UNIVERSIDAD DE CÁDIZ

TESIS DOCTORAL

Técnicas de Prueba Avanzadas para la Generación de Casos de Prueba

Advanced Testing Techniques for Test Case Generation

Autor:

Kevin Jesús Valle Gómez

Directores:

Dra. Inmaculada Medina Bulo

Dr. Pedro Delgado Pérez

Escuela Superior de Ingeniería

Programa de Doctorado en Ingeniería Informática

Fecha: 31 de enero de 2023

Conformidad de los Directores

Dª María Inmaculada Medina Bulo y D. Pedro Delgado Pérez, profesores del Departamento de Ingeniería Informática de la Universidad de Cádiz, siendo Directores de la Tesis titulada *Técnicas de Prueba Avanzadas para la Generación de Casos de Prueba*, realizada por D. Kevin Jesús Valle Gómez y enmarcada en el Programa de Doctorado en Ingeniería Informática, para proceder a los trámites conducentes a la presentación y defensa de la tesis doctoral arriba indicada, informan que se autoriza la tramitación de la tesis.

Los directores de tesis

Firmado por MEDINA BULO MARIA INMACULADA - ***5650** el día 31/01/2023 con un certificado emitido por AC

Inmaculada Medina Bulo

DELGADO PEREZ PEDRO - 44965478H

Firmado digitalmente por DELGADO PEREZ PEDRO - 44965478H
Fecha: 2023.01.31 11:21:34 +01'00'

Pedro Delgado Pérez

Puerto Real, a 31 de enero de 2023.

Agradecimientos

Qué difícil es mencionar a todas las personas que tengo en el pensamiento, que habéis hecho que esta tesis doctoral sea posible. Tenéis mi más sincero agradecimiento, especialmente:

- *Inmaculada* y *Pedro*, mis directores y compañeros, grandes ejemplos en lo personal y en la investigación. Confiasteis en mí a pesar de todas las dificultades y cambios a los que nos hemos enfrentado. Gracias por vuestra paciencia y buen hacer.
- Mi *madre* y mi *padre*, por su incondicional apoyo y ayuda desde siempre. Me habéis arropado en todas mis decisiones, y habéis hecho posible que sea quien soy ahora.
- Mi hermano, *Abel*, mi ingeniero por referencia. Siempre conmigo, intercambiando *memes*, noticias interesantes y no siempre muy sensatas, y guiándome por mundos nuevos. El mejor apoyo y aliado que se puede tener.
- Todos los miembros del grupo UCASE. En los buenos y en los malos momentos habéis sido como mi segunda familia.
- *Carmina*, por ser mi compañera y mentora desde mis inicios en este mundo de la investigación. Esto es un sitio mejor gracias a ti.
- *Antonio García*, por hacerme ver que este trabajo realmente era posible y servir de guía con su paciencia infinita. Mi experiencia en Birmingham fue fundamental, imprescindible para estar donde estoy ahora.
- Toda la gente que ha estado conmigo en la Universidad. Como *David*, a quien llevo presente gracias a los desayunos y viajes "divulgativos" que hemos tenido. *Juan Carlos*, quien día a día sigue ayudándome a tomar las mejores decisiones. *Dani*, quien se ha ganado con creces su sitio, por todas las risas que hemos tenido juntos. Y otros muchos, como *Fernando* y *Damián*, que hacen que volver al trabajo sea mucho más agradable.
- Quien me aguanta personalmente día a día, con su cariño y paciencia. Sin darse cuenta, ha aportado citas que quedarán para siempre reflejadas en este documento de tesis. Lo mejor está por venir.
- *Luna*, cuyas siestas en mi mochila me recargaron el espíritu de energía.

Con todo mi corazón, a mis abuelos.

Agradecimientos Institucionales

Este trabajo fue financiado por la beca para la realización de tesis doctorales en la industria con referencia 2017-083/PU/EPIF-FPI-NAVANTIA/CP de la Universidad de Cádiz y la empresa Navantia, así como los fondos FEDER a través de los proyectos nacionales del Ministerio de Ciencia e Innovación con referencia TIN2015-65845-C3-3-R, RTI2018-093608-B-C33, RED2018-102472-T, PID2021-122215NB-C33 y PDC2022-133522-I00.

"The saddest aspect of life right now is that science gathers knowledge faster than society gathers wisdom."

Isaac Asimov

Abstract

Software testing is a crucial phase in software development, particularly in contexts such as critical systems, where even minor errors can have severe consequences. The advent of Industry 4.0 brings new challenges, with software present in almost all industrial systems. Overcoming technical limitations, as well as limited development times and budgets, is a major challenge that software testing faces nowadays. Such limitations can result in insufficient attention being paid to it.

The Bay of Cadiz's industrial sector is known for its world-leading technological projects, with facilities and staff fully committed to innovation. The close relationship between these companies and the University of Cadiz allows for a constant exchange between industry and academia.

This PhD thesis aims to identify the most important elements of software testing in Industry 4.0, based on close industrial experience and the latest state-of-the-art work. This allows us to break down the software testing process in a context where large teams work on large-scale, changing projects with numerous dependencies. It also allows us to estimate the percentage benefit that a solution could provide to test engineers throughout the process.

Our results indicate a need for non-commercial, flexible, and adaptable solutions for the automation of software testing, capable of meeting the constantly changing needs of industry projects. This work provides a comprehensive study on the industry's needs and motivates the development of two new solutions using state-of-the-art technologies, which are rarely present in industrial work. These results include a tool, ASkeleTon, which implements a procedure for generating test harnesses based on the Abstract Syntax Tree (AST) and a study examining the ability of the Dynamic Symbolic Execution (DSE) testing technique to generate test data capable of detecting potential faults in software. This study leads to the creation of a novel family of testing techniques, called *mutation-inspired symbolic execution (MISE)*, which combines DSE with mutation testing (MT) to produce test data capable of detecting more potential faults than DSE alone. The findings of this work can serve as a reference for future research on software testing in Industry 4.0.

The solutions developed in this PhD thesis are able to automate essential tasks in software testing, resulting in significant potential benefits. These benefits are not only for the industry, but the creation of the new family of testing techniques also represents a promising line of research for the scientific community, benefiting all software projects regardless of their field of application.

Resumen

La prueba del software es una de las etapas más importantes durante el desarrollo de software, especialmente en determinados tipos de contextos como el de los sistemas críticos, donde el más mínimo fallo puede conllevar la más grave de las consecuencias. Nuevos paradigmas tecnológicos como la Industria 4.0 conllevan desafíos que nunca antes se habían planteado, donde el software está presente en prácticamente todos los sistemas industriales. Uno de los desafíos más importantes a los que se enfrenta la prueba del software consiste en superar las limitaciones técnicas además de los tiempos de desarrollo y presupuestos limitados, que provocan que en ocasiones no se le preste la atención que merece. El tejido industrial de la Bahía de Cádiz es conocido por sacar adelante proyectos tecnológicos punteros a nivel mundial, con unas instalaciones y un personal totalmente implicado con la innovación. Las buenas relaciones de este conjunto de empresas con la Universidad de Cádiz, sumadas a la cercanía geográfica, permiten que haya una conversación constante entre la industria y la academia.

Este trabajo de tesis persigue identificar los elementos más importantes del desarrollo de la prueba del software en la Industria 4.0 en base a una experiencia industrial cercana, además de a los últimos trabajos del estado del arte. Esto permite identificar cada etapa en la que se desglosa la prueba del software en un contexto donde trabajan equipos muy grandes con proyectos de gran envergadura, cambiantes y con multitud de dependencias. Esto permite, además, estimar el porcentaje de beneficio que podría suponer una solución que ayude a los ingenieros de prueba durante todo el proceso.

Gracias a los resultados de esta experiencia descubrimos que existe la necesidad de soluciones para la automatización de la prueba del software que sean no comerciales, flexibles y adaptables a las constantes necesidades cambiantes entre los proyectos de la industria.

Este trabajo aporta un estudio completo sobre las necesidades de la industria en relación a la prueba del software. Los resultados motivan el desarrollo de dos nuevas soluciones que utilizan tecnologías del estado del arte, ampliamente usadas en trabajos académicos, pero raramente presentes en trabajos industriales. En este sentido, se presentan dos resultados principales que incluyen una herramienta que implementa un procedimiento para la generación de arneses de prueba basada en el Árbol de Sintaxis Abstracta (AST) a la que llamamos ASkeleTon y un estudio donde se comprueba la capacidad de la técnica de pruebas Ejecución Simbólica Dinámica (DSE, por sus siglas en inglés) para generar datos de prueba capaces de detectar fallos potenciales en el software. Este estudio deriva en la creación de una novedosa familia de técnicas de prueba a la que llamamos *mutation-inspired symbolic execution (MISE)* que combina DSE con la prueba de mutaciones (MT,

por sus siglas en inglés) para conseguir un conjunto de datos de prueba capaz de detectar más fallos potenciales que DSE por sí sola.

Las soluciones desarrolladas en este trabajo de tesis son capaces de automatizar parte de la prueba del software, resultando en unos beneficios potenciales importantes. No solo se aportan beneficios a la industria, sino que la creación de la nueva familia de técnicas de prueba supone una línea de investigación prometedora para la comunidad científica, siendo beneficiados todos los proyectos software independientemente de su ámbito de aplicación.

Contents

List of Figures

List of Tables

Abbreviations

DSE	**D**ynamic **S**ymbolic **E**xecution
MT	**M**utation **T**esting
SUT	**S**oftware **U**nder **Test**
AST	**A**bstract **S**yntax **T**ree
MISE	**M**utation-**I**nspired **S**ymbolic **E**xecution
IT	**I**nformation **T**echnology
TCE	**T**rivial **C**ompiler **E**quivalence

Chapter 1

Introduction

"Don't ask whether you can do something, but how to do it."

Adele Goldberg

This introductory chapter presents a summary of the main aspects of the conducted research. To this end, it sets out the motivation behind this research work, as well as the objectives to be achieved. Then, this chapter presents the contributions produced during the PhD thesis period and ends with a description of the structure of the rest of the document.

1.1 Introduction and motivation

Software testing has always been a fundamental pillar during the development of any software project. Nowadays it is common to find bugs in all kinds of devices that implement software: from the simplest ones in consumer computing, such as errors in video games [15] or smartphones that consume too much battery [72], to errors in more complex systems like medical devices that apply radiation to people or military defence systems [101], where a small software error can lead to catastrophic consequences. The technological advances in all systems today add value to and complicate the entire software testing process.

Regarding industrial systems, these projects are characterised by having hundreds of thousands of lines of code developed by large teams, coordinated with external elements

such as devices whose availability may be limited. This type of development involves additional integration work, both between program modules and with the devices that will ultimately store and execute the various programs. All this has intensified with the advent of Industry 4.0, a new paradigm that foresees the interconnection of all systems through technologies such as IoT (Internet of Things) or Big Data. In such a context, where the presence of errors should be minimal or non-existent, it is natural that the cost of software testing skyrockets. For this reason, there is a need for the creation of new testing techniques to reduce the total cost of projects while increasing the confidence in the quality of the software.

Thanks to the good relations between the University of Cadiz and the business fabric of the Bay of Cadiz, it is possible for both researchers and industrial technical staff to get first-hand knowledge of the latest state-of-the-art techniques, as well as the current industrial needs. In the first approach between both entities, the enormous personal and economic effort that goes into the creation and execution of test cases in software development projects stands out. We observe that part of the manual effort required in the testing process could be replaced by automated techniques well established in the research community.

This PhD thesis aims, with the help of these good relations, to identify the characteristics and requirements of software projects in today's industrial environment. We have established two main research lines: the study of the needs of Industry 4.0 in terms of software testing and the improvement of the test generation process.

For the first line, we rely on the experience of the business fabric of the Bay of Cadiz and other related work in the literature. An exhaustive study of the state of the art in terms of Industry 4.0 and the application of testing processes to projects related to it is carried out. Afterwards, the stages of software testing in an industrial environment are identified, as well as the benefits that applying good strategies and the latest technologies in each stage entails in the economic, development quality and product quality areas. One of the needs identified early on is based on the lack of an automated or semi-automated process that facilitates the generation of test cases for test engineers. Current commercial solutions do not fit the needs of this set of companies, not only because of their high cost but also because of the integration and maintenance work involved. In addition,

their compatibility with future projects is not assured, forcing test engineers to perform a great part of the testing process manually.

The latter motivates the second line of research. Currently, diverse powerful techniques for test case generation exist in the literature that could be integrated into the testing processes of the industry. Some techniques rely on the Abstract Syntax Tree (AST), which translates the source code into a tree-like structure, facilitating its analysis for the detection and generation of elements based on it. Another technique is Dynamic Symbolic Execution (DSE), which is capable of traversing the source code without the need for specific values, proposing a set of values that go to each branch of the source code and, therefore, serving as effective test data in terms of structural coverage criteria (coverage of lines, branches, etc.). Beyond structural coverage criteria, it is possible to use more advanced test adequacy criteria to measure the quality of the generated tests, such as Mutation Testing (MT), which introduces small changes in the source code that simulate mistakes commonly made by programmers. The aim of this technique is to evaluate the ability of the test cases to detect potential defects.

By applying these techniques and aided by industrial monitoring, two tools have been developed that implement these techniques for test generation. Firstly, we have developed ASkeleTon, a tool that obtains and analyses the AST, being able to generate test harnesses ready for compilation and execution based on a code under test written in C or C++. Secondly, we have conducted a comprehensive study on the ability of DSE to detect faults based on MT. The results indicate that, in its current state, DSE is not as effective in terms of mutant detection as it could actually be. Our research has uncovered limitations in the generation of test data. To address these limitations, we propose a new technique called *mutation-inspired symbolic execution* (MISE). This approach combines DSE and MT to improve the generation of test data by incorporating the information provided by MT into DSE. Finally, we have presented several different implementations for *MISE*, to potentially obtain test data with a high capacity to detect mutants.

1.2 Objectives

This PhD thesis work pursues the following objectives.

Objective 1 (O1) - Complete review of the current state of Industry 4.0, with a special focus on software development and testing. In order to achieve this objective, an analysis of the most current works in the literature is performed. Furthermore, the collaboration with part of the industrial fabric of the Bay of Cadiz allows us to observe first-hand how they work in the different parts that comprise Industry 4.0. This makes it easier to understand and document as much as possible the characteristics, benefits and limitations associated with the software testing stage in these industrial environments.

Objective 2 (O2) - Literature review of techniques for the generation and validation of test cases in any environment, either industrial or open source. To achieve this objective, based on the results obtained in O1, it is necessary to identify and analyse the most current state-of-the-art testing techniques. Specifically, we focus on AST for the analysis and generation of test code, DSE for the generation of test data and MT for the evaluation of the generated tests. This also allows us to identify which current tools are best suited to industrial needs in order to design and implement a new and complete test generation process.

Objective 3 (O3) - Design of a process for the complete generation of test harnesses. To achieve this goal, the AST will be studied in depth to understand which elements are useful and how they can be used for test case generation, with a special focus on the test case structure.

Objective 4 (O4) - Implementation of a tool that incorporates each of the elements studied in O3. This objective aims to develop a new tool that, based on the AST, is able to generate a complete test harness ready for execution. With the help of the industrial experience and the results of O1, several aspects that are worth taking into consideration are identified and applied:

- Modularity: the development of the tool, as well as its results, will have a modular perspective from the start, in order to be able to use its modules in future projects and adapt to the constantly changing needs in the industry.

- Ease of use: the tool should not require a large configuration, making its integration a simple task.
- Readability of results: the tool should produce elements such as logs, initial data and test cases that are easily readable by the human eye.
- Ease of integration with other techniques: integration with data generation techniques should be a simple task, making the tool valuable to both industry and the research community.

