present our standardized proposal. Afterwards, we will present some use cases of our test minimization tool and its user interface. We will conclude by presenting a final discussion about our tool.

4.1 Components

The creation of our effective test suite minimization tool was a complex process. The most complete common execution pattern is identical to this one:

1. Handle an event that triggers minimization process.
2. Collect the input information. This involves getting per test case code coverage information, and test execution time from Visual Studio projects.
3. Use all the collected information to model a multi-objective optimization problem.
4. Use the approach explained in chapter 3 to minimize the test suite.
5. Present the Pareto front of minimized solutions an user-friendly way. For each solution, show its objective values and the test cases it contains.
6. Allow the user to select the best solution(s), according to their needs.
7. Represent the solution(s) chosen in a special format that Visual Studio can read, so it will know that, next time, it should only run tests that are part of the minimized solution.

As we can see, there are many complex tasks to undertake. Most of the task are independent from each other. They depend only on their input and, with that, they produce an output that passes to the next task. As we have many independents and we used the best software engineering practices, we end up dividing our solution in many different projects independent of each other. In the next subsections, we are going to describe each of the modules.

Excluding module "Graphical User Interface", that was implemented in web technologies (HTML, CSS & JavaScript), all the other modules were implemented using the c# programming language.
4.1.1 Inputs Collector

The objective of the inputs collector module is to get the input data described in section 3.3. In order to get a coverage matrix, it needs to collect the coverage information per test case. To determine the cost vector, it needs to collect the cost (execution time) of each test case.

We tried every imaginable solution and spoke with people from Microsoft, but, as far as we know, there are no API’s provided by Microsoft to get code coverage per test case. It is only possible to get code coverage from all the tests executed.

We need coverage per test case, not only to implement our approach, but also to validate it. Without coverage per test case, we could not run an automatic fault localization approach like SBFL, which we used for validation.

Microsoft has a tool that allows us to execute just the tests we want, even if it is just one. That tool can also provide the code coverage of the tests executed, the execution time and other information.

We tried an indirect approach to get code coverage per test case. We used the test execution tool from Microsoft to run one test case each time in a cycle. After the execution, we collected the output and parsed the execution time and the location of a structure that contains the coverage of
that test case in a binary representation from Microsoft.

The only disadvantage in using a test execution tool to run one test each time, in order to
get code coverage per test case, is that starting a test execution takes some time and, with this
approach, we need to do it many times. However, as far as we know, this was the only possibility.
Besides, minimization is a rarely done task and it is not time-sensitive to find a minimized solution.

We used Microsoft API's to interpret the code coverage structures. For each component, we
need to know the test cases that exercised that same component. We have many different coverage
structures (one for each test case), but the structures are not the same. The identifiers of the
components are also not the same on each structure. And we need to know if a given component
(line of code), covered on test case A, is the same component covered in test case B, but with
a different identifier. So as to relate the components across the different coverage structures, we
used the pair (source file, line number) to uniquely identify a component. With this process, we
managed to generate a coverage matrix.

After the collection of the input information, we represented it according to a JSON structure,
which will be described further on. If, after the collection of inputs, we want to continue the
minimization process and we do not want to save this information for later use, that information
is passed directly in-memory, using a c# equivalent structure for efficiency reasons.

4.1.2 Test Execution Logger

After the test minimization process, we need to make Visual Studio only run the tests that are part
of the minimized test suite. To do this, we need a special identifier for each test case. After the
minimization process, we pass a list containing all the identifiers of test cases that are part of the
minimized test suite.

The name Microsoft gives to the identifier is "Fully Qualified Name". This identifier is not
more than a string containing the name-space, the class and the method where the test case is
implemented. By default, the test execution engine that we use during the input collecting process,
described in the last subsection, provides the test execution time, the pass or fail information, the
code coverage of the tests and the test name, which is just the method name. It does not provide the
"Fully Qualified Name" that is necessary later on to pass the minimized test suite back to Visual
Studio.

For it to be possible for us to collect the test case "Fully Qualified Name", during the inputs
collecting process, we write a small extension for the Microsoft test execution system, which,
besides the information it already provided, also gives us the fully qualified name of the test cases.

4.1.3 Minimization System

The minimization system is where our minimization approach described in chapter 3 is imple-
mented.
The minimization component receives the inputs described in section 3.3 in a format that will be specified and, using that inputs’ model, a MOO problem solves it and produces an output that represents the minimized test suite Pareto front, according to our objectives.

Given that the genetic algorithm NSGA-II that we chose to solve the MOO problem was already implemented on some public available libraries, it would not make sense for us to be implementing it again.