Objective 5 (O5) - Conduction of a comprehensive study on the capacity of DSE for the detection of potential faults in software. The good results when using DSE for test generation in terms of structural coverage criteria are well known. However, in the literature, the performance of this technique in relation to more advanced testing criteria like MT is not so clear. To achieve this goal, we will take a set of real utilities and will analyse the ability of tests generated with DSE in the detection of mutants.

Objective 6 (O6) - Design and implementation of a new test data generation process by applying DSE. Based on the results obtained in O5, it is proposed to improve the current DSE techniques and tools in order to improve their criteria for test case generation. To achieve this goal, it is foreseen to include other techniques in an auxiliary way, forming a new family of techniques that goes beyond what is offered by the current DSE solutions.

Objective 7 (O7) - Validation of the obtained results with open source case studies. We postpone a complete industrial validation for the future; this validation is more complex, challenging and, more importantly, the results cannot be fully disseminated due to confidentiality clauses. Moreover, the correct accomplishment of all the steps of an industrial validation is time-consuming, incompatible with the limited development time of a PhD thesis. For these reasons, the aim is to validate all the products generated using example codes or programs available as open source of diverse sizes.

1.3 Contributions

This PhD thesis provides several contributions, which are listed below.

- A complete and up-to-date state of the art on the most relevant topics covered in this PhD thesis work. Specifically, software testing in general is reviewed, with a special focus on the techniques for its automation and the existing tools. Additionally, Industry 4.0 is reviewed, with a special focus on the role of software testing in this new paradigm, where it becomes clear how current testing techniques will have to be adapted. All of this is covered in Chapter 2.

- An elaborate study on the situation of software testing in industry based on the literature and real experience with companies in the business fabric of the Bay of Cadiz. Specifically, each of the stages of software testing carried out in large industrial projects have been identified. In addition, the economic, product quality and development quality benefits associated with an appropriate software testing performance have also been identified. Finally, all this is discussed and leads to the rest of the contributions of this PhD work. All of this is covered in Chapter 3.

- A new process, accompanied by a new tool, for the generation of test harnesses.This new process is based, in part, on the experience and knowledge acquired from the business fabric of the Bay of Cadiz. The tool, known as ASkeleTon, is based on AST analysis for the generation of ready-to-run test harnesses. Its modular design facilitates essential software quality factors such as maintenance and extension of the tool as well as of the test harnesses it generates. The latter include templates in their implementation, thereby separating the test data from the test cases. Hence, it enables the inclusion of external test data, which can be manually provided by experts in the software domain, or automatically generated by more advanced state-of-the-art techniques. The result is a modular, easy-to-use and easy-to-integrate tool. All of this is described in Chapter 4.

- A study on the ability of DSE, in its current state, to generate test cases capable of detecting mutants introduced by MT. Starting from a set of open-source utilities widely used in GNU operating systems, DSE is applied to produce a set of test cases. A set of mutants is then generated for each of the utilities and then it is checked whether the test cases can detect them. The results seem to indicate that

DSE is not able to kill a considerable number of mutants, thus requiring further research work in this area. All of this is covered in Chapter 5.

- A complete analysis of the improvement opportunities detected in DSE to increase its ability to kill mutants compared to its current state, including a study where DSE is combined with MT to improve the results. Following the identified improvement opportunities, we present *MISE*, a new family of techniques that combine DSE with MT to generate test data capable of killing more mutants. We present a prototype combining both techniques directly, where it is possible to see an improvement in the results. Finally, different implementations of MISE are proposed to reduce some limitations of the prototype and lower the cost of the process. All of this is covered in Chapter 5.

1.4 Structure of the doctoral thesis

This section outlines the structure of the PhD thesis document.

- Chapter 1 shows an introduction along with the motivation for the whole work. This is followed by an enumeration and description of each of the objectives pursued. Finally, it describes the main contributions and ends with the structure of the document.
- Chapter 2 covers the background and state of the art of this PhD thesis. It begins by defining the terms error, failure, and defect and their role in software testing. The chapter then discusses the various definitions of software testing and the importance of quality in this process. The chapter also examines the different stages of software testing, including unit testing, integration testing, and acceptance testing. Finally, it covers the challenges and limitations of software testing and the need for techniques such as Dynamic Symbolic Execution (DSE) and Mutation Testing (MT) to overcome these challenges and improve the quality of test data.
- Chapter 3 is an in-depth study of the current situation of software testing in industry. Based on the industrial experience of the business fabric of the Bay of Cadiz, it begins with a motivation followed by a complete description of each of the stages of software testing in this type of environment. Next, the benefits that

a solid software testing process produces are shown from different perspectives. Finally, a discussion is established on the current and future state of software testing in this field, identifying the elements that give rise to the development of the following chapters.

- Chapter 4 shows the design and implementation of a new tool for the automatic generation of test harnesses from the AST, known as ASkeleTon. After a description of the motivation and the main design decisions that guide the development of the tool, there is a review of each of the elements of the tool. Specifically, we highlight the modular design in both its design and its generated files, exposing particularities and decisions taken during the development. Finally, a small case study is shown with a test class, and the chapter ends with some conclusions.

- Chapter 5 shows a complete study where the initial ability of DSE to detect mutants is evaluated. After initial results confirming that this technique is not capable of obtaining strong test cases on its own, it is combined with MT, showing a substantial improvement. This led to the emergence of a new family of techniques called *naive MISE*, which consists of extending DSE to consider MT elements while generating the tests. As the first implementation entails a high increase in cost, we propose three different, more sophisticated implementations capable of performing *naive MISE* at a lower cost.

- Chapter 6 is a review of the results obtained during the PhD work as a whole, serving as a general discussion (please note that, when applicable, each chapter already includes its own particular evaluation and discussion). In particular, an extensive analysis of the results of MISE is conducted, where we have developed one of the more sophisticated implementations proposed in the previous chapter. This involves the implementation of specific mutation operators, resulting in promising results and motivating the development of this family of techniques.

- Chapter 7 discusses the final conclusions of all the work done. It sets out some lines of future work along which this PhD work can be further expanded. Finally, there is a list of all scientific publications made during the PhD thesis period.

Chapter 2

Background and State of the Art

"Program testing can be used to show the presence of bugs, but never to show their absence!"

Edsger W. Dijkstra

2.1 Software testing

Software engineering is a constantly evolving field. Topics such as software development techniques, programming languages and hardware configurations undergo radical changes every few years. As Myers, Sandler, and Badgett [58] discuss in their book, computers are almost everywhere: smartphones, smartwatches, smart TVs, and so on. Computers are more powerful than ever and are part of our daily lives. As software is everywhere, it becomes more and more important that it has as few bugs as possible. This is where software testing plays a major role, being an essential stage of software development.

In order for software testing to be carried out in the best possible way, it is important to have a proper awareness of its definition. Throughout the bibliography it is possible to find a variety of definitions.

Since the beginnings of software development, experts such as Dijkstra have pointed out that the effectiveness of software testing lies in the sample of evidence that there are bugs, but never in their absence [28]. This concept, whose quote heads this chapter, serves as the basis for all modern definitions that have been proposed to date.

Hetzel [42] defines software testing as the activity of evaluating an attribute or capability of a system to determine whether it meets the established requirements. In the same work, the authors state that software testing is a measure of quality for programs. Craig and Jaskiel [22] analyse some definitions available in the literature and conclude that software testing is a concurrent lifecycle process of engineering, using and maintaining tests with the goal of measuring and improving the quality of programs. Informally speaking, it could be said to be any activity intended to obtain sufficient evidence that the software performs its functions as expected.

If there is one thing that all definitions have in common, that is the concept of quality. However, can we be sure that a project has sufficient quality after passing the software testing stage? Going back to the work by Myers, Sandler and Badgett [58], they talk about a psychological concept related to software testing. When testing a program, it is best to start with the premise that it contains bugs and our goal will be to find as many as possible. Therefore, the authors propose the following definition: *"Software testing is the process of running a program with the intention of finding bugs"*. Being clear about this definition makes all the difference when constructing tests for software projects. If our goal is to prove that programs work, we will instinctively build tests that are less likely to show bugs. However, if we take a destructive approach, having as a premise to find bugs, we will build more critical tests, capable of finding bugs in the program. This is also why development and testing in projects are usually assigned to different teams. When it is assumed that a software program has bugs —as it is usually the case—, it is tested with the aim of finding those bugs.

In software testing, it is important to distinguish between different types of issues that can arise in a software system. These issues are often referred to as *errors*, *failures*, *bugs*, *faults* and *defects*. However, it is important to note that these terms are not always used consistently and may be used interchangeably in different contexts. In this work, we will define these terms as follows: an error is a deviation from the expected behaviour of a software system, which can be caused by a flaw in its design, implementation, or operation. A failure is the inability of a software system to perform its intended function, which may be caused by one or more errors in the system. A bug is a mistake or flaw in the software that causes it to behave incorrectly. A fault is a potential cause of an error or failure in the software. A defect is a deviation from the specified requirements of the software, which can be identified through testing and may be the result of a bug

or fault in the system. Besides these terms, we may also use the terms *potential* and *real* to distinguish between issues that have the potential to cause errors or failures (but have not yet been observed to do so) and those that have been observed to cause errors or failures in the software. Throughout this document, we will refer to these terms as necessary.

2.2 Automatic Test Generation in the literature

The high cost and effort involved in the testing process during software development have made automatic test generation an active topic in the literature. Nevertheless, there is a documented gap between academic and practitioner (e.g., industry) views on such tools and techniques [77]. This is notably caused by the limitations that still exist in this domain [6]. For instance, current solutions struggle to find real faults [5] and engineers often reject automatically generated tests due to the difficult readability of such tests [84]. However, some tools have demonstrated an exceptional ability to automate the generation of test cases from different approaches and programming languages.

Two of the best known Java tools are EvoSuite and Randoop. The first one, Evosuite [34], generates test cases specifically designed for object-oriented programming through an evolutionary approach. It means that, during the automatic construction of the tests, it uses a Search-Based Software Engineering approach by applying a genetic algorithm. Its good results allowed it to win eight out of nine editions of the International Workshop on Search-Based Software Testing (SBST) test generation tools competition [67, 93]. The second tool, Randoop, follows a completely different approach to generate tests: feedback-oriented random testing. This technique not only generates random or semi-random data (from a seed) for test execution, but also incorporates some knowledge of the tester about the software under test. Thus, the generated tests are more readable and effective than those generated by a purely random and automatic procedure. The random testing technique has shown to be effective in finding real bugs in Microsoft .NET code [65]. Randoop identified errors in the component that previous testing had missed, completing the task significantly faster than a typical test engineer could, including time spent reviewing the tool's results. In addition, the tool enabled the test team to uncover errors in other testing and analysis tools and deficiencies in previous best-practice guidelines for manual testing.

Regarding the C and C++ language, one of the most widely used and supported tools in recent years is KLEE [13], a tool that generates unit tests based on Dynamic Symbolic Execution (DSE). Before running KLEE, the software under test is compiled in LLVM bytecode with an appropriate compiler (e.g., Clang). It is also convenient to indicate which variables will be symbolic, either by selecting them directly in the code or by delegating this selection to the tool in those arguments received via the command line. By using a variety of search strategies to guide the exploration of the entire code, KLEE can check each of the conditional branches in it. More information about this tool, its underlying technique and other tools that incorporate this technology are discussed in Section 2.3.3.

2.3 Software Testing Techniques

Software testing techniques are methods and processes used to evaluate the functionality, reliability, and performance of a software application. These techniques aim to identify defects, bugs, and other issues that may impact the quality and effectiveness of the software. There are various levels and objectives of software testing, including unit testing, integration testing, system testing, acceptance testing, and performance testing. Each level or objective has its own set of goals and is applied at different stages of the software development life cycle. Effective software testing is crucial for ensuring the overall quality and integrity of a software application, and it helps to improve the user experience and reduce the risk of software failures.

2.3.1 Classification of software testing strategies

There are several classifications of software testing techniques [58], including black-box and white-box testing, which refer to the level of visibility of the internal workings of the software being tested [62]. Black-box testing involves testing the software from the perspective of an external user with no knowledge of the internal implementation, while white-box testing involves testing the software with knowledge of the internal implementation, including the source code. Other classifications of software testing techniques include functional testing, which focuses on the intended functionality of the software, and non-functional testing, which evaluates the performance of the software and other

quality properties such as usability and security [9]. Structural testing is a technique that tests the internal structure of the software, and regression testing is used to ensure that changes to the software have not introduced new defects [48]. We can also classify software testing as either static testing, which involves analysing the software without executing it, or dynamic testing, which involves executing the software and observing its behaviour [41]. These various software testing techniques are essential for ensuring the quality and effectiveness of a software application. In this work, we will focus on black-box and white-box testing as they are two of the most widely used software testing techniques related to this PhD thesis.

A well-known black-box testing technique is random testing [30]. This technique involves generating random data for a specific set of test cases in an iterative or searching manner. After a certain number of generations, which can be determined by the user or determined by the rool, it is verified how many tests pass or fail. A variation of this technique is adaptive random testing, which provides feedback to the random data generator to somehow generate the data in a semi-random way and with a certain level of guidance, aiming for better results [19]. If this information comes from the implementation of the method under test, it could then be considered an evolution towards white-box testing. This combination of black-box methods with a white-box part is known as grey-box testing [46].

Another popular technique for test generation is the application of genetic algorithms [75]. This technique seeks to evolve the initial data in search of improved versions, capable of obtaining effective results. It is a white-box technique when the algorithm uses information from the source code, while it is a black-box technique when it uses information from other sources, such as the requirements. Its use in automatic test generation has led to good results in languages such as WS-BPEL [33] or Java [34].

Lastly, there are also two well-established techniques in the scientific literature that are worth mentioning. One is metamorphic testing [83], a technique whereby new test suites can be derived from existing ones, based on conditions that the method to be tested must always meet, known as metamorphic relations. For example, if a method has to produce the same result when given the same input even when the order of the input has changed, a metamorphic relation can be created to test this condition. This technique is especially useful for solving the oracle problem, where it is difficult to determine whether a system

under test has produced the correct output. Then, there is mutation testing [71], which is a technique that introduces small modifications in the software to be tested in order to evaluate the quality of the test cases. Later in this chapter, we will see in detail how mutation testing works and how this technique can guide the improvement of an initial set of test cases.

In the following, the two testing techniques used throughout the thesis project are presented: mutation testing and dynamic symbolic execution.

2.3.2 Mutation Testing (MT)

MT is a fault-based white-box technique [71] to evaluate the quality of test cases by introducing slight changes, known as mutations. This technique was first presented as a new approach to traditional software testing, aimed at helping programmers create test cases that are able to detect more potential defects [25, 26]. The programs that result from applying the mutations are called mutants. These changes are implemented through mutation operators. Each mutation operator is a series of instructions that describe the change made to the original program. Mutation operators are classified by purpose (e.g., change of class modifiers or replacement of arithmetic operators). This technique is usually applied with traditional mutation operators. These kinds of mutation operators introduce simple changes in arithmetic and logical expressions or typical control structures in structured imperative languages. It aims to replicate real coding mistakes made by programmers. Listing 2.1 is an example of a mutant produced by an operator called `ARS`, where the arithmetic operator `++` has been replaced by `--`.

```
int exampleMT () {
    int a = 10;
    //a++;
    /* ARS */ a--;
    return a;
}
```

Listing 2.1: ARS mutation operator example

A successful test suite should be able to detect the faults that might be present in the code. According to this logic, at least one test case in the test suite should produce

a different output when run on the mutants that change the semantics of the original program. When the test cases detect a mutant, it is said that the mutant has been *killed*, otherwise, it is known as a *surviving* or *alive* mutant. The evaluation of the test cases is done by means of the mutation score. This metric, which ranges from 0 to 1, correlates with the ability of the test cases to kill mutants.

$$S = \frac{K}{M - E} \tag{2.1}$$

Formula (2.1) shows how this score is calculated, with S being the mutation score, K the number of killed mutants, M the total number of mutants, and E the number of equivalent mutants. A mutant is equivalent when its behaviour does not differ from the original program. It may happen that the mutation affects a piece of code that is never reached, or that the semantics of the program remains intact. These are changes that cannot be detected by test cases. Equivalent mutants can affect the reliability of the final result [40], so great efforts are made to identify them before calculating the mutation score. As for detecting equivalent mutants, one of the best-known techniques is Trivial Compiler Equivalence (TCE) [70]. This is a technique that allows detecting equivalent and duplicated mutants through the source code optimisations provided by the compiler. Two or more mutants are duplicated when they are equivalent to each other, but not necessarily to the original program.