We checked the public available libraries that implement NSGA-II in c# and decided to use JMetal.NET. According to the authors:

“JMetal, an object-oriented Java-based framework aimed at the development, experimentation, and study of metaheuristics for solving multi-objective optimization problems. JMetal includes a number of classic and modern state-of-the-art optimizers, a wide set of benchmark problems, and a set of well-known quality indicators to assess the performance of the algorithms. The framework also provides support to carry out full experimental studies, which can be configured and executed by using JMetal’s graphical interface.[DN11]”

JMetal comes from scientific research and has become widely popular in solving multi-objective optimization problems. Although JMetal was initially created for Java, with its growing popularity, many independent ports have been created for different programming languages. In our work, we used JMetal.NET library.

Our Test Minimization component, after receiving the inputs, instantiates a JMetal.NET problem, sets the parameters we want to use, sets the number of variables to optimize that it is equal to the number of test cases, and pre-computes all the necessary information, like the one described for $\rho$, mentioned in section 3.5.3.

After the instantiation, we start the JMetal.NET optimization process. JMetal.NET calls our evaluation functions, that compute the objectives, with a binary flag (set of 0’s and 1’s) as input, whose size is equal to the number of test cases. Each flag passed represents a possible minimized test suite. As explained in section 3.4.1, a value of one on position $N$ means test case $N$ is part of the minimized test suite. Using that flag, our evaluation functions give JMetal an array whose values are the value of each objective.

JMetal runs NSGA-II algorithm that was described in section 3.4.2, calling our evaluation functions thousands of times for each individual generated, until the stop condition is met, and then passes back to our code a set of binary flags. The set of binary flags is the solution to the problem - it represents the Pareto front of minimized solutions.

After having the Pareto front of minimized solutions, our module creates the correct output structure and passes it to the next step.
4.1.4 Visual Studio Extension

Our Visual Studio extension module is the only module that is dependent on Visual Studio and the only module that can access Visual Studio API’s.

This module is responsible for integrating our system with Visual Studio. This integration allows a better user experience. The main tasks covered by this module are:

1. Use Visual Studio API’s to discover the open projects and Paths where they are located. This information is necessary for the input collecting process.
2. Integrate into Visual Studio a graphical user interface created in HTML5, which will be described in the next section.
3. Handle the events generated by the user interface and do the actions required, like starting a minimization process, set the minimization solution, etc.
4. Create a structure that Visual Studio understands and represents the minimized test suite, called a test play list. This is necessary to make Visual Studio only run the tests that are part of the minimized test suite.
5. Coordinate all the modules. When using the Visual Studio version of our tool, all the other modules never communicate with other modules aside from this one. For example, these modules start a minimization input collecting process, get that input, call the minimization system with that input and get the solutions, and then call with those solutions.

4.1.5 Graphical User Interface

The graphical user interface module is a module used by the Visual Studio extension to interact with the user. On the Interface and use cases section 4.3, we are going to show screens of this module.

We tried to make this module as simple and easy to use as possible, so as to make automatic test suite minimization techniques available to everyone.

This module was created using web technologies, in order to make it possible to use exactly the same code in other platforms and IDE’s. For example, if we want to create a multi-objective minimization system for eclipse IDE, we can use exactly the same interface. If, later on, a user of our system decides to change IDE, we can continue to use our system in the exact same way, reducing the learning curve of our solution.

Moreover, with the growing use of cloud computing, it is expected that test minimization system could soon be offered on the cloud in a test minimization, as a service system.

Much of the interfaces of cloud services run directly on a browser. By creating our interface in web technologies that are already able to run on the browser and by having a command line interface, we are very close to offer our approach in test minimization as a service that runs on the cloud.
4.1.6 Command Line Interface

The command line interface module serves as a command line interface to interact with almost all the features offered by the other modules, using an easy to understand command line interface.

The main features offered are:

- Save the information collected by the "inputs collecting system" to a file. If we can save the inputs information, we can try many different minimization approaches without spending time collecting this information over and over again. This was used during our empirical evaluation. Collecting the inputs is the most time consuming part. By being able to use the inputs collector independently of the other modules, if, later on, someone wants to implement a totally different minimization approach from the one we proposed, our module for collecting inputs can still be used, provided the other approach is able to interpret our inputs.

- Start minimizing a test suite. For this process we can specify the directory of a Visual Studio project and our system runs the input collecting process first, and then starts the minimization. Conversely, we can specify the path of a file with that input - this way the inputs are passed directly and our system starts the minimization using that information. This allows our minimization system to run on any software project, provided there is some component that generates the coverage matrix and cost execution vector of that project.

- Provide utilities that are useful to automate our empirical evaluation.

4.1.7 Data Manager

The data manager module is responsible for handling data manipulation tasks on our system. For example, if we want to save some structure, like the coverage matrix as JSON, we use methods provided by the data manager. To empirically evaluate our approach, we needed to save much data in a format compatible with statistic analyzers - it is also the data manager that handles this operation.