There are many tools available for performing mutation testing in various programming languages, such as the ones surveyed in [24, 50]. In this work, we will focus on MuCPP [24], a mutation testing tool for the C and C++ languages that implements different types of mutation operators. We will delve into the details of MuCPP and the mutation operators it implements in Chapter 5.

2.3.3 Symbolic Execution

When analysing or testing the functionalities of a program, it is common to run its functions and methods several times with different values. Normally, a specific set of values runs a particular path of the code. Symbolic Execution is a technique that allows more than one code path to be explored simultaneously. From its origin in the 1970s [49] to the present day, this technique has been adapted in the field of code debugging and

software testing [7]. The inputs to the program are symbolic values. This representation implies any value, without specifying a concrete one, which in practice is equivalent to numerous concrete values. A symbolic execution engine handles this type of execution via two fundamental elements: a formula with a set of restrictions to be met during the execution and an index or dictionary that maps code variables to their symbolic values. During the symbolic execution period, a solver [23, 79] checks that the properties of the code are not broken and verifies whether the formula has a solution. Otherwise, it discards the formula.

Listing 2.2 shows an example function written in C with two execution paths: one that throws an exception (path 1) and another that returns a value (path 2). To make this example as illustrative as possible, we will perform a theoretical analysis without relying on any tool. A manual execution with concrete values would have the behaviour shown in Table 2.1.

```
1 int example(int b) {
2     int a;
3     a = b * 10;
4     if ( a == 20 ) {
5         throw Exception( );      //Path 1
6     } else return a;             //Path 2
7 }
```

Listing 2.2: Example code

Input	Output	Path explored
b = 4	40	Path 2
b = 0	0	Path 2
b = 2	Exception	Path 1

Table 2.1: Example of concrete execution

In this example, there is only one input value capable of throwing the exception: `b = 2`. Any other input will cause the function to return its value multiplied by 10. An example like this can be easily analysed at a glance; however, in more complex programs, it can be difficult to realise that there is one value that triggers a different behaviour than the others.

Execution with symbolic values acts differently. This symbolic value is assigned to `b`, without being specified concretely. When the symbolic execution engine reaches line 4, it performs two formula evaluations: '`b` * 10 == 20' for the first path and '`b` * 10 != 20' for the second. Note that the formula is built up as each line of code is analysed. This continues once the symbolic execution engine analyse line 4, as it will now have two distinct formulas that can be further expanded according to the elements present in each

path. Eventually, it should be possible to solve both formulas to obtain concrete values that run each path. The values in this case could be 2 and any value other than 2, such as 4 or 0.

Symbolic Execution is a technique that allows test cases to be generated with several values. However, it has some limitations, such as the possibility of generating unsolvable formulas (due to, e.g. unavailable external dependencies) or an unmanageable path explosion, where the time needed to solve the formulas is unmanageably long. Currently, there are solutions that not only apply this technique but also focus on overcoming these limitations [7] by providing solutions focused on the automatic generation of test cases. These solutions include randomised path exploration and Dynamic Symbolic Execution (DSE).

DSE combines traditional symbolic execution with concrete execution (actually running the program with specific values), generating a set of values that can be used as test data. Thus, it facilitates path exploration while maintaining strict control of the execution flow in the code. This combination of techniques is also known as *concolic execution* (**conc**rete and symb**olic** execution). When applying DSE to generate test data, the path explosion challenge is combined with the need to achieve a certain level of code coverage. Achieving high code coverage is especially important when using DSE, as this technique generates test cases by exploring different execution paths in the program. This can result in a large number of test cases being generated, making it more difficult to ensure that they all provide adequate coverage of the code. Unsolvable constraints, in this case, should not be a major problem, as the symbolic execution engine will discard them and not generate any test case. When automatically generating test cases, it is convenient to use a metric to know the scope of the test cases in the software under test.

It is possible to find in the literature several works that use DSE as the main technique for the automatic generation of test cases. One of the earliest works in this area is DART [38], a technique that combines DSE with random testing, another well-known testing technique. Combining these two techniques and including model checking techniques, it is able to execute as many paths in the code as possible and generate a set of unit tests for programs written in C with a high structural coverage. One of the most used and updated tools in the research community over the last few years is KLEE [13] (an evolution of EXE [14], another earlier tool), which applies DSE to generate unit tests

for C and C++ code. This tool uses the LLVM bytecode derived from the program and a few small indications about which variables or inputs should be symbolic. Through the use of search strategies to guide code exploration, KLEE generates a set of unit tests with a high structural coverage. Proof of its effectiveness is that it has been the winner of several testing tool competitions [11].

Other tools rely on the limitations of DSE (mainly path explosion and code coverage) for the development of new solutions that overcome them. KLOVER [97] is a framework for automatic test generation in industrial systems that integrates KLEE at its core. These types of systems, which are often very large, tend to generate a number of paths that hinder the application of DSE. KLOVER overcomes this limitation by selecting and exploring those paths that are most likely to generate test cases, avoiding those that have already been explored or do not lead to any new test data. Another tool to alleviate the path explosion issue is Much [64]. It is a framework that combines DSE with fuzzing [54], a technique that introduces invalid test data into the test cases to find bugs. The combination of these techniques achieves greater code coverage than both techniques alone.

There are several other tools dedicated to software testing via Symbolic Execution, such as Manticore [57] for smart contracts, DeepState [39] for C and C++ programs and mCute [3] for UML state machines.

2.3.4 Exploring the Benefits of Combining Dynamic Symbolic Execution (DSE) with Mutation Testing (MT)

In recent years, there have been several studies in the literature that combine DSE with MT to achieve different goals. One of the first works that lays the foundations for the combination of both techniques comprises the generation of test data using algebraic constraints and mutants [27]. Specifically, the authors introduce a set of mutations in C and Fortran programs along with a set of algebraic constraints that must be satisfied to kill the mutants. Solving these constraints results in test data that, when used as inputs to the program, are likely to have a high probability of killing the mutant. Although this tool does not use DSE, it shares similarities with this technique, since in both cases constraints or formulas are solved to get test data. The differences with this PhD thesis are notable, as they rely on an external program (Godzilla) to satisfy the algebraic

constraints and use a different set of mutation operators. Our proposal incorporates DSE to automate the constraint generation and resolution process, applies to C and C++ programs and incorporates current MuCPP mutation operators, getting results that are more comparable with other more up-to-date works.

Another approach is to use DSE to improve other testing techniques, such as MT. In fact, the combination of DSE and MT has been shown to be effective in improving the quality of test cases [69]. In this work, the authors use DSE, among other techniques, as auxiliary to introduce mutations in certain areas of the programmes under test. To verify the feasibility of their proposal, the authors use three different test generation techniques on a set of programs and check the mutation score of the tests generated by each one. This score turns out to be low, despite obtaining good results in terms of branch coverage, which coincides with part of the conclusions got later in this PhD thesis. The authors then propose a technique to reduce the application of mutants to the branches covered by the different test techniques, as well as a framework to introduce weak mutation at these points. In their study, introducing mutants in the automatic test generation process considerably increases the mutation score, coinciding with part of the results of this work. Other authors use DSE as an auxiliary technique to classify and detect equivalent mutants [37]. By focusing on a subset of mutation operators, the authors achieve good results by automatically detecting equivalent mutants. The difference with this PhD work is that the authors combine DSE with MT to improve the application of MT, whereas our proposal involves improving DSE with the help of MT.

Other works aim to improve the automatic generation of test cases by combining DSE with MT. The extension of DSE to generate test data focused on killing mutants is an approach that achieves good results in killing most non-equivalent mutants [68]. In this work, the authors develop a process that uses mutant schemata and control flow graphs to produce conditions suitable for DSE to reach the mutation point more easily, getting over 85% mutation coverage in their case study. The main difference with our PhD work is that we combine DSE with MT without using external elements. From a basic implementation where none of the techniques is modified, to possible more advanced implementations that introduce MT elements in DSE, as shown later in Chapter 5. The results are not comparable: while the authors select in their study five tools of small-medium size (between 40 and 500 lines of code), in this PhD thesis we use a set of utilities that, together, reach thousands of lines of code.

In addition, three tools that combine DSE and MT achieve good results in terms of test generation: PexMutator [100], SEMU [18] and SAFL [94].

PexMutator [100] is a test generation tool that employs MT and DSE to create a meta-program from the SUT and embed mutations within specific constraints. This allows DSE to generate test cases that target the constraints and effectively kill most mutants. In its case study, PexMutator can kill up to 80% of non-equivalent mutants. However, this level of effectiveness cannot be directly compared to our study, as the authors use a different set of mutation operators and test a limited set of 5 utilities drawn from a real library ranging in size from 1 to 12 methods. In contrast to this study, we do not plan to use meta-mutations in our approach. Instead, our aim is to integrate the concept of MT into DSE, enabling it to run the original program and generate enhanced test suites in terms of mutation coverage without incurring in the additional cost associated with the generation and execution of mutants.

In SEMu [18], the authors aim to leverage surviving mutants (i.e., stubborn mutants) to improve DSE, but at a lower cost than exhaustive exploration. To achieve this, they implemented SEMu as an extension of KLEE, which combines various strategies to make the process more efficient, such as the use of meta-mutation and a suite of heuristics to reduce the number of paths to explore. Through the use of meta-mutation, paths shared by the original program and the mutant are not explored again in each mutant. The optimisation of the heuristic search allows for more efficient use of the search budget to explore paths affected by the mutation. As their approach does not involve exhaustive exploration, this increases the likelihood of finding an infected state that also propagates to the outputs. There are some differences between their work and ours. For example, SEMu mutates code at the bytecode level, whereas our MuCPP tool operates at the source code level, and the authors of SEMu use test cases manually developed by the authors of GNU Coreutils as a seed in the tool's execution. The authors of SEMu reported better results than KLEE to kill stubborn mutants in a selection of Coreutils utilities. Like us, they also reported high costs and had to discard several Coreutils utilities because of these costs. While their approach improves the performance of the technique, the number of stubborn mutants still limits the overall time present. The more exhaustive the configuration for the heuristic search, the more paths are explored and the higher the computational cost becomes. Therefore, there is still potential for

improvement if we can take advantage of the information provided by mutants without executing DSE for each mutant.

Finally, in SAFL [94], DSE is first used to generate some initial seeds, which will later feed a mutation-based fuzzing process. SAFL proves to be an efficient tool that can explore deep paths easier and earlier thanks to both its fuzzing algorithm and the classification of seeds according to the path coverage. However, the data is fuzzed without being aware of the exact mutations injected in the program, so the approach is in principle limited for killing some more sophisticated mutants, especially in complex software systems (e.g., those that require some specific values for different variables before reaching the mutation). As with previous work, it is difficult to compare the set of tools used. We should note, however, that they use DSE to get an initial set of qualified seeds instead of random ones for the later fuzzing process, while we aim to employ DSE as the main technique to generate test cases able to detect faults.

The aforementioned work has successfully showed the efficacy of combining DSE and MT, which is an important step forward and motivates further development of these techniques. These studies also serve as a complementary resource to advance the family of techniques presented later in Chapter 5.

2.4 Abstract Syntax Tree (AST)

The Abstract Syntax Tree (AST) is a textual tree like representation of the source code received by the compiler. The syntax is said to be abstract because it does not show all the details appearing in the actual code, but it focuses on the structural content of the code, as well as on different details related to variables and other similar elements.

Regardless of the language, the compiler performs a series of steps to transform the source code into an executable file. The AST appears in the beginning of this process after these two first steps [4]:

1. **Lexical analysis**: the compiler takes the source code (usually modified by the language preprocessors), written in form of sentences. A lexical analyser is responsible for decomposing this syntax into sets of lexemes, known as tokens, so that elements such as comments, line breaks and whitespaces are removed. It is possible for the

lexical analyser to encounter an invalid token, generating an error at that moment. As the lexical analyser checks that the tokens are valid, these are sent to the syntax analyser, usually on its demand.

2. **Syntax analysis**: this is the step where the AST is created. The tokens received from the previous step are organised in order to build the tree that represents the actual structure of the code. It is a step similar to natural language, where we connect a list of individual words (in this case tokens) into a structure that represents ideas, such as sentences (in this case the tree). Similar to sentences, where we identify the function and hierarchy of each word (verbs, adjectives, nouns, etc.), the AST allows us to identify the purpose and hierarchy of each element of the code (methods, functions, classes, variables, etc.).

Once we have the AST, is it not only possible to understand the structure of the code more easily, but we can also manipulate it for any purpose, such as optimising the code or finding bugs in early stages of development.

Figure 2.1 shows an example of what an AST looks like with a graphical interpretation. The function in which it is based performs a sum between two values. It can be observed that the root node is of type *FunctionDeclaration*, as it is a declaration of a function. The three child nodes are the *id*, whose child in turn will be the name of the function, *params*, whose children will be the parameters received by the function, and finally, *body*, whose descendants form the body of the function. This interpretation allows us to observe at a glance the structure of the code, its hierarchy and its dependencies.

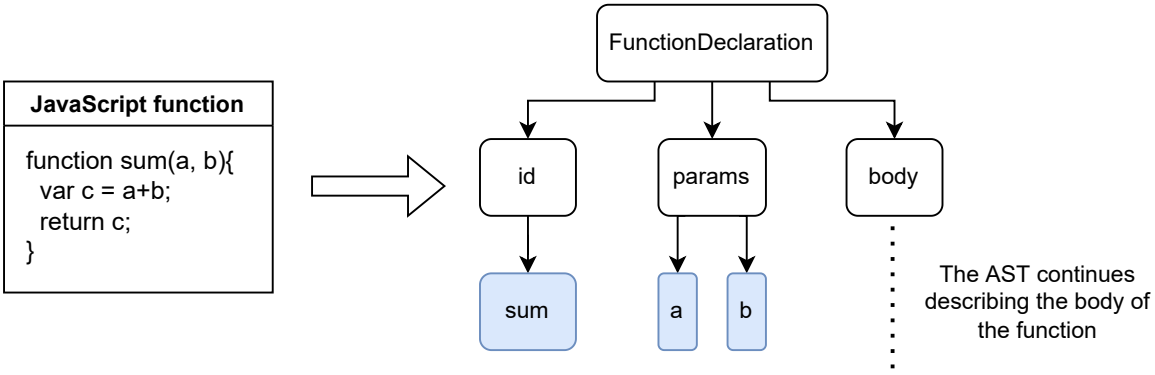

FIGURE 2.1: Example of an AST tree

The AST is an intermediate component in the compilation of programs, regardless of the programming language (although different compilers may use a different syntax to

represent the nodes of the tree). Its use, both to analyse and modify existing code as well as to produce completely new code, is widespread in the literature for different purposes. One of the most widespread uses of AST is the detection of similar code (commonly known as clone, equivalent or plagiarised code). Different authors take advantage of the characteristics of the AST to analyse the structure of the code and its elements, in order to determine whether two pieces of code are similar.

Wang et al. [95] construct a graph to reflect syntactic and semantic information of the ASTs generated from the codes to be compared. Specifically, the graph links the ASTs with explicit details of the flow of information. With the help of graph neural networks, they are able to compute a vector representation of the code, which finally allows them to measure the similarity of the code pieces by comparing these vector representations. Their experiments on real datasets show that the detection of code clones is improved compared to the direct comparison of unmodified ASTs. This allows the code to be optimised by identifying and eliminating similar code, reducing the number of final lines to be compiled.