Data manager contains a set of data manipulation methods that are used across every other component. So, those methods were all implemented in this component, in order to promote code reuse.

4.1.8 SBFL

Spectrum based fault localization techniques are techniques that, given a set of tests and its pass or fail information and the components exercised by each test case, can compute an error score for each component. Components should be shown to the programmer in descending order by the debugging tool. If the component that has the bug appears first on the ordered list, it has fault localization effort (FLE) equal to zero, given that no effort was done in localizing it by inspecting other components first. If the faulty component appears on 10th position of the list, it has a fault localization effort of nine, given that nine different components were unnecessarily inspected first.
An Effective Minimization Tool

There were proposed many formulas for computing the error score by the scientific community. We used Ochiai coefficient, one of the best approaches according to the literature [AZvG06, AZGvG09]. Like our approach to solve the minimization that was a genetic algorithm, Ochiai coefficient is also based on biology. The ochiai score for a given component \( j \) (e.g. line of code) is defined as follows:

\[
OchiaiScore(j) = \frac{n_{11}(j)}{\sqrt{(n_{11}(j) + n_{01}(j)) \cdot (n_{11}(j) + n_{10}(j))}}
\]

(4.1)

Where:

- \( n_{11}(j) \) - Number of times component \( j \) is exercised during a test case that failed.
- \( n_{10}(j) \) - Number of times component \( j \) is exercised during a test case that passed.
- \( n_{01}(j) \) - Number of times component \( j \) is not exercised during a test case that failed.

The normal input of spectrum based fault localization techniques is a coverage matrix similar to the one described in section 3.3.1 and the test pass or fail information. We already had access to structures with all this information so as to be able to minimize the test suites. Given that, when minimizing test cases, we are just removing lines from the coverage matrix and positions from the error vector, during the Ochiai computation we can just ignore discarded lines and vector positions associated with minimized test cases. This makes it possible for the computing of Ochiai score, according to all the different minimized test suites, to be an efficient process.

We needed to implement this spectrum based fault localization module in order to ascertain if the test suites generated with our approach are in fact better in locating faults or not. These results will be reported in chapter 5.
4.2 Communication

For the test minimization process, there are two completely distinct, unavoidable and complex tasks:

1. Collect the information necessary for the minimization process.
2. Do the minimization process itself.

In every test minimization system, there is an unavoidable information exchange between the two modules that implement each task.

The problem is that, although all the existing minimization approaches implement this information exchange, that exchange is not documented or inter-operative between different programming languages and platforms, making it impossible to combine different test case information collectors with different test suite minimization approaches.

If it was possible to combine these information collectors with test cases minimizers, we would be able to test our approach with a java or c++ project, given that a collector for these languages was available. If a researcher wanted to improve the minimization process, they would not need to implement a collector - they could use an existing one. Implementing a collector is a time intensive task. On the other hand, if some IDE developer wanted to offer an automatic test minimization system for its users, they could implement a collector for that environment and use one of the minimization approaches available.

To try solve this problem, we are going to document the approach used on our test minimization information exchange. To exchange information, we used JSON. That format of exchange is growing in use, is efficient and is wholly inter-operable between different languages and platforms.

With a view to formally specify our information exchange, we created a JSON schema. JSON is json specification language. Its main function is to enable developers to validate if the json structures they are using are valid, according to some specification. Besides its main function, a JSON schema enables other things - for example, it is possible to automatically generate code for almost every programming language that implements our information exchange.

![Minimization Inputs](image)

Figure 4.2: JSON representation of the minimization inputs

Figure 4.2 shows us the general representation of the JSON schema of our inputs. It consists of an array that represents the components and an array that represents the test cases.
Figure 4.3 represents the contents of the array of components. Only "testCoverage" is a mandatory field. It is an array that stores the test cases that exercised a given component. If position number $N$ is greater than 0, it means that test case on position number $N$ on tests array exercised this component. All the other fields allow the minimization system to apprehend information about this requirement, such as the source-file where it is implemented, or the method this requirement belongs to, etc. Different properties from the ones we propose may be used. In our approach, the input collector passes all this data, but we do not make use of it for the minimization process. In the future, it could allow for the use of metrics like the average number of times each method is called.

In figure 4.4, we can see that the tests are represented by an array. The required fields are testID - which contains an identifier of the test case. In our case, this identifier is the fully qualified test name - and cost - which contains the cost of executing that test case (it should be greater or equal than 0). In our case we used time as a cost.

After the minimization process is finished, a minimization solver needs to pass back the results of the minimization. This results are presented in figure 4.5. The output we propose is an array of solutions - it could be just one or several of them. For each solution, we require that it contains a Boolean array. The position $N$ on the boolean array represents if test case number $N$ is part of the minimized solution. Besides this required field, we allow the use of other files to pass back data about a given solution - in our approach, we pass back the value of each objective we used.
An Effective Minimization Tool

Figure 4.4: JSON representation of the test cases

Figure 4.5: JSON representation of the minimization output
During this dissertation, not only did we create a noble minimization approach, we also created an extension for Visual Studio that uses our approach.

In this section, we will detail the user interface of our Visual Studio extension.

With a view of increasing the adoption of automatic test suite minimization systems, we endeavoured to make our extension as simple as impossible.

Our extension creates a space on the Visual Studio development environment, called "Test Suite Minimization System", where the minimization process is done. This space can be shown, hidden, moved around and resided on the development environment. The minimization process happens inside this space.

We can see in figure 4.6 (left), that, at the beginning, the user is presented with only a big start button. When the user presses that button, the minimization process starts: all the paths of the projects currently open on Visual Studio are discovered, all the test cases are found, all the input information described in section 3.3 is collected, and then the minimization process runs. All this happens with only a click of a button, no other action being necessary.

When the minimization process is concluded, the Pareto optimum solutions are shown. As we can see in figure 4.6 (right), the solutions are displayed in a table. For each solution, we present the value obtained on each of the objectives we have taken into consideration (described in section 3.1). If the user wants to go back to the start, they can press the back button anytime.

The users are able to sort the solutions according to any objective they desire. This allows the user to, for example, select the solution with best code coverage that runs under 5 minutes (300 seconds), or select the solution with biggest diagnosability, in an easy way. Given that we do not make assumptions about the objectives of the user, the user can choose the one that best fits their needs.
An Effective Minimization Tool

If the user wants to know the test cases that are included in a given solution, they can press the "plus" button of that solution - a list of the test cases included on that minimized solution and the pass-or-fail information from the last execution is shown. We can see an example of this in figure 4.7.

Figure 4.7: Display of test cases on a given solution and save button

Figure 4.7 also displays a save button. When the users believe to have found the ideal solution, they can save it by pressing the save button. After pressing the save button, a "Save as" window opens, as we can see in figure 4.8.
An Effective Minimization Tool

The users are able to save more than one solution, according to their different needs. For instance, one user may save a solution that is really fast, while providing good coverage and save another one with a very good $\rho$, to be used when automatic fault localization tools are run.

After the desired solutions are saved, the minimization process is concluded. The users can then use the normal test explorer interface, offered by Visual Studio, and choose an easy way to run all tests (as shown in figure 4.9, left), or a regression test suite previously saved with our tool (as shown in figure 4.9, right).

After having the regression test suites available on Visual Studio test explorer, the user can close our extension until they need to repeat the minimization process - if, for example, the test suites have changed considerably since the last execution. The regression test suite files - test play lists, as they are called in Visual Studio - can be sent by email or stored on company repository, and all the developers can use those regression tests, without even needing to have our extension installed. They would not be able to rerun the minimization process and generate different solutions, though.

Figure 4.10 shows a general overview of how our minimization system is integrated in Visual Studio development environment.
An Effective Minimization Tool

Figure 4.9: Visual Studio test explorer with the chosen minimized solution

Figure 4.10: General Overview of Visual Studio environment with the proposed minimization approach
An Effective Minimization Tool
Chapter 5

Empirical Evaluation

In this chapter we are going to describe the empirical evaluation done to our multi-objective test suite minimization approach and the results we got. In order to empirically evaluate our solution, we tested it with real world software programs. At the beginning of the chapter, we will describe the programs used and, after that, we will explain the methodology used to test our approach. Then, we will reach a conclusion, showing the results and giving our opinion about them.

5.1 Experimental Subjects

In order to test our minimization approach, we needed real world programs with automatic software tests using a support framework in Microsoft stack.

In order to be able to test a software, we needed access to the source code so we could collect the code coverage per unitary test. Although the development stack we support is one of the most used in the world, there are not many open source projects (with automatic tests) as compared to other stacks. One reason for this is that the stack we support is proprietary. Therefore, finding the right considerable open source projects was not an easy task.

After our research for experimental subjects, we decided to use these four subjects:

1. **Mathos Project** - “Mathos is a collection of tools for any kinds of mathematical calculations. It allows you to work with shapes, finance, unit conversions, and more!”
   
   Mathos is single library that contains different functions in areas like: Arithmetic, Coordinate Geometry, Fractions[Dag14].”

2. **PayPal.NET SDK** - Paypal is one of the most commonly used systems to make electronic payments for e-commerce. In order for Paypal to able to be integrated with different software that controls an e-commerce system, it provides API’s that allow external software to communicate with its service. Besides offering API’s, Paypal also offers open source SDK’s for several programming languages that easily and transparently allow the programmer of those languages to consume/use Paypal API’s. We used one of those SDK’s, the .NET one [Pay12].
Empirical Evaluation

If the SDK made available by Paypal contain some type of bug or are not secure, both an e-commerce enterprise and all its clients could suffer financial damage. Ensuring the correction of SDK is critical and automated tests are essential for this project.