Situations such as the emergency caused by COVID-19 have made plagiarism detection tools more valuable, as students, working from home, could be tempted to copy code in the absence of direct teacher supervision. AST is useful in this regard, authors as Fu et al. [36] propose WASTK (Weighted Abstract Syntax Tree Kernel), a method for detecting code plagiarism that takes the AST as a fundamental part of the process. Their method not only takes into account the similarity of two code pieces, but also considers the context in which the programs are asked to be developed. In their experiments, using a dataset composed of real submissions, they are able to detect plagiarism more effectively than other solutions that do not use AST.

Other authors take this concept further and take advantage of AST features to find clone code written in different languages [74]. Using a semi-supervised machine learning model trained on the ASTs produced by the programs, they are able to detect whether two code pieces have similar behaviour, even if they are written in different programming languages. In their case study, they are able to detect clones written in Java and Python.

It is possible to make modifications to the AST, so it is not always necessary to use the default AST as provided by the compiler. Zhang et al. [99] propose a modification of AST known as AST-based Neural Network (ASTNN) so that the code, instead of

being represented by a single large AST, is represented by a sequence of smaller ASTs. From this sequence, they use a bidirectional neural network model to produce a vector representation of the source code, taking advantage of the naturalness of the sentences. Finally, the authors evaluate this new representation of the AST using it for clone code detection and code classification. The results show that this ASTNN improves the overall performance compared to the classical AST, resulting in an interesting evolution of the AST.

Other authors make use of the AST with different goals, such as intelligent code completion [96] to help programmers write code faster and with fewer typos, detection of equivalent mutants [73] and even the incorporation of mutants in the program itself [98]. All these applications of AST serve as inspiration for its use in this work, where it will make possible the analysis and generation of new code pieces in the field of software testing.

2.5 Industry 4.0

Industry 4.0 is also known as the fourth industrial revolution. To comprehend this, it is worth bearing in mind the evolution that industry has undergone through the industrial revolutions over the course of history. Freeman and Louçã [35] describe in detail all the stages of each industrial revolution, from the first, where new materials were introduced (chiefly iron and steel), to the second, which changed the way of working in industry, and the third, which introduced new information and communication technologies, through the incorporation of the Internet and the development of renewable energies.

The concept of Industry 4.0 is presented for the first time during the Hannover Messe in 2011 [45]. In this document the authors analyse the impact and evolution of current systems in such a way that they see signs of a new industrial revolution that will mainly affect the way industry interacts with people, the development of science and political actions. Since then, many of these estimates have become a reality.

A few years later, already immersed in Industry 4.0, Lasi et al. [51] describe in great detail the social, political and economic changes that the fourth industrial revolution is bringing about. These changes include shorter development periods compared to traditional

development in manufacturing processes, product flexibility due to new production requirements, decentralisation of processes through the reduction of hierarchies and greater efficiency in the use of resources, thanks to the particular economic and ecological context of the 21st century society. In terms of technology, all machines are moving to a smart environment where data plays a fundamental role. The application of new technologies makes it possible to take decisions quickly and even predict the behaviour of operations, thereby reducing the risk of defects. Industry 4.0 tries to integrate all the elements in a safe, accessible and organised way through cyber-physical systems. These are industrial automation systems that extend their functionality thanks to networking and their access to other systems, so that their operation is affected by the surrounding environment.

Figure 2.2 shows a summary of all the fields covered by this industrial revolution based on compilation of Industry 4.0 works [55]. However, this PhD thesis focuses on the field of software development, which is a transversal competence present in all stages.

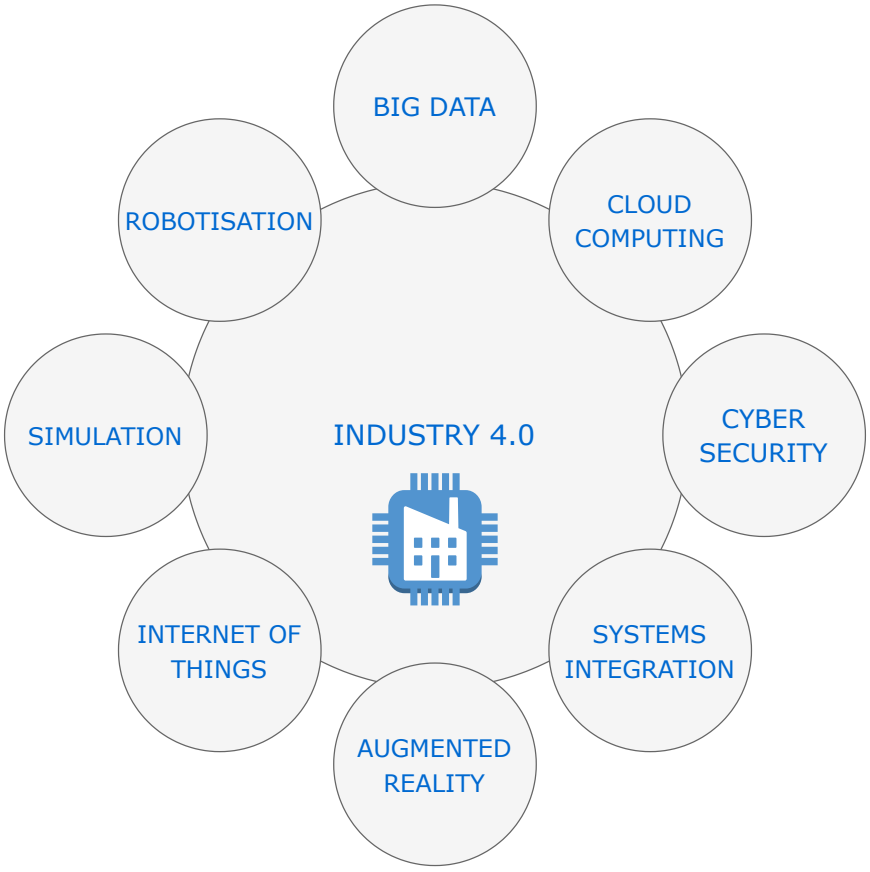

FIGURE 2.2: Industry 4.0 schema

Most of the elements involved in this industrial revolution have a software element embedded in them. Therefore, software development necessarily plays a key role in the implementation of the different technologies. Industrial software, unlike that developed for traditional projects, faces a series of challenges caused by the very nature of Industry 4.0. For instance, the concept of Big Data is associated with the need for software capable of handling huge amounts of data. In cybersecurity algorithms, software must be robust to prevent unauthorised accesses or data leaks. Simulation applications, on the other hand, must have efficient algorithms able to represent reality as faithfully as possible in real time. Finally, software related to the Internet of Things (IoT) must be able to operate in heterogeneous systems and transport information efficiently. These are just a few examples of the situations that software engineers face in this new context.

All this software needs to work properly when it goes into production, as their malfunctioning can have serious consequences. Software testing is one of the solutions proposed to find defects that may be present in the software. In the context of Industry 4.0, with an increasing number of interconnected subsystems, encompassing various technologies, automation becomes essential. Automation allows the mechanisation of the testing of large systems and subsystems, helping integration and validation before final delivery.

2.6 Software testing in Industry 4.0

This section discusses the role of software testing in Industry 4.0, based on the key characteristics of Industry 4.0 information technology (IT) projects. It also shows how the application of software testing can help to improve projects in every aspect.

2.6.1 The transition of software testing to Industry 4.0

It is known that software testing is an essential stage of any software development project. The different techniques make it possible to be confident in the quality of the software developed. However, the evolution of technologies driven by Industry 4.0 has led to the emergence of testing needs that might not have been considered in a classic vision of software development. It is worth considering the role of software testing in Industry 4.0 in order to understand how it should move forward to adapt to current and future paradigms.

The role of software testing in Industry 4.0

There are numerous techniques that allow software testing to be applied to projects in general. With regard to IT projects within the scope of Industry 4.0, even though each one will have its own specific needs, there are a series of common characteristics that should be met in order for the software to successfully reach the production phase. It is possible to classify these characteristics into different fields of study [32]. In our particular context, they can be divided as follows:

- *Data collection, cleaning and analysis:* it is essential to be able to acquire and homogenise all the data generated in the production facilities. It is not only a matter of collecting the raw data, but also of filtering and analysing it in real time. This is where Big Data has its place [47]. This paradigm, inherent to Industry 4.0, requires software capable of digitising and formatting information, storing it and making decisions swiftly and automatically. A defect in this kind of software could lead to data loss or corruption and erroneous predictive models, thus directly affecting decision-making in project management.

- *Security of IT systems:* related to the data collection feature, one of the new developments in Industry 4.0 concerns remote access to the information in factories. This presents new challenges, as factors such as information access control, encryption and software updates must be addressed. Likewise, some well-known threats are of particular relevance in this area. These include information theft, denial of service (DDoS) attacks or the transmission of malicious software within factories [21]. Software testing can act as a safety net in its own right, by detecting and correcting faults that could be exploited by attackers.

- *Device management:* interconnected devices must be able to include a routine procedure for their initial configuration. After this process, there is a need for regular checks, addition of new functionalities and, ultimately, disconnection and deactivation of devices. Software testing must be able to detect the incorporation, modification and deletion of new elements within Industry 4.0 projects. In this way, it contemplates a flexible set of devices, adaptable to the new needs that arise in the production processes.

- *The digital twin:* one of the key innovations of Industry 4.0 is factory simulation, which allows an easy integration of all kind of technologies virtually instead of physically. This makes it possible the digital simulation of the production and the detection of opportunities for improvement without impacting the actual manufacturing. The creation and management of this virtual factory, known as a digital twin [85], is done using specific process modelling and interaction software. The testing of this software needs to be adapted to this new approach to be able to check whether the simulations correspond to reality and, thus, allow for an effective decision-making.

We have seen how Industry 4.0 poses unforeseen situations in the management and development of traditional IT projects. Loss of information, incorrect simulations or security breaches are just a sample of the new challenges faced by software testing. Not only that, but systems are growing at a very fast pace, considering that around 90% of the data generated by companies has emerged in the last few years, and this amount is expected to double in a very short period of time [82].

Software testing needs to be able to cope with such situations. Cryptography algorithms, process simulations or the handling of sensitive data on a large scale represent further challenges that naturally call into question whether current testing techniques are up to the challenge.

Suitability of current testing techniques

There are more than a few existing testing techniques beyond what is seen in this work, and it is possible to find different approaches. Their application is not limited to the experimental field, but some of them are successfully used in industry. As an example, we can see that search-based test generation, using Evosuite, has been evaluated in an industrial context [5], detecting up to 56.40% of the identified faults in a financial software. Random testing, using Randoop, has been applied to different industrial tools [78], with success in finding bugs in most of them.

Despite the many studies concluding with promising results, it seems that industry may be reluctant to integrate such tools into their development processes. As we can see in another study [6], this reluctance is caused by several and diverse issues. Among them,

we can highlight that many of the tools proposed to the industry are commercial and closed solutions, far from the open source philosophy. In general, these tools can involve very high adaptation costs and limited support. Other problems are the poor readability of the automatically generated tests, which makes test engineers reluctant to use them in real projects. Ultimately, this paper identifies the need for more usable tools in industry.

We envisage that the constraints will intensify due to the challenges posed by Industry 4.0. When we talk about software testing, in any of its types, we find common constraints. For example, the oracle problem [8], which consists on the lack of an agent capable of determining the correct output for a given input. This is because, in such a dynamic and changing environment, it can be complex to know with certainty what the expected results are.

Chapter 3

Software testing needs in industry

"Pay attention to negative feedback and solicit it, particularly from friends. Hardly anyone does that, and it's incredibly helpful."

Elon Musk

The good relationship between the University of Cadiz and the businesses in the Bay of Cadiz is reflected in the various *cátedras* (chairs) [87–89] and successful joint projects [1]. This translates into good opportunities for both industry and academia. In terms of research, it allows researchers to get closer to the real needs of the industrial environment and provides use cases that allow them to test the viability of their research beyond closed experimental environments. On the other hand, the industry benefits from cutting-edge and up-to-date research, helping it to improve the production processes, which is reflected in the creation and improvement of employment.

This chapter shows the needs of industry based on our experience with local companies in the Bay of Cadiz. It serves as a basis and motivation for Chapters 4 and 5, whose work arises from the experience described below.

3.1 Motivation

As we have seen in Section 2.5, *Industry 4.0* is a transformation of the entire industrial domain. Manufacturing and human-machine interaction are being shifted towards a connected environment known as *smart factories*: plants that dynamically adapt to changing industrial needs [43]. A necessary evolution in IT systems comes with this transformation, bringing particular value to the quality assurance process of software projects. Software testing tries to find as many defects as possible during software development. Often, in industrial projects, this phase is linked to tight deadlines and high costs. For this reason, in addition to the resources and expertise required, it may sometimes not be given adequate attention [58].

On other occasions, we find companies with an exhaustive and highly mature Quality Assurance System [60], where great efforts are dedicated to completing the stages of the project validation and verification process. In the industrial fabric of the Bay of Cadiz, we find companies dedicated to the construction of large means of transport, such as vessels and aircraft. Such companies develop large software projects integrating each of the components of the means of transport they manufacture. Based on our experience with these companies, it is common to find situations in which the different phases of software testing are conducted manually and in a repetitive manner, with considerable human effort. This issue gets more relevant in critical systems, where there is particular pressure in gaining confidence in the quality of the development, as any failure can have fatal consequences. This adds considerably to the cost of projects, which, based on their experience, can be as much as 40% of the total development cost. Despite this, software testing is of great importance and cannot be disregarded. *Industry 4.0* introduces complex and interconnected systems that could face problems never considered before, which promise to further increase the software testing needs. Therefore, in order to reduce the high cost of a manual application of testing, there is an increasing trend towards automation.

This chapter describes the knowledge acquired as part of the experience of collaboration between companies in the Bay of Cadiz and the *University of Cadiz*. This experience has been crucial in identifying and understanding the real needs of existing companies and serves as the initial impulse for the work carried out in Chapters 4 and 5. The main motivation lies in reducing the cost of industrial projects by automating the software

testing phase. Nowadays, there are different free and commercial applications; however, none of them seem to be fully suited to the needs of the companies surveyed. One of the main problems with the use of these applications is that there is no guarantee that the integration with their systems will be maintained in the future, as well as the fact that the purchase of licences often entails high additional costs. For this reason, companies are looking for a customised solution, adapted to the changing needs of the industry.

3.2 Test generation in industrial environments

In this section we discuss the challenges that current software testing techniques face in Industry 4.0 projects. Next, we describe two aspects related to the application of software testing in industry: the stages of the industrial test generation process and the potential benefits of its automation. This information is based on the knowledge acquired from some of the projects studied in our experience with companies in the Bay of Cadiz. Section 3.2.2 describes the aforementioned stages as well as it mentions different technologies broadly known in research work but not so common in industrial work and which could be used to improve the workflow. This collaboration aims to bring the automation approach in line with the needs of Industry 4.0. In addition, unlike other solutions with commercial products, which may have a specific, closed purpose, this project looks for a modular and extensible solution that can be adapted to future needs.

3.2.1 Software testing challenges in Industry 4.0

To understand how current testing techniques should be adapted to Industry 4.0, we will consider the key challenges faced by any IT project in this paradigm.

In Industry 4.0 IT projects, there is usually a wide diversity of devices. The rise of technologies such as the Internet of Things (IoT) and smart factories means that we find systems dependent on a large number of different devices. While this may simply seem a technical issue, it has significant consequences. For example, consider a smart factory where we have a digital twin. This consists of several software applications installed on a particular system that, based on its simulations, should be able to communicate with the machines in the supply chain and the management team. At the same time,

the machines will be connected to other devices handling the information and making it available to the people. With all of this information, the management team can then make better decisions and share them with the development team, who will have their own devices for this purpose. Normally, each of these elements has a software associated. Even if the software is in a different environment, there is a need for an independent software testing process, capable of adapting to the different situations that may arise.

Current testing techniques shall be able to adapt to these situations. When dealing with Industry 4.0 projects, software testing needs to be done with a vision that goes beyond finding possible defects in the functioning of the tools. Instead, let's focus the testing stage on two fundamental aspects of production: the production chain and the personnel.

- *The production chain*: software testing shall be adapted to a multi-platform and multi-disciplinary environment. In addition to intensifying integration tests, it is necessary to check for faults in the different platforms, making it necessary to create complete test scenarios, where not only the project is tested but also its operation with the configuration of the environment with which it interacts. The ability to reproduce errors is particularly important when dealing with different environments.

- *Personnel*: all the actors involved in IT projects are considered; the management team, the development team and the final customers. Software testing shall be adapted to take into account the different roles, simulating the different scenarios in which the software can be put into operation. This will reduce the risk of discrepancies between developers and their customers' needs. It can also take a more 'negotiating' role than in traditional development, serving as a communication channel between the two. Developers will be able to obtain usage data for their application in testing, without any detriment to customers.

These factors are directly influencing the emergence of new testing techniques, designed to cover new needs not contemplated in traditional software development. A good example of this is *crowdtesting* [53], which consists of the use of software under development by a large group of users who provide information on all kinds of errors, vulnerabilities or details for subsequent improvement. Its use in Industry 4.0 can bring great benefits, as the development team will have access to information on usage and feedback from

customers. This makes it possible to detect and correct potential bugs faster, as well as to adapt the development according to the customer's needs in real time. The more users participate in the tests, the more complete the information will be and the easier it will be to advance in the development of the final product.

Crowdtesting, along with the modifications that traditional testing techniques may undergo, are a direct effect of Industry 4.0, which, far from being a simple theoretical paradigm, is increasingly present among us. Software testing is evolving to an approach where the technical and personal aspects of IT projects are taken into account.

3.2.2 Stages of industrial software testing

Software testing in industrial environments is apparently similar to that of any other kind of software, mainly differing in the size and granularity of the process. Interviews with the surrounding companies referred to in this chapter allowed us to identify four major activities to which they devote a large part of their resources: (1) source code analysis, (2) test generation, (3) data generation and (4) industrial validation. These activities are described below:

1. *Source code analysis*: this analysis consists of obtaining information from the source code, either manually or using static analysis methods. The C and C++ languages are well established in the industry and its projects, as they are well known for their support for legacy code, their efficiency, ease of maintenance and code reusability. It is common for this analysis to be performed manually, often by the developers themselves, which can be a practice that introduces errors and a high workload due to the large scale of projects. Therefore, this stage can benefit from the use of automated and more robust analysis techniques, reducing the workload and producing reliable and verifiable results.

2. *Test harness generation*: after the previous stage, test engineers are familiar with the classes and methods in the code, so it is time to design the so-called test harnesses or test skeletons [63]. These test harnesses, normally accompanied by initial data, allow the industry to define its test case strategy, subsequently allowing the inclusion of data from external files, the reuse of test cases in different projects

and their maintenance by different work teams. Test harnesses can also be represented conceptually, so that the transition from one test framework to another is straightforward. In this sense, the most popular testing frameworks for C and C++ in the Bay of Cadiz industry are *BOOST*[1] and *Google Test*[2]. After a report, the engineers try to fix any errors found until the team is satisfied. Again, these tests are frequently designed in a manual way, based on the knowledge of the test engineers. This stage could greatly benefit from test automation techniques, significantly reducing the time and effort of the staff while resulting in a more robust set of test cases.

3. *Test data generation*: after source code analysis and test harness generation, efforts are mainly devoted to test data generation. One of the main advantages of this approach is that the same test harness with different data can be reused to test different functionalities of the code. This stage can be time-consuming and labour-intensive, depending on the number of test cases and the methods used to generate the test data. It is important to note that different techniques may be used to generate test data, such as manual creation or automatic generation using specialised tools. However, often test data generation is carried out more manually, and, in general, the lack of a complete and suitable solution means that testing often represents a burden for the team, especially considering the size of the projects being developed.

4. *Industrial validation*: consists of the complete testing of systems, both in real and simulated environments. This is particularly important as a failure in the system could have serious consequences. During industrial validation, various test adequacy criteria can be applied to ensure that the tests are thorough and effective. One example of such a criterion is mutation testing (MT). While these criteria can be useful in ensuring the quality of tests, they are not commonly used in industry.

To sum up, our experience with the companies in the environment has allowed us to identify a test development and execution stage divided into four different steps. The first one, initially, analyses all the intrinsic details such as classes, their dependencies, methods, etc. After this analysis, engineers develop test harnesses to design a more

[1] https://www.boost.org/doc/libs/1_66_0/libs/test/doc/html/index.html
[2] https://github.com/google/googletest

robust test strategy, which they can then use as a basis for the choice of test data. Throughout the process, industrial validation is present, which provides cross-cutting information to improve the process as it occurs. Likewise, the phases of test harness and test data generation can provide feedback to each other, in order to generate more refined harnesses or data, respectively. Figure 3.1 represents the flow followed by the feedback until the whole process is completed.

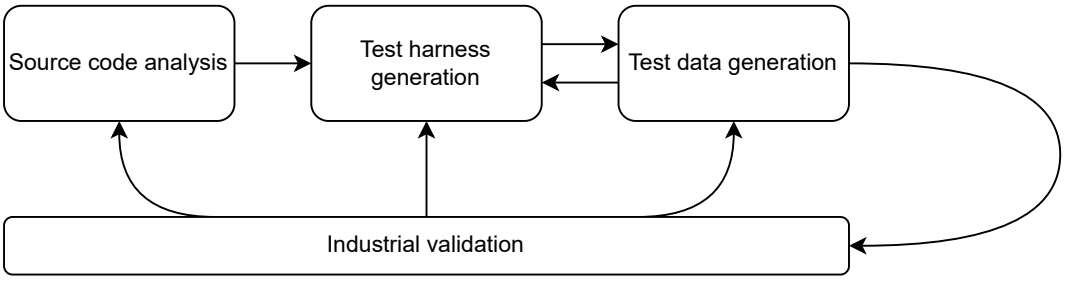

FIGURE 3.1: Interaction between different steps

3.2.3 Current limitations in software testing

All four stages are usually performed by different software solutions or even by different team members in a distributed approach. It is common to see how, due to the size of the projects, the four stages overlap and interact with each other, making the process not completely linear in almost any case. In this section, we review some solutions that aim to reduce the limitations and difficulties at each of the stages described above.

1. *Source code analysis*: in order to ease and improve the static code analysis, it is possible to study the *Abstract Syntax Tree (AST)*, a structure containing detailed information about each element of the source code. In particular, we highlight the AST generated by Clang [52], which is highly useful due to its similarity to the written C++ language and compatibility with most versions of the languages in the C family. Initially, the idea of using concrete syntax-based code processing was considered. However, after some initial testing, this approach was discarded due to the difficulty and potential for errors in pattern matching to cover all possible cases. The AST provides a well-structured representation of the code, making it easier to identify and analyse each element.

2. *Test harness generation*: thanks to the libraries implemented by Clang for source code generation and the information obtained from the AST, it is possible to build

a set of test harnesses written in a specific test framework, as well as a small set of initial test data (random or default depending on the type of data) that serve to check the functionality and viability of these harnesses.

3. *Test data generation*: the techniques range from the most basic, such as *random testing* [30], using random inputs for test harnesses, to more complex testing methods, involving more sophisticated techniques like genetic algorithms [80] or *Dynamic Symbolic Execution (DSE)* [12], which is an effective technique for generating comprehensive and high-quality test data, as it can automatically create a large number of test data that can handle complex input domains and manage dependencies between input data. In this sense it is interesting to explore the existing techniques and how they can be improved in order to find as many bugs in the code as possible at early stages.

4. *Industrial validation*: beyond manual inspection and strict standards [44], there are techniques for the evaluation of test cases such as mutation testing [71], which to date has not been integrated into this kind of projects.

The implementation of these solutions as a whole is a major challenge. A successful implementation brings considerable benefits to society as a whole, beyond those industrial projects where these stages are identified, which serve as inspiration for the development of the proposed solutions for automating software testing. Regarding industrial benefits, the following section describes in more detail the areas where a potential improvement is identified.

3.2.4 Benefits

Based on the close collaboration with the group of companies referred to in this chapter, according to the results obtained and their estimation, the incorporation of state-of-the-art technologies for automating software testing can be associated with three potential benefits: economic benefits, product quality benefits, and benefits in the entire development process. The following is a detailed description of these three benefits:

- *Economic benefits*: the overall cost reduction of projects is in the range of 6% to 13%. At first glance, this may seem a small percentage, but looking at the various

transparency portals [16, 61], it is common to find industrial projects costing up to billions of euros. Of particular note is the contract between the company *Navantia* and the Saudi Arabian government for the construction of five corvettes at an estimated cost of €1,800,000,000 [59]. Taking this well-known project as an example, the estimated economic savings would be between €108,000,000 and €234,000,000. A large part of this estimate is derived from the implementation of automatic test generation software and validation processes. This makes it possible to reduce the total time of the testing phase, detecting defects in less time, thus relieving the workload of engineers and avoiding the particularly high cost of fixing problems in the more advanced stages of development.

- *Benefits in product quality*: as a consequence of the economic benefits, there is an associated benefit in the quality of the final product. Techniques such as mutation testing automatically provide a numerical metric (mutation score) to estimate the ability of tests to detect real faults. This estimation is beneficial for documentation and comparison with other projects beyond knowing that the tests cover certain lines of the code or the subjective opinion of the test engineers. In this sense, projects will probably contain fewer defects upon delivery, producing a source code that is better designed and documented.

- *Benefits in the development process*: development time and the well-being of engineers are a clear benefit of automating software testing. Testing time applying automated techniques is significantly reduced when compared to the manual crafting of tests, making it easier to integrate other software components and software engineers throughout the development process. In addition, this reduction in time allows engineers to be able to take on a larger number of projects while reducing the overall manual workload.

3.3 Chapter conclusions

The experience with the companies in the Bay of Cadiz has been fruitful in terms of observation and information gathering. This close collaboration has resulted in a publication at a national conference [90], a publication at an international conference [92] and a book chapter [91].

Our experience allowed us to observe that the automation of software testing brings a number of important economic and quality benefits that should be taken into account in industrial software development projects. Although the stages of software testing in this context are well defined, they still rely on a very high manual workload and are often limited by external and uncontrollable factors such as deadlines or budget. Commercial solutions are often very costly and their integration between projects is not guaranteed. The evolution of the systems inherent to Industry 4.0 makes it impossible to overcome this issue, which is why there is a need for a new solution that is open, general and easily adaptable to the new circumstances.

After understanding the workflow described in Section 3.2.2, we note that normally the generation of test harnesses and test data are separated. Depending on the size of the project, they may be separated in time or by different teams. Although at first sight this may seem an obstacle, this has many benefits, as both exchange information, resulting in a generally more robust set of test cases than covering both stages at once. Therefore, two separate lines of research have been identified, one for each of these two stages.

The following chapters describe the research work conducted for the two lines of research. Chapter 4 presents a solution for the generation of test harnesses based on the Abstract Syntax Tree (AST), while Chapter 5 describes a novel combination of testing techniques for the generation of test data specially designed to find more potential defects in software.

Chapter 4

Automatic generation of test harnesses via AST

"Intelligence is the ability to avoid doing work, yet getting the work done."

Linus Torvalds

Automatic test generation involves the generation of test cases from existing source code. While it is true that test data can be obtained as a result of the conceptual analysis of the SUT design itself, without the need to view the software as a white box, the test structure itself (assertions, function calls, etc.) should be done based on the code syntax. From the industrial experience described in Chapter 3, along with the review of other similar works, it follows that the analysis of the AST is a natural solution for the generation of the test structure, as it provides the necessary information without the need to analyse the code implementation manually.

This chapter describes the development of a solution for the generation of test harnesses based on the information contained in the AST. This solution is strongly inspired by the industrial experience described in Chapter 3. As such, the product resulting from the research conducted in this chapter serves as a lightweight test framework adaptable to new projects and situations.

4.1 Motivation

It seems evident that a significant gap exists between the views of academics and practitioners in software development, especially at the software testing stage [77]. The good relations between the University of Cadiz and the business fabric of the Bay of Cadiz, as we have seen in Chapter 3, are an important source of inspiration for knowing and understanding the development needs that current industry projects may have.

A test case generation process, either automatic or manual, involves the development of two elements that, although closely related, are often separated: the test harness, which consists of a set of test cases and the infrastructure needed to execute them and evaluate the results, and the test data, which, while being part of the test harness, requires a more specific analysis of the software and its domain. To obtain the former, we mainly need structural information from the source code, while to obtain the latter, we need more detailed information on the inner logic, which can be obtained even at a theoretical level. Listing 4.1 shows an illustrative example.

```
1 class A {
2 public:
3     A(int value) : value(value) {}
4     int pow() { return value*value; }
5 private:
6     int value;
7 }
```

Listing 4.1: SUT example

This class allows initialising objects from an integer and provides a method that returns the value of the power of that number. If anyone were tasked with testing such a class, they would most likely design test cases based on two sets of questions:

1. *What are the constructors like? What methods and attributes does the class have? Are they public or private? What are the input parameters, and what type of data does each method return?*

2. *How does the mathematical power operation work? What values should I use to test this mathematical function?*

To answer question (1) the programmer needs to have knowledge on the language rules and the structure of the code. For example, how constructors work and what they consist of, what the accessibility of the class methods is, how a function call is performed, etc. These are trivial questions for any test engineer who has access to the source code specification. However, question (2) requires mathematical knowledge, independent of programming, in order to select a set of test data that we consider sufficient to be satisfied with the operation of the program (although this is sometimes more of a philosophical than a technical problem [58]). Of course, this does not mean that they are entirely separate questions, as often, and especially in complex problems, the design of the code helps to determine the test values.

A literature review in Chapter 2 highlights the importance of the AST in terms of automatically gathering structural information from source code. This information can be used to generate new pieces of code, which fits seamlessly with the creation of the test structure, including the initialisation of the objects of a class, method calls, etc. Moreover, the elements can be identified in a flexible way from the intermediate stage of the compilation process. This means that it is possible to generate code for different test frameworks with a single reading of the AST. Regarding test data generation, it is possible to parse part of the AST to obtain an initial data set. However, there are other powerful testing techniques that focus mainly on test data generation, fulfilling different coverage criteria. The application of novel techniques in this area provides motivation, as it will be later on discussed in Chapter 5.

Considering the experience described in Chapter 3 and the elements needed during the creation of a test suite, we propose the creation of a new framework for the automatic generation of test cases. This framework incorporates the AST to retrieve, in a fast, simple and understandable way, structural information from the source code, producing a set of test cases with calls to the different constructors and public methods available in the classes of the code. This chapter shows the design and implementation of this solution, as well as a brief case study to illustrate the viability of this proposal.