3. **ReportGenerator** - Report Generator is a software solution written in c# that, given a XML file representing the code coverage from many different code coverage analyzers, interprets and outputs to the user an easy to understand and user-friendly report on the same structure, no matter how the code coverage was collected [Pal13].

4. **Subtitle Edit** - "Subtitle Edit is a free (open source) editor for video subtitles - a subtitle editor :)

With SE you can easily adjust a subtitle if it is out of sync with the video in several different ways. You can also use SE for making new subtitles from scratch (do use the timeline/waveform/spectrogram) or translating subtitles.

A Subtitle Edit dll (Subtitle Edit Light Library) is available for programmers (BSD New/Simplified license). Initially created and used by Sublight (a free Windows application for searching and downloading movie subtitles)[Ols14]". This software offers a great amount of features, ranging from automatic subtitle translation (using different API’s), to an advanced import/export system that supports almost all existing subtitles and text on video formats. It also uses ocr to extract subtitles from graphical formats.

The wide range of features offered by Subtitle Edit make it our biggest experimental subject. It has more than 100k lines of code not including comments.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Number of test cases</th>
<th>LOCs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mathos [Dag14]</td>
<td>205</td>
<td>4445</td>
</tr>
<tr>
<td>PayPal .NET SDK [Pay12]</td>
<td>249</td>
<td>4542</td>
</tr>
<tr>
<td>ReportGenerator [Pal13]</td>
<td>160</td>
<td>4754</td>
</tr>
<tr>
<td>Subtitle Edit [Ols14]</td>
<td>306</td>
<td>128454</td>
</tr>
</tbody>
</table>

Table 5.1: Subjects used on empirical evaluation

Table 5.1 presents the subjects used on the empirical evaluation, along with the number of automatic test cases and the number of lines of code (not including comments) that they contain. As we can see, our solution was evaluated with projects up to more or less 128 kLOC.

The value for the number of lines of code was the one reported by Visual Studio code metrics tool for the respective "solutions" of those subjects.

### 5.2 Experimental Setup

In order to evaluate if our approach can produce Pareto fronts for minimized test suites that are better at localizing a "bug" in case of it happening, we injected a collection of five faults/mutations.
Empirical Evaluation

on each of the experimental subjects. The only requirement for the mutations was that the program still compiles, in order to be able to run the tests, and that at the mutation make at least one test case fail, so as to allow automatic fault localization techniques to try and locate the bug.

To make it possible to have a value for the effort of localizing a fault and be able to compare that effort on different test suites and Pareto fronts, we implemented a spectrum based fault localizing technique.

In order to be able to have a compassion to know if our approach taking the triple of objectives \((\text{Cov}, \text{Cost}, \rho)\) was in fact better at localizing faults, we implemented two other approaches to minimize test suites:

1. Minimize using the tuple of objectives \((\text{Cov}, #)\). Try to maximize the coverage with the minimum number of test cases possible.

2. Minimize using the tuple of objectives \((\text{Cov}, \text{Time})\). Try to maximize the coverage, while spending the minimum amount of time running the test cases. This approach is the one described in section 2.6 and is the most common multi-objective optimization approach for test suite reduction.

For each project with the bugs injected, we minimized the test suite of that project using the three different approaches. The output of the minimization process is a set of test suites that are Pareto optimal, according to the objectives in consideration. For each Pareto front we save a structure that contains: the approach used to generate it, the project and bug it refers to, and all the different minimized test suites it contains. Besides the objectives values of each minimized test suite part of a given Pareto front we also include the FLE value of locating the faults injected.

To compute the value of FLE of a minimized test suite, we did the following steps:

1. We run Ochiai on that test suite and obtained an error score for each component.

2. We ordered the components in descending order, according to their Ochiai score.

3. We recorded the position of the component that contains the injected bug on the ordered vector. That was possible because we knew where the bugs where located. The position of the bug on ordered vector is the FLE value.

More formally, let \(j\) be a component, and \(C\) the set of all components. We can define FLE as:

\[
FLE(j) = |\{c \in C \land \text{Ochiai}(c) > \text{Ochiai}(j)\}|
\]  

(5.1)

In other words, equation 5.2 tells us that the FLE of a component \(j\) is equal to the number of other components with an higher Ochiai score.
5.3 Experimental Results

The output of the steps described on the Experimental Setup section 5.2 is a structure representing each Pareto front, the test suites it contains and its metrics (including FLE). We used four experimental subjects, each one with five faults injected, so we have a total of 20 Pareto fronts for each approach. In total, given that we tried three different approaches, we have 60 Pareto fronts.