4.2 ASkeleTon: test harness generation from the AST

This section introduces ***ASkeleTon***, a tool for automatic generation of test harnesses for SUT written in C/C++ based on the AST. ASkeleTon, like other popular tools [24, 29], incorporates the Clang AST to obtain complete and easy-to-interpret information about the source code structure. The result of applying this tool is a test suite organised and identified by classes, with at least one generic call to each method and constructor in the code, allowing test data to be changed or further calls to be generated quickly and easily. ASkeleTon offers the following features:

- **Automatic generation of test harnesses.** Being the main purpose of the tool, it is able to generate a set of directories and test files ready to be compiled and executed with randomly generated default data.
- **Generation of log files.** While the AST is being analysed, the tool leaves a record of which elements are found along with their location in the source code. These elements, regardless of whether they are used in the tests or not, are useful for test engineers to have a snapshot of the code structure available at a glance, thereby facilitating the task of testing and subsequent maintenance.
- **Use of templates for testing frameworks.** ASkeleTon uses BOOST as a test framework by default, however, it is extensible to any other framework designed for testing C/C++ code. The use of templates detached from the source code makes the incorporation of other test frameworks an elementary task, as it will be seen later in this chapter.
- **Separation between test cases and test data.** The test cases contain references to the test data found in external files, allowing the parameterisation of the data in the test cases. Therefore, it is possible to work separately in the generation of the test data, helping to incorporate new state-of-the-art techniques without the need to modify the test harness or the code of ASkeleTon.
- **Fully modular development.** ASkeleTon has been designed so that each group of functionalities can be found in a specific module. Specifically, and apart from the main program, there is a module for generating code from templates and another for analysing and obtaining information from the AST. The purpose of this design is

to facilitate the maintenance and extension of tools, as well as the reuse of modules in other tools, in line with the changing needs of projects in the industry.

Figure 4.1 presents a schematic depiction of the general operation of ASkeleTon. The process begins with the specification of the requirements for the SUT to be met. Then, an AST is generated. Subsequently, Matchers are applied to the AST to extract information regarding the structure of the source code. The output of this process is then used to construct the test harness, which is composed of a multitude of software components. In the following section of this chapter, we will describe the various elements comprising the test harness in greater detail.

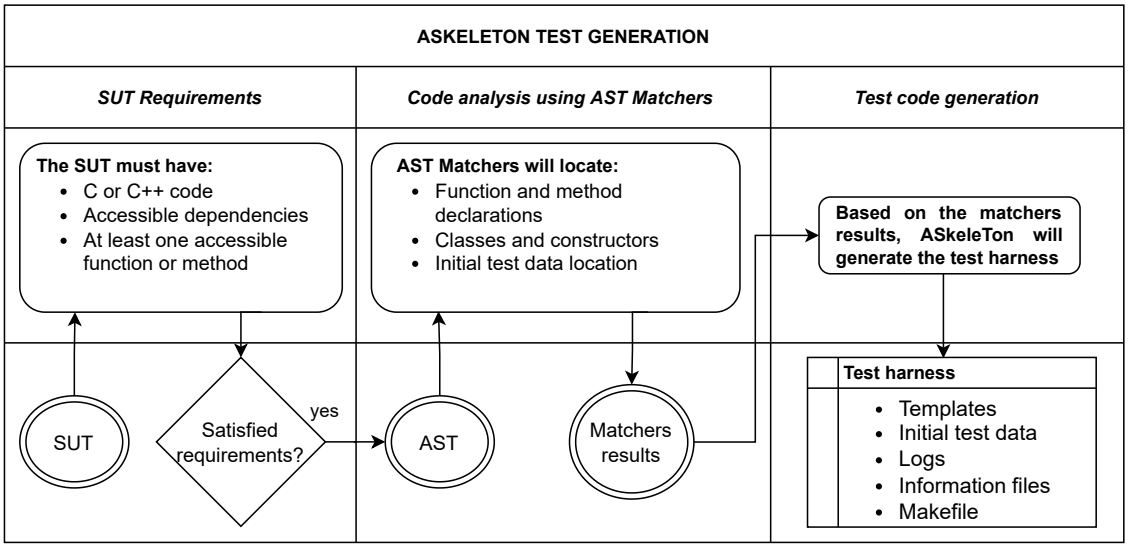

FIGURE 4.1: *ASkeleTon* general schema

ASkeleTon is one of the results derived from the industrial experience described in Chapter 3. Based on the needs observed in the development of the projects and the valuable feedback received, this tool aims to provide a complete test harness that serves as a basis for the accomplishment of the stages of software testing in any context. Its application to open source projects shows promising results, obtaining not only the test harness, but also valuable information for code maintenance and reuse.

4.3 Test harness design

This section illustrates the test harnesses that ASkeleTon is capable of generating. It does not rely on any specific testing framework, but includes BOOST by default. In

Section 4.3.1, we introduce BOOST as a testing framework. In Section 4.3.2, we provide an illustrative example of the test harnesses generated by ASkeleTon.

4.3.1 BOOST as default test framework

BOOST [81] is a widely-used open-source library for C++ programming, offering a range of algorithms and data structures, as well as compatibility with various platforms and operating systems. It has become a popular choice among C++ developers and has been employed in numerous projects, such as operating systems, web browsers, desktop, and mobile applications. BOOST provides a variety of components for improving code efficiency and productivity, including tools for handling strings, numbers, and complex data structures, as well as algorithms for data processing and code optimisation.

A key component of BOOST is the BOOST Testing Framework, which assists in the creation and execution of unit tests for C++ applications. The BOOST Testing Framework provides a range of tools and features for creating and running unit tests efficiently, including macros for value comparison and condition verification, as well as functions for creating tests that run in various contexts and environments. It also includes a command-line console for easy unit test execution and a test tracking tool for monitoring progress and results. In conclusion, the BOOST Testing Framework is an invaluable resource for ensuring the quality and reliability of C++ code.

Listing 4.18 is an example of using the BOOST testing framework to test a small function that returns the larger of two integers:

```
1  #include <boost/test/unit_test.hpp>
2
3  int max(int a, int b) {
4      return (a > b) ? a : b;
5  }
6
7  BOOST_AUTO_TEST_CASE(test_max)
8  {
9      BOOST_CHECK(max(1, 2) == 2);
10     BOOST_CHECK(max(2, 1) == 2);
11     BOOST_CHECK(max(2, 2) == 2);
12     BOOST_CHECK(max(-1, 2) == 2);
13     BOOST_CHECK(max(1, -2) == 1);
```

```
14 }
```

Listing 4.2: BOOST Testing Framework example

In this example, the `BOOST_CHECK` macro is used to verify that the *max* function is returning the larger of two integers as expected. The `BOOST_AUTO_TEST_CASE` macro is used to define a test case and all checks performed with `BOOST_CHECK` within that test case will be considered part of the same test case. If any of the checks fail, the test case will be considered failed.

4.3.2 Structure of ASkeleTon test harnesses

The modular design of ASkeleTon generates a range of files that together constitute the test harness. This section presents the most significant elements to illustrate the concrete output of ASkeleTon before examining each part in detail.

First, we will start with a function that will act as a SUT (see Listing 4.3). This is the same example used in Section 4.3.1 above.

```
1 int max(int a, int b) {
2     return (a > b) ? a : b;
3 }
```

Listing 4.3: Max value function

A single execution of ASkeleTon generates the log shown in Listing 4.4. As it is a small function with only one binary operator (the > operator) and one function, the log contains only two lines. The first line indicates the presence of the binary operator, and the second line indicates the presence of the function.

```
1 Found BinaryOperator at 2:13 — from function max
2 Found FunctionDecl at 1:1 — max in file Max
```

Listing 4.4: Log produced by ASkeleTon

Listing 4.5 shows the test file, also referred to as the Test Skeleton. It contains a test case with an assertion that serves as a unit test. The values for this assertion are

parameterised, meaning that they are not written directly in this file. Instead, they are specified in an external file with a *.cfg* extension (as shown in Listing 4.6). The test case reads these values from the *.cfg* file using auxiliary functions such as *Read_int*.

```
1  #include "Max_fixture.hpp"
2
3  BOOST_FIXTURE_TEST_CASE(Max_ReadParams, Fixture)
4  {
5      Date("Start");
6      BOOST_CHECK_EQUAL(max(Read_int("max.a"),Read_int("max.b")),Read_int("max.
       return_int"));
7  //{assert}
8
9      Date("End");
10 }
```

Listing 4.5: BOOST Test Skeleton

Listing 4.6 shows the aforementioned *.cfg* file. It includes the data that is used to populate the test cases. Particularly, it specifies the name of the function along with its parameters and the expected return value for the test to be successful. This file is in plain text, making it easy to edit either manually or with external data generation techniques. ASkeleTon generates initial test data input randomly, but the expected output must be filled in by the test engineer. In the current version of ASkeleTon, this value is automatically filled in to allow for immediate test compilation, but it is expected to be incorrect. It is the responsibility of the test engineer to provide the correct expected output value for each test case in order to ensure that the tests are valid and effective in finding defects in the software.

```
1  max:
2  {
3      a=-6;#int
4      b=8;#int
5      return_int=8;#int
6  };
```

Listing 4.6: Test data (.cfg file)

Besides these files, several auxiliary files are generated to facilitate the correct compilation and execution of the test harness. These include the Fixture file, which contains the

methods for reading data from the *.cfg* file and parametrising it in the test cases, the information file (SupportedTypes.txt), which shows the test engineer which data types are supported in the tests, and the Makefile, which allows for the compilation of the test harness with a single command. We discuss the role and organisation of all these files in detail in Section 4.4 below.

4.4 ASkeleTon: design and implementation

ASkeleTon has been designed to be user-friendly: starting from a SUT written in the C/C++ language, one gets a test harness to start working on. This harness is perfectly compilable and runnable right away. The main execution flow is shown in Figure 4.2.

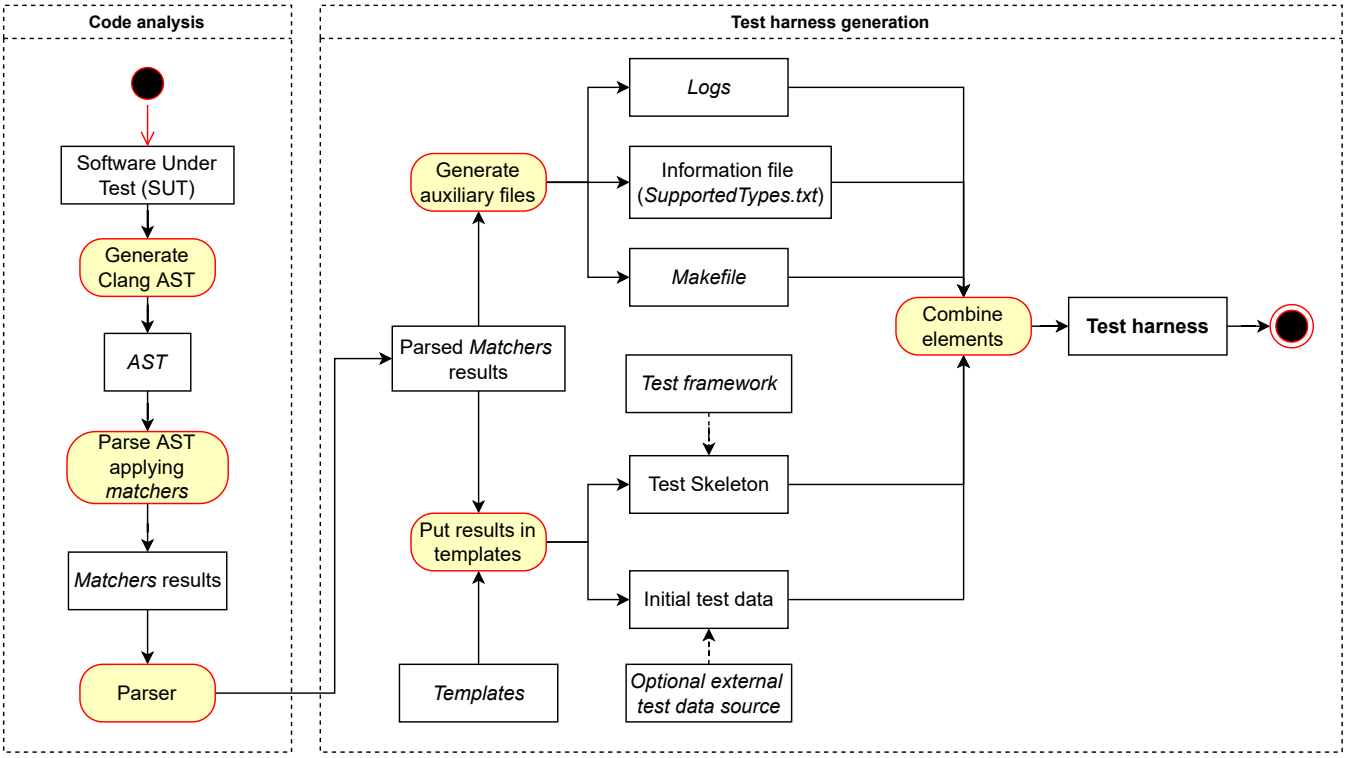

FIGURE 4.2: *ASkeleTon* workflow

In short, by looking at Figure 4.2, it can be seen how ASkeleTon produces the test harnesses. First, it obtains the AST from the Clang compiler. After that, a series of specific functions known as *Matchers* are applied to extract the necessary information to build the test cases. Finally, using the results of the *Matchers* and external templates, it will generate a set of files that make up the test harness. This includes code for the test

cases and some initial test data. This process is performed for each of the SUT classes, as well as for those global functions that do not depend on any class.

The modular structure of ASkeleTon allows each step to be performed by a different module in the application. One of the main purposes of this design is to facilitate the maintenance, reuse and extension of the source code of both the test harness and ASkeleTon itself. In the following, each of the steps, as well as small parts of the implementation and the results, are described in order to better understand how they work.

4.4.1 SUT Requirements

ASkeleTon receives as input the path to the file containing the SUT, which must meet a number of requirements:

1. It must be C or C++ code, so file extensions should be among the following for ASkeleTon to work as expected: `{.c, .cpp, .h, .hpp}`.

2. It must contain at least one accessible function or method that receives some parameter and returns a specific value. ASkeleTon bases its tests on assertion checking, so it will not automatically generate tests for *void* functions that do not return any value.

3. If the code is based on orient-objected programming, its classes must have at least one public method. Assertions that test private parts of the class will not be generated automatically, as the tests will not have access to them.

4. If the SUT has any specific compilation needs (dependencies, specific parameters, optimisation levels, etc.) it is the responsibility of the test engineer to incorporate the corresponding makefile into the final result.

Although testing with user-defined types is allowed, it is advisable to pay attention and review the way they are incorporated in the tests, as they will be used with the default values of the attributes of the class defining the object.

Once ASkeleTon receives —as a parameter in its main class— the path to the file to be tested (e.g. `/route/to/file.cpp`), and, as long as this file contains C or C++ code,

the log where the whole process is recorded will be created. At this point, the files are opened and the start time is saved in the log, continuing with the next step, which would be to obtain the AST.

4.4.2 Generation of the AST

Thanks to the Clang compiler, obtaining the AST can be done quickly and smoothly. The main class of ASkeleTon contains two lines that will give the program immediate access to the AST.

```
1 ClangTool Tool(OptionsParser.getCompilations(),
2                OptionsParser.getSourcePathList());
```

Listing 4.7: Getting the AST in the code

This particular AST has a syntax similar to that of C++, so a test engineer should be able to read it without particular difficulty. However, for automatic processing, it is preferable to make use of *Matchers*. These are predicates executing queries on the AST node. One of the main features is that *Matchers* can be nested in the form of a tree, adding constraints to the predicates, thus allowing to find concrete situations. In the following, there is an example of how *Matchers* work, as well as those used internally by ASkeleTon.

4.4.3 Code analysis: AST Matchers

The way *Matchers* work is relatively simple. For example, to obtain the methods declared within the classes in C++, it would be enough with the Listing 4.8 statement.

```
1 DeclarationMatcher MD1 = cxxMethodDecl( );
```

Listing 4.8: Obtaining method declarations

It is possible, as noted above, to nest the *Matchers* (they can be seen as constraints), so that we get more specific elements. The example above can be modified, for example, so that it only shows methods that have input parameters, as shown in Listing 4.9.

```
1 DeclarationMatcher MD1 = cxxMethodDecl(
2                                 hasAnyParameter(...));
```

LISTING 4.9: Obtaining method declarations with parameters

The nesting of these restrictions has no limit and allows describing code situations in a generic way (functions with parameters, classes with constructors, etc.). This facilitates the creation of procedures to automate the analysis of the AST and thus obtain key information for test case generation.

ASkeleTon implements six different *Matchers*, four of them to obtain structural elements and other two that will be useful when generating test data. The following section describes each of them. Please note that we apply *Matchers* to retrieve the elements of the code required for the process of test generation; later, however, these elements are further processed for their breakdown (obtaining lists of parameters, names, etc.).

Function and method declarations

The first two *Matchers* are used to obtain the functions and methods from the code. Looking at Listings 4.10 and 4.11, it can be seen that not every element is captured, but rather a series of restrictions is applied to retrieve only the most relevant data for the generation of test cases.

Listing 4.10 will first capture the AST nodes of type *functionDecl*, i.e. function declarations, excluding methods of classes. Within the call to *functionDecl* the following restrictions are included:

- `unless( isImplicit() )` - It excludes from the result implicit functions, those that the compiler assumes are defined or declared elsewhere. We have observed that the compiler often includes functions in the files that are not defined by the programmer. We are only interested in functions explicitly declared in the code, which are obtained by applying this restriction.
  - **Breakdown of the instruction**
    - `unless( )`: negation operator.

    - `isImplicit()`: returns *true* if the declaration has been implicitly generated. Conversely, if it has been explicitly written in the code, it returns *false*.

- `unless( returns( voidType() ) )` - Exclude from the result those functions that do not return anything, of type *void*.

  - **Breakdown of the instruction**

    - `unless( )`: negation operator.
    - `returns( )`: returns *true* if the function or method returns a value of the type specified as a parameter. Otherwise, it returns *false*.
    - `voidType( )`: returns *true* if the function or method is void (does not return any value). Otherwise, it returns *false*.

- `isExpansionInMainFile()`: returns *true* if the function was expanded within the main-file. This means that only the code of the file being analysed is considered, excluding all results coming from the headers. This avoids generating duplicate tests or testing other modules that need a separate analysis. Otherwise, it returns *false*.

```
1 DeclarationMatcher FD1 =
2     functionDecl(
3         unless( isImplicit ()),
4         unless(returns(voidType())),
5         isExpansionInMainFile()
6     ).bind("FD1");
```

Listing 4.10: AST Matcher for function declarations

Listing 4.11 is similar to Listing 4.10, only changing the *functionDecl* method to *cxxMethodDecl*. In this way, the AST nodes corresponding to method declarations defined within classes will be captured. Although both *Matchers* share the same constraints, this one includes one more constraint due to the inherent characteristics of the methods.

- `isPublic( )`: returns *true* if the visibility of the element is set to *public*. Otherwise, it returns false.

```
DeclarationMatcher MD1 =
    cxxMethodDecl(
        unless( isImplicit ()),
        isPublic(),
        unless(returns(voidType())),
        isExpansionInMainFile()
    ).bind("MD1");
```

LISTING 4.11: AST Matcher for method declarations

With just these two *Matchers*, we have captured all the functions and methods available for use in software testing. Next, two more *Matchers* are shown for obtaining typical language constructs of C++ programs, not included in the ones presented so far.

Classes and constructors

Listing 4.12 and 4.13 show *Matchers* dedicated to obtaining structural information. Specifically, they seek to capture those structures of type `struct`, `interface`, `class`, `union` and `enum`, as well as their constructors where appropriate. The constructors will be treated as functions, although they will be given their specific functionality in the test cases later.

Listing 4.12 captures all nodes of the AST of type *cxxRecordDecl*. This refers to all code elements containing the keywords `struct`, `interface`, `class`, `union` and `enum`. In an early version of ASkeleTon, five different *Matchers* were included to capture each of the elements separately. However, the subsequent processing of the captured nodes makes it possible to differentiate them, so we have finally opted for this single and more general *Matcher* with two restrictions.

```
DeclarationMatcher CT1 =
    cxxRecordDecl(
        unless( isImplicit ()),
        isExpansionInMainFile()
    ).bind("CT1");
```

LISTING 4.12: AST Matcher for C++ struct/union/class

Listing 4.13 is specifically designed to obtain the constructors of those elements, if necessary. Certainly, with the elements obtained from the *Matcher* presented in Listing 4.12, we could have obtained the constructors. However, this would require a more intensive post-processing of the AST node in which we had to filter out constructs without constructors, as there are elements that cannot have a constructor, do not implement it or it is not accessible. With this *Matcher*, all constructors that meet certain conditions are captured quickly and easily to be used in software testing.

```
1 DeclarationMatcher CC1 =
2     cxxConstructorDecl(
3         unless(isImplicit()),
4         isExpansionInMainFile()
5     ).bind("CC1");
```

LISTING 4.13: AST Matcher for constructors

The tests performed, as well as the external feedback received, show that with just these four *Matchers* we can get enough structural information to generate a good initial test suite. An illustrative example is shown below.

Example

Listing 4.14 shows an example class on which the *Matchers* described so far are applied. Each of them will match the following AST nodes:

- Listing 4.10 will capture the function `int externalFunction()`.
- Listing 4.11 will capture the method `int pow()`.
- Listing 4.12 will capture the class `AST`.
- Listing 4.13 will capture the constructor `AST(int value)`.

Therefore, the method `void showMessage()` will not be captured as it does not return anything, and neither the method `int getNumber()` for being private.

```
1  class AST {
2  public:
3      AST(int value) : value(value) {}
4      int pow() { return value*value; }
5      void showMessage() {
6          std::cout << "Hello there!";
7      }
8  private:
9      int getNumber() { return value; }
10     int value;
11 }
12
13 int externalFunction(int a) { return a*a; }
```

LISTING 4.14: SUT example for *Matchers*

Potential test data location

We have previously seen that the generation of test data in ASkeleTon is separated from the structural information of the tests. By default, this data is generated randomly. In line with industrial experience, where it is common for software domain experts to input data manually or use automated techniques, we chose to delegate the input of more complex test data to an external source. During the development of ASkeleTon, however, two *Matchers* have been created to locate conditional statements within the code and the comparisons appearing in them. This allows identifying which variables may be relevant when generating random values.

These patterns help detecting meaningful values that are worth considering in the testing process. The results of these *Matchers* are directly shown in the logs. This allows the test engineer to detect them at a glance and thus use the information to, for example, prioritise these values when designing test data.

Listing 4.15 shows a *Matcher* capable of finding all relational operators. This time, the constraints are a bit more particular, so they are defined below:

- `functionDecl( )`: matches all function declarations.

- `forEachDescendant( )`: it explores the descendants of a specific node. Thus, the constraints apply not only to the node being analysed, but to each of its descendant nodes.
- `binaryOperator( )`: returns *true* if the node represents a binary operator, *false* otherwise. At the same time, this operator enables the use of internal constraints, so that only the binary operators meeting certain conditions return *true*.
- `anyOf( )`: returns *true* if any of the given statements returns *true*, otherwise returns *false*. It is equivalent to the logical operator `OR`.
- `hasOperatorName("opName")`: returns *true* if the node represents an operator of the specified type. It returns *false* otherwise.
- `unless(isImplicit())`: only matches operations explicitly defined by the programmer.

```
DeclarationMatcher DG1 =
    functionDecl(
        forEachDescendant(
            binaryOperator(
                anyOf(
                    hasOperatorName("=="),
                    hasOperatorName("!="),
                    hasOperatorName(">"),
                    hasOperatorName(">="),
                    hasOperatorName("<"),
                    hasOperatorName("<=")
                )
            ).bind("DG1")
        ),
        unless( isImplicit ())
    ).bind("DG1b");
```

Listing 4.15: AST Matcher for conditionals

The *Matcher* shown in Listing 4.15 consider all the conditional constructs provided by the language, except for the *switch* type statements. For this reason, the Listing 4.16 *Matcher* has been designed to capture this type of elements.

```
DeclarationMatcher DG2 =
```

```
    functionDecl(
        forEachDescendant(
            switchStmt().bind("DG2")
        ),
        unless( isImplicit ())
    ).bind("DG2b");
```

LISTING 4.16: AST Matcher for switch statements

Please note that Listings 4.15 and 4.16 are designed for nodes of type *functionDecl*. The final ASkeleTon implementation includes another two analogous *Matchers* for class methods, only changing *functionDecl* to *cxxMethodDecl*. As their operation is exactly the same as described in Listings 4.15 and 4.16, a copy is not included in this document.

These last two *Matchers* have an informative purpose, so previous constraints such as the visibility of the analysed methods are removed here. This is because the test engineer might want to know, for example, when the attributes of a class are modified within a private method. Whenever the *Matcher* finds a result, it will be recorded in the logs.

At this point, ASkeleTon has enough information from the SUT to generate test cases. The output of the *Matcher* consists of ready-to-use structural information and can therefore be used to build test cases in any test framework.

4.4.4 Test code generation

ASkeleTon implements a template-driven modular design for the generation of test harnesses. There are templates in `.tpl` format that shape the test files, so that ASkeleTon can replace a number of parameters with the information extracted from the AST. There are modules, internally referred to as *generators*, which are responsible for processing and inserting such information into the templates. Furthermore, a set of default files assists in test compilation and provide useful information to test engineers.

Templates

ASkeleTon incorporates a .tpl template file for each test harness file to be generated beyond those generated by default (described later in this section). In its default state,

ASkeleTon incorporates a template for the generation of test cases with two main files: a C++ file containing test cases in the BOOST framework and a *Fixture* file, which contains all the methods necessary for the initialisation and parameterisation of the test case. The BOOST Test library is a testing framework designed to automate the most basic tasks found in most test cases (such as assertions, variable initialisation, etc.) [81]. This easy-to-use library, which is well established in the industry, provides all the necessary methods to perform complex validation tasks. Listing 4.17 shows the template for the BOOST tests.

```
#include "{className}_fixture.hpp"

BOOST_FIXTURE_TEST_CASE({className}_ReadParams, Fixture)
{
    Date("Start");
    {pointerInitToken}
        //{assert}
    {pointerDestroyToken}
    Date("End");
}
```

LISTING 4.17: Boost template

The *BOOST* template is designed to be compact, simple and easy to understand. All the complex methods for initialisation and data reading are contained in the *Fixture* file (described below). It is convenient to differentiate between fixed and configurable elements in the template, as configurable elements will define the specific behaviour of the tests. It is possible to quickly identify them by the braces (`{elementName}`):

- **{className}:** the name of the class under test. There shall be at least one file for each class. If the functions under test do not belong to any class, this parameter shall be replaced by the name of the file to which they belong.
- **{pointerInitToken}:** when test cases include variables that require memory reservations (mainly in C programs), this element will be replaced to include them.
- **{assert}:** the assertions to be run in the test cases. ASkeleTon shall replace this element by as many assertions as necessary to complete the test execution.

- **{pointerDestroyToken}:** if the test case has required memory reservation, this element shall be replaced by statements with the actions responsible for releasing the reserved memory.

Note that the **{assert}** element is commented out, so at first glance it may seem to have no effect on the code. However, this is merely an aid during the execution of ASkeleTon. As the AST is traversed, as soon as any of the *Matchers* gets enough information to generate an assert, ASkeleTon incorporates it into the result. To avoid constant comparisons like: "is this the last possible assert in the test case?", we chose to always leave a commented assert block. Therefore, in the test harness, all the assertions will be added together up to a last line, which will be the same as the one we see in the template.

Within the assertions, there are three types depending on the case:

- `BOOST_CHECK(new Object( ... ));` this assertion is generated once for each constructor defined in the class, if any. It checks that the object has been created correctly.
- `BOOST_CHECK_EQUAL(call_to_function, expected_output);` this assertion is generated at least once for each function or method available in the code. It checks if a call to a function with specific data generates the expected output.
- `BOOST_CHECK(call_to_function == expected_output);` `BOOST_CHECK_EQUAL` does not support somewhat more complex data types such as C++ Standard Template Library (STL) containers, user defined types, etc. Therefore, methods using these types are identified and their assertion will be of type `BOOST_CHECK`, which allows comparisons to be made by using logical operations.

As for the fixed elements, they are elements of the *BOOST* framework. Line 1 in Listing 4.17 includes the *Fixture* file, whose name will always match the name of the class or file to be tested. Line 3 consists of the creation of a *BOOST* test case that uses *Fixture*, and it is called as shown in Listing 4.18. Within the test case, besides the configurable elements described above, there are two lines of type `Date("Start/End")`. These calls allow the programmer to know the start and end time of each test case.

```
1 BOOST_FIXTURE_TEST_CASE(test_case_name, fixture_name)
```

Listing 4.18: Boost Test Case example

On the other hand, the template of the *Fixture* file in the initial version of ASkeleTon contains 522 lines of code, as it includes everything necessary for the initialisation, reading and parameterisation of the data during the execution of the tests. This code is mainly composed of *getters* and *setters* specific to each data type, so we will focus only on the configurable elements.

- **{className}:** the name of the class under test (same as in *BOOST* template).
- **{classNameTest}:** the name of the test case, which will consist of the name of the class followed by the "test" keyword.
- **{includes}:** if the class under test needs to include a specific library, ASkeleTon will include it in the Fixture. This parameter has been included following industrial feedback, as there are projects where the access to libraries changes between the test and production environment. Normally, the whole code is accessible, so by including the class under test in the Fixture, it will not be necessary to add additional libraries.
- **{namespaces}:** if the class under test needs to include a specific namespace, ASkeleTon will include it in the Fixture.
- **{readObject}:** if the class under test uses an object of a specific class as an attribute, and that class is accessible, a method shall be added to the Fixture to facilitate reading and writing it during testing.

Once the files that make use of templates are created, it is time to generate the rest of the files. ASkeleTon generates these files from scratch, being two of them based on the AST: the *.cfg* file with the test data and the logs.

Generators: test data and logs

The test data is parameterised in a *.cfg* file. This file does not need to use templates, as ASkeleTon will be able to generate them dynamically while the assertions are being

generated. This file contains the data to be used by the test assertions, which have three possible forms, as we can see in Listings 4.19, 4.20 and 4.21.

Listing 4.19 represents a call to a default constructor. Line 1 is the name of the class which the object to be created belongs to. Then, nothing is included between the braces, representing that there are no parameters.

```
1 className: { };
```

LISTING 4.19: Test data for empty constructors

Listing 4.20 represents a call to a constructor with parameters. Line 1, again, represents the name of the class on which an object is to be created. After that, there is a list of parameters. It is convenient to analyse each line of Listing 4.20 to better understand the syntax of this file.