We want to compare the twenty Pareto fronts generated by our approach with the ones generated by the other approaches. Comparing Pareto fronts is not easy, since the Pareto front is not only one minimization solution, but rather a set of them. The final one(s) will be chosen by the programmer/tester.

We want to test if the solutions present on the Pareto fronts of our approach are generally better in terms of localizing the randoms bugs inserted, when compared to the ones generated by other approaches.

If our approach is effectively better at localizing faults, it is expected that the average of the fault localization effort of the solutions on our Pareto fronts is lower. This happens mainly because of two reasons:

1. When two different solutions have the same value of coverage and time, when \( \rho \) is not used as an objective, a random one is chosen. When \( \rho \) is used as an objective, the one with best \( \rho \) is chosen. Given that solutions with best \( \rho \) are expected to be better at localizing faults, it is expected that the selected solution with \( \rho \) has better FLE.

2. When \( \rho \) is used as an objective, the algorithm will try to find solutions with a good value for \( \rho \) that otherwise would not be considered Pareto optimum and would not be found.

Besides the expectation that solutions on the Pareto fronts of our approach have a better average FLE, it is also expected that our approach generates Pareto fronts with the minimum value of FLE of all the solutions being better when compared to the minimum value of FLE of the other approaches, because we are also optimizing for \( \rho \) and finding solutions that otherwise would not be found.
## Empirical Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>⟨\text{Cov, #}⟩</th>
<th>⟨\text{Cov, Time}⟩</th>
<th>⟨\text{Cov, Time}, ρ⟩</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FLE</td>
<td>(\mu_{\text{FLE}})</td>
<td>(\text{Min}_{\text{FLE}})</td>
<td>(\mu_{\text{FLE}})</td>
</tr>
<tr>
<td><strong>Mathos</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>30</td>
<td>789</td>
<td>3</td>
<td>801</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1374</td>
<td>0</td>
<td>1578</td>
</tr>
<tr>
<td>3</td>
<td>92</td>
<td>1256</td>
<td>33</td>
<td>921</td>
</tr>
<tr>
<td>4</td>
<td>835</td>
<td>800</td>
<td>761</td>
<td>801</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>231</td>
<td>19</td>
<td>197</td>
</tr>
<tr>
<td><strong>PayPal .NET SDK</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>32</td>
<td>197</td>
<td>57</td>
<td>106</td>
</tr>
<tr>
<td>2</td>
<td>37</td>
<td>41</td>
<td>3</td>
<td>76</td>
</tr>
<tr>
<td>3</td>
<td>634</td>
<td>632</td>
<td>632</td>
<td>626</td>
</tr>
<tr>
<td>4</td>
<td>1046</td>
<td>1046</td>
<td>1046</td>
<td>1041</td>
</tr>
<tr>
<td>5</td>
<td>44</td>
<td>194</td>
<td>6</td>
<td>393</td>
</tr>
<tr>
<td><strong>ReportGenerator</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>187</td>
<td>10</td>
<td>136</td>
</tr>
<tr>
<td>2</td>
<td>236</td>
<td>1927</td>
<td>90</td>
<td>653</td>
</tr>
<tr>
<td>3</td>
<td>360</td>
<td>485</td>
<td>110</td>
<td>723</td>
</tr>
<tr>
<td>4</td>
<td>231</td>
<td>960</td>
<td>210</td>
<td>792</td>
</tr>
<tr>
<td>5</td>
<td>414</td>
<td>704</td>
<td>11</td>
<td>467</td>
</tr>
<tr>
<td><strong>Subtitle Edit</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>46</td>
<td>8779</td>
<td>9</td>
<td>14182</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>56331</td>
<td>1</td>
<td>34655</td>
</tr>
<tr>
<td>3</td>
<td>2787</td>
<td>59221</td>
<td>660</td>
<td>80459</td>
</tr>
<tr>
<td>4</td>
<td>64</td>
<td>42229</td>
<td>38</td>
<td>55530</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>53659</td>
<td>9</td>
<td>53492</td>
</tr>
</tbody>
</table>

Table 5.2: FLE value on original test suite and average and minimum FLE of each Pareto front
Empirical Evaluation

On table 5.2, we present the value of FLE on the original test suite and the average and minimum FLE of the solutions on the Pareto fronts of each approach, by experimental subject and fault.

Independently of the approach, we observe two common patterns: the average FLE of the Pareto fronts is higher than the FLE value of the original test suite; the minimum FLE of the Pareto fronts is lower than the FLE value of the original test suite. This could be explained by the fact that there are solutions on the different Pareto fronts which discard many test cases, making their diagnosability very low, and these solutions make the average higher. On the other hand, we need only a solution which discards a test case that made a component rank higher than the faulty one, to have a lower minimum value for FLE on the Pareto front.

Using the information from table 5.2, we present on figure 5.1 a graphic that shows the improvement on average FLE, when comparing our approach with two of the most common multi-objective approaches. The graph shows the improvement on FLE for each injected fault, according to the order of table 5.2.

![Figure 5.1: Improvement of average FLE when comparing our approach with time and coverage](image)

On average, for the samples used, the Pareto fronts generated with our approach had a 31.3% improvement on the average FLE value. We can see on figure 5.1 that on 25% of the cases our approach was able to get an improvement on the average FLE value of over 50%. In our sample of 20 faults, our approach presented a better average FLE on 90% of the cases. This allows us to conclude that, 

at a 95% confidence level, our approach generates Pareto fronts with a better average FLE between 77% to to 100% of the times

when compared with the use of coverage and time.

On figure 5.2 we can see a graph that shows the improvement on minimum FLE, comparing our approach with coverage and time approach. On average, we observed 18% on the minimum FLE value. The Pareto fronts generated with our approach had a strictly better minimum FLE 50% of the times (10), a better or equal minimum value 95% times (19) and a worse minimum value 5% of times (1).
Empirical Evaluation

Figure 5.2: Improvement of minimum FLE when comparing our approach with time and coverage

We can see on table 5.3 that, a significant number of times, our approach is going to generate a Pareto front with a solution that presents a better FLE than all the other solutions generate with the time and coverage approach.

There was a fault where our approach had a worse minimum (fault 3), but we can see on graph 5.1 that, even in that case, our Pareto front presented a better FLE effort. That could be explained by the NSGA-II algorithm, which is genetic and has some random behavior, being able on the Pareto front of coverage and time approach to find a solution with a very good FLE that also was Pareto optimal, while on our execution it was not. According to the confidence level, that event is not statistically significant.

Although the coverage and time approach is the more common multi-objective test minimization approach and coverage and cardinality is rarely used, we also compared our approach with coverage and cardinality. We observe very similar results in comparison between our approach and coverage and time approach. Our approach presented an average improvement on average FLE of 28%. Our average FLE was better 90% of the times. Our minimum FLE was strictly better 50% of the times and better or equal 90% of the times.

Our results also allow us to conclude that approach and our evaluation functions are able to scale. We tested our approach on a subject with +100 kLOC, the coverage of which had more than three million elements. After having the input data described on section 3.3, our approach was able to minimize the test suite of that big subject without any problem.

5.4 Threats to Validity

We can identify some threats to both external and internal validity of our results. To the external validity of the empirical results we reported, the main threats are:

- **Only four experimental subjects were used.** We cannot be sure that there are not some types of projects where our approach does not work as well. To minimize this threat, we
Empirical Evaluation

<table>
<thead>
<tr>
<th>Minimum:</th>
<th>Confidence Interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is strictly better</td>
<td>39% to 61%</td>
</tr>
<tr>
<td>Is better or equal</td>
<td>90% to 99.9%</td>
</tr>
<tr>
<td>Is worse</td>
<td>0.1% to 10%</td>
</tr>
</tbody>
</table>

Table 5.3: Confidence for minimum FLE results when comparing our approach with time and coverage approach

used different types of real world software projects and we could not identify a project where our approach worked worse when compared to the others. However, it is plausible to assume that if our approach is used on a project with very different characteristics, when compared with the experimental subjects used, the results could be different.

- **Our approach was only tested on the c# programming language.** Different programming languages have different constructs and are used in different ways. We are not certain if this results are valid in a different programming language, although c# is an Object oriented programming language with the constructs very similar to c++ and java. Thus, it is plausible to assume that the results would also be similar on those languages.

- **The faults may not represent the sample of all possible faults.** There are many different kinds of faults on software programs. It would be impossible to try our approach on all of them. Programmers can be very creative when they make faults on a software. To minimize this threat, we used a total of twenty faults and tried to use different types of faults.

The main threat to the internal validity of our approach is that there could be an error on the implementation of our approach, or on the implementation of the other approaches. Besides errors on the approach itself, there could be other errors, as well: when collecting the coverage matrix, which involves getting code coverage per test case and getting this information is a complex process involving lots of code and the use of external software interfaces; on spectrum based fault localization implementation, although less probable; and on the external libraries like Jmetal.NET that implements NSGA-II.

To minimize the internal validity threats, we used the best practices for the software development process and we tested the components before the experimental phase and individual result checking.
Chapter 6

Conclusion and Future Work

We started this document by highlighting that software nowadays has an even higher rule in our world than before. We mentioned that software is not perfect and software errors could cause enormous catastrophes. Therefore, software testing is a very important step in the software development process.

As software became more complex, so did test suites, in order to accompany its evolution. Test suites grew to a point where it is impractical to run all the tests after each code change.

A subset of the test suite has to be chosen. It is important that running that after each code change becomes a practical task.

We checked that most of the existing proposed approaches are single-objective, by using heuristics or by solving the minimal hitting set problem. The multi-objective ones only focus on detecting faults, normally by trying to maximize code coverage while trying to minimize the execution cost. Tests are not only useful in detecting faults, they can also give valuable information that helps to locate them. Until now, there was not a multi-objective test suite minimization approach that tried to minimize the fault localization effort of the test suites, when they are used as input of automatic fault localization techniques.

We were able to implement a test suite minimization approach that, besides time and coverage, also uses $\rho$ - a metric from the literature, which was proven to be related with the ability of a given test suite to locate faults, when used with autonomic fault localization techniques.

There is currently a lack of practical tools - the one that allows automatic test reduction works only for requirements coverage. At present, if a company requires a minimization solution that needs to take into account more than code coverage, there is no public available automatic tool to solve the problem. Visual Studio is one of the most used IDE’s in the world, and is used in big industrial projects. Some of these projects could probably benefit from test minimization. Until now, there was no automatic test suite minimization tool for Visual Studio.

We were able to implement a practical tool in the form of an extension, that is integrated on Visual Studio, thus solving with success the software engineering problems that implementing a system like this involves. We propose a modular architecture that is easy to port to other IDE’s and programming languages. In addition to a Visual Studio extension, we proposed an approach
Conclusion and Future Work

that works with any language or platform, provided there exist a collector for the minimization information.

We checked that implementing a test minimization system always involves a minimum of two modules: an information collector, which collects the necessary information and is language/platform specific; a test minimization system that, using the input, models a problem, solves it and returns the minimized solutions. The minimization is not dependent on the platform or programming language. Right now there is no documented way of exchanging information between these two modules. In order to solve that issue, we proposed an efficient JSON structure and created a JSON schema representation for it, so as to allow the validation of JSON and automatic code generation.

After implementing our solution, we did an empirical evaluation on it. Our results indicate than the test suites minimized with our approach perform better at localizing faults when they are used with spectrum based fault localization techniques. We were able to improve the average fault localization effort (FLE) value by 31%. We were able to conclude, at a 95% confidence level, that our approach produces a better average (FLE) between between 77% to 100% of the times, when compared to the most commonly existing multi-objective approach.

6.1 Main Contributions

During this work these main contributions were made.

1. We demonstrated that minimization can have a substantial impact on the quality of the diagnostic of automatic fault localization techniques. Different minimization approaches had different values for the average and minimum FLE on a random sample of bugs.

2. We proposed a multi-objective test suite minimization approach that, besides trying to maximize fault detection ability and decreasing the execution cost, also tries to produce minimized test suites that perform well in localizing faults.

3. We were able to empirically evaluate our solution and quantify our added value, when compared with most common multi-objective approaches.

4. We proposed and documented efficient ways of computing each of the objectives we used (coverage, time and $\rho$), which take advantage of the fact that both the coverage matrix and the solution encoding are binary.

5. We were able to implement the first Visual Studio test minimization extension, after overcoming all the technological challenges associated with that task.

6. We publicly documented the first language-and-platform-independent format for a test minimization information exchange format. We chose JSON for our format - the most used data exchange format for cloud scenarios, opening ports for test minimization as a service.
6.2 Future Work

Even after having finished this work, we think it is still possible to do some improvements and extend this work.

**Improve service performance**

Performance is not a main concern in the test minimization process, although there are some advantages if the minimization process is fast to run.

Given that our work objectives were very ambitious, and focusing on the performance of the minimization process would be in its own right a possible project, performance was not a main objective in the proposed approach.

In the future, an improved and more performant version of our multi-objective approach could quite certainly be proposed. More research could be done in order to find more performant solutions for minimizing test suites.

**Create new test case information collectors**

The test suite minimization process is dependent on collecting test case information, such as requirements’ coverage (e.g. code coverage), test case run-time, test case historic, pass or fail, or any other relevant information.

Although we propose a generic service, which is able to work with any software development process independent of the programming language, testing framework or device, test case information collectors need to exist for the specific testing scenario.

Until now we just created an information collected for the .NET environment, other collectors can be created.

**Improve communication between information collectors and minimization service**

A standardized approach for communicating the necessary test suite information for the minimization process has never been proposed before.

We believe a standardized approach is possible and a welcome advance because it will create a clear separation between the information collection process and the minimization process. This separation will make it possible for different information collectors and minimization services to work together.

Given that the approach we proposed is the first general documented approach for test minimization information exchange, and the standards for other information exchange have had a notorious evolution, we expect the same to happen to this information exchange.
Conclusion and Future Work
References


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