- `className_1`: the name of the class under test. The *_NUMBER* termination is added by ASkeleTon automatically in case there is more than one constructor to avoid naming conflicts when parameterising data.
- `first_param=value;#type`: the name, value and type of each input parameter.
  - **Breakdown of the instruction**
    - `first_param`: the name of the input parameter.
    - `value`: the value of the input parameter.
    - `#type`: a comment about the input data type. ASkeleTon uses this information to know which *Fixture* method should be used to parameterise the data. As the *.cfg* file is a text file, this will be useful for parsing the values.

```
1 className_1:
2 {
3     first_param=value;#type
4     second_param=value;#type
5 };
```

LISTING 4.20: Test data for constructors

The last of the elements that can be found in a *.cfg* file are the method and function calls, as shown in Listing 4.21. There are some notable differences with respect to the previous case:

- `functionName`: The name of the method or function under test. In the case of methods, it is not necessary to specify the class, as ASkeleTon generates a file for each class under test, so there is no conflict. As with constructors, ASkeleTon can add the *_NUMBER* termination to provide different data for the assertions.
- `first_param=value;#type`: the same as the previous case.
- `return_type=value;#type`: at the end of each method or function call, there is a parameter like the one in the example, which represents the expected output value.

```
functionName:
{
    first_param=value;#type
    second_param=value;#type
    return_type=value;#type
};
```

LISTING 4.21: Test data for methods and functions

ASkeleTon is prepared to generate assertions using STL containers as well as user-defined types. Listing 4.22 shows an extract of a *.cfg* file, which includes values for a vector and a map parameter in the call to the functions *vectorMethod* and *mapMethod*, respectively.

```
vectorMethod:
{
    v_param={-73,98,13,79,-54};#vector<int>
    return_vector<int>={-57,78,39,52,-5};#vector<int>
};

mapMethod:
{
    m_param={(68,-88),(18,66),(23,-49),(51,-93),(67,3)};#map<int, int>
    return_map<int, int>={(37,24),(28,-20),(-82,-15),(-4,-42),(-16,-6)};#map<int, int>
};
```

LISTING 4.22: CFG example using the STL

Finally, regarding the AST information used by ASkeleTon, a log is generated with all the usable elements during software testing. Listing 4.23 shows an extract with a log that is saved, as well as displayed to the user during the execution, for a set of test files.

```
1 Found CXXRecordDecl (struct−customtype) at 15:1 − customType in file ASK.cpp
2 ...
3 Found FunctionDecl at 108:1 − conditionalMethod in file ASK.cpp
4 Found BinaryOperator at 118:6 − from function conditionalMethod
5 ...
6 Found CXXConstructorDecl at 25:3 − FirstClass from class FirstClass
7 Found CxxMethodDecl at 59:5 − iObtainAString from class FirstClass
8 ...
9 Found CxxMethodDecl at 92:3 − testPointer1 from class SecondClass
```

LISTING 4.23: ASkeleTon log example

- Line 1 informs the user that a data type defined in the class has been found. The name of the type is *customType* and is found in the *ASK.cpp* file.
- Line 3 informs the user that a function has been found on line 108 of the *ASK.cpp* file. The name of the function is *conditionalMethod*. Line 4 also informs the user that a conditional exists within that function.
- Line 6 informs the user that a constructor defined for the *FirstClass* class has been found. In addition, line 7 informs the user that a method called *iObtainAString* has been found in that case.
- Finally, line 9 informs of the existence of a method called *testPointer1* belonging to the *SecondClass* class.

This is just an extract of any log file containing the information on the AST nodes scanned during the generation of the test harnesses. Although the size of these files will logically depend on the size of the code, they are relatively easy to read.

Default files

Along with all the elements described up to this point, two auxiliary files are always generated to facilitate the initial tasks of the test engineer: an information file and a default makefile. These files are known as *default files*.

- **Information file (SupportedTypes.txt)** This is a file containing information on the data types supported by the automatically generated test harnesses. The purpose of this file is to inform the test engineer which data may or may not be used without modifying the Fixture file. By default, ASkeleTon supports all elementary data types as well as STL-derived types. However, it also allows the use of user-defined data types, which are detected and added automatically (both to the test harness and to this file). To do this, ASkeleTon will use public constructors if they exist to construct these data types, creating the corresponding 'Read_<type>' function in the *Fixture* file. If there are no public constructors, or if the data type needs a specific procedure for initialisation, this is the responsibility of the test engineers. At this point, the test engineer can check at a glance, for example, whether more complex data types can be used in these test cases. Listing 4.24 shows an extract of a SupportedTypes.txt file, where you can see a list of types supported by the test cases.

```
1 ...
2 float
3 double
4 long_double
5 std::string
6 ...
```

LISTING 4.24: SupportedTypes.txt example

- **Makefile** A default *makefile* is included to compile the test harness along with the SUT. As the test harness generation process involves the execution of the test harness as part of the process, it is necessary to have access to the SUT code and, if necessary, to the SUT compilation instructions. The purpose of this file is to provide a simple method for the compilation of the more standard test harnesses, which do not have excessive complexity or dependencies.

  The inclusion of these files is entirely auxiliary. However, their inclusion facilitates the task of test case execution and maintenance. In more complex SUTs, it will be the responsibility of the test engineers to incorporate a compatible *makefile* (usually the same as the SUT with minor modifications to include the compilation of the test harness).

4.5 Resulting test harness

In Section 4.4 we have seen the complete workflow of ASkeleTon. From the SUT requirements, through the generation and analysis of the AST, to the generation of the code that forms the test harness. The final result, as shown in Figure 4.3, is a set of directories and files that form an initial test harness ready for execution.

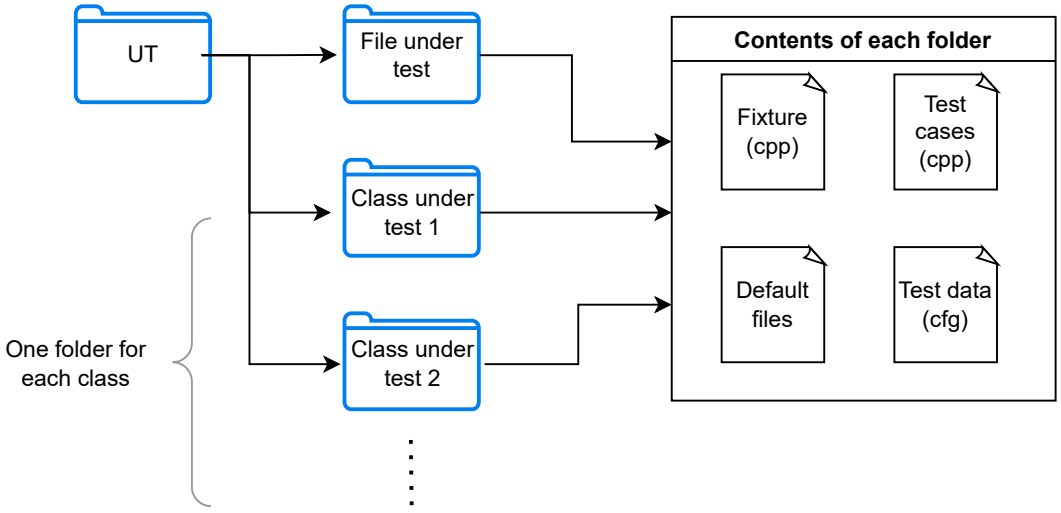

FIGURE 4.3: Structure of the files and directories generated by ASkeleTon

The test harness is stored in a directory called UT, whose content is made up of a series of directories. Each of these directories contains the files generated to test the code of a file or a class. Now, a closer look at Figure 4.3 reveals two types of directories:

- **File under test:** holds all files containing tests related to functions in the code that are not associated with any class. There shall be one directory for each file containing such functions and it shall have the same name as that file. For example, if there is code under test in a file named *test.cpp*, the directory will be named *test*.
- **Class under test:** holds all files containing tests related to a class under test. There will be one directory for each class and it will have the same name as that class. For example, if a class under test called *Thesis* exists in a file called *test.cpp*, the directory will be called *Thesis*.

It is possible to find name conflicts between directories under certain conditions. Mainly when a file containing non-class functions shares a name with one of the SUT classes. In this case, ASkeleTon resolves this by appending a number to the directory name (e.g.

"FileUnderTest_1"). Please note that the name of the directories is purely informative, so it will not alter the functioning of the test harness.

Finally, each of the directories contains the files for the execution of the tests. These are described in Section 4.4, i.e. the test skeleton, some initial data, and the default files to assist in the test case execution process.

The program will output a message for each failed test. This message informs the test engineer that an error was encountered in a function call, where one output was expected, but a different one was obtained. The test data is retrieved in real time, as evidenced by the calls to *Read_int*, *Read_float*, etc. in Listing 4.26. This allows the data contained in the *.cfg* file to be configurable, as seen in Listing 4.25, and re-running the tests with different data without the need to re-compile the test harness. Those tests that pass successfully will not provide any output.

```
sum:
{
        a=2.0;#double
        b=1.5;#double
        return_double=4.0;#double
};
```

LISTING 4.25: Test data file example

To run the test harness, all that is needed is to enter the directory and run the command `make`. This will generate an executable named `test`, which will perform all the tests. Running this executable will lead to an output similar to Listing 4.26.

```
Running 1 test case...
Calculator_test.cpp(10): error: in "Calculator_ReadParams": check Calculator_test.sum(Read_int(
    "sum.a"), Read_int("sum.b")) == Read_float("sum.return_float") has failed [3.5 != 4.0]
```

LISTING 4.26: Test harness execution example

4.6 Case Study: use of ASkeleTon

In this section, we run ASkeleTon on an ordinary C++ class. Specifically, the Calculator class (see Listing 4.27) is used as a base, which models a simple calculator with four operations: addition, subtraction, product and division.

```
1  #include <iostream>
2  class Calculator {
3  public:
4      Calculator() {}
5      Calculator(double lastRes) : lastRes(lastRes) {}
6      double sum(double a, double b) {
7          double res = a+b;
8          setMem(res);
9          return res;
10     }
11     double sub(double a, double b) { return sum(a, -b); }
12     double prod(double a, double b) {
13         double res = a*b;
14         setMem(res);
15         return res;
16     }
17     double div(double a, double b) {
18         if (b == 0) {
19           std::cout << "DIVIDED BY ZERO, RETURNING 0" << std::endl;
20           setMem(0);
21           return 0;
22         } else {
23           setMem(a/b);
24           return a/b;
25         }
26     }
27 private:
28   double lastRes;
29   void setMem(double res) { lastRes = res; }
30 };
```

Listing 4.27: Class under test: Calculator.cpp

This class also incorporates a memory value that stores the last result obtained. This value can be initialised when the object is built, as can be seen in the constructor on

line 5, which complements the empty constructor. The use of this class is intended for illustrative purposes, to help the reader better understand the operation and results of ASkeleTon.

Listing 4.28 shows the result of the execution, which is also stored in the corresponding log. We see how ASkeleTon has detected two constructors (`CXXConstructorDecl`), four public methods (`CxxMethodDecl`) as well as a logical operation (`BinaryOperator`) inside one of the methods. These elements are displayed by default, and they are precisely the ones used for the creation of the test harnesses. Going back to Listing 4.27, we can see that the class has indeed all the elements shown during the execution of ASkeleTon.

```
1 $   ./askeleton Calculator.cpp --
2 Found CXXConstructorDecl at 6:5 - Calculator from class Calculator
3 Found CXXConstructorDecl at 7:5 - Calculator from class Calculator
4 Found CxxMethodDecl at 9:2 - sum from class Calculator
5 Found CxxMethodDecl at 15:2 - sub from class Calculator
6 Found CxxMethodDecl at 17:2 - prod from class Calculator
7 Found CxxMethodDecl at 23:2 - div from class Calculator
8 Found BinaryOperator at 24:10 -  from function div
```

LISTING 4.28: Running ASkeleTon

A quick look at the *UT/Calculator* directory generated by ASkeleTon shows a series of files corresponding to each of the elements of the test harness.

- **Test cases:** *Calculator_test.cpp*, which includes the BOOST test files by default.
- **Test fixture:** *Calculator_fixture.hpp*, which is linked to the test cases and includes all the initialisation and auxiliary methods.
- **Test data:** *Calculator.cfg*, which includes all data used by the test cases.
- **Default files:** *makefile* and *log.txt*, which allow the compilation of the test harness and store the log generated by ASkeleTon, respectively.

```
1 $   ls
2 Calculator.cfg  Calculator_fixture.hpp  Calculator_test.cpp  makefile log.txt  SupportedTypes.txt
```

LISTING 4.29: Test harness

After compiling the test harness, it is ready to run. As the data generated by default is random, a small adjustment has been made to the *.cfg* file to include data which make the tests pass. As we can see in Listing 4.30, the test harness finishes its execution successfully. Please note that, although the output says *1 test case...*, it actually contains all the necessary test cases represented by assertions, each acting as an independent unit test.

```
$    ./test
Running 1 test case...

*** No errors detected
```

Listing 4.30: Test harness

Finally, we show what happens if a test, for whatever reason, does not pass successfully. Listing 4.31 shows a small modification to the data of a test, where a wrong expected result is intentionally included.

```
sum:
{
        a=2.0;#double
        b=1.5;#double
        return_double=4.0;#double
};
```

Listing 4.31: Wrong .cfg file

Without the need to re-compile the test, it is run again and shows a failed test (see Listing 4.32), with all the necessary information for the test engineer to fix the corresponding code piece. Ultimately, ASkeleTon has been able to generate a basic test harness, ready to incorporate any test data.

```
1 $    ./test
2 Running 1 test case...
3 Calculator_test.cpp(10): error: in "Calculator_ReadParams": check Calculator_test.sum(
      Read_double("sum.a"),Read_double("sum.b")) == Read_double("sum.return_double") has
      failed [3.5 != 4]
4
5 *** 1 failure is detected in the test module "Calculator_TEST"
```

LISTING 4.32: Failed execution

4.7 Chapter conclusions

This chapter shows ASkeleTon, a tool for the generation of test harnesses for programs written in the C/C++ language. ASkeleTon has been developed to be modular, so that its functioning, its maintenance and extension is a simple task. Moreover, it is not tied to any specific test framework (although BOOST is included by default). Therefore, the creation of a module for the generation of test harnesses in other frameworks does not require the modification of the main code of the tool.

The development of ASkeleTon has benefited from the industrial experience described in Chapter 3. Therefore, the structure of the code, the generated test harnesses and the analysis of the AST are strongly inspired by the continuous feedback received. The results, although also supervised and approved by this entity, have been validated against some illustrative and open source projects.

Considering the type of supervision ASkeleTon has had during its development, industrial validation is a logical and natural step. In a software development process, it is common to validate the process in intermediate phases with somewhat more illustrative pieces of code (such as the one shown in Section 4.6). Once the tool has reached a certain maturity, it is possible to start testing it with industry-wide projects, as well as distributing early versions to test engineers for them to use and provide valuable feedback. However, this type of validation has faced several obstacles.

The industrial projects studied have certain characteristics: they are large, often interact with critical systems and are exposed to a strict confidentiality contract. The size of the project, at first, is not a major issue, as ASkeleTon has shown to scale well when applied to large-sized programs. However, this feature adds to the interaction with critical systems,

so projects are often linked to external libraries that are either not accessible or need an active physical element (such as a radar, a scanner, etc.). We have worked closely to extract results from such projects, both ongoing and legacy, in order to improve ASkeleTon and demonstrate its feasibility in the best possible way. However, the strict confidentiality contract has led to severe limitations. Normally, access to source code for research purposes requires authorisation from senior management, which can take months to be granted. Until such authorisation is received, one does not know if the code is complete or viable for use (i.e., if it has too many dependencies on external libraries or other programming languages). This period of time to find a case study that produces interesting results for the research community turned out to be difficult to reconcile with the limited time available for conducting a PhD thesis, especially with the halt and the new restrictions imposed by the COVID-19 pandemic. In this respect, moreover, the strict confidentiality contract expressly prohibits the dissemination of results linked to company projects. This is the main reason why, at the time of writing, there are no publications describing ASkeleTon in detail or showing fragments of the industrial projects examined. There are, on the other hand, publications that describe the industrial collaboration without going into technical details [91, 92].

Up to this point in development, almost all of the feedback received had to do with the structure of the test harnesses. Although the test data is generated randomly, this is not a problem for test engineers working in this industrial context, where they often need tests to include specifically calculated data. Thanks to the modular development of ASkeleTon, as well as the separation of test data from the test structure, it is possible to research and develop the data generation process in a different research line. For this reason, and without leaving industrial validation behind, a new line of research arises where the most advanced state-of-the-art techniques are combined to obtain test data whose objective is the discovery of real faults in all kind of projects.

In conclusion, the use of ASkeleTon makes it possible to obtain a test harness ready for execution. However, the steps involved in industrial validation are not compatible in time with the period of development of the PhD thesis. Without leaving behind this validation, which is still ongoing and will be continued as future work, a new method for test data generation has been developed. This new method will be described in detail in the following chapter. The modular design of ASkeleTon will allow it to be easily integrated. Chapter 5, which follows, shows all the research work carried out for the

development of this new technique, as well as the result of its application to a real set of utilities.

Chapter 5

Combining MT and DSE for test data generation

"Science makes people reach selflessly for truth and objectivity; it teaches people to accept reality, with wonder and admiration, not to mention the deep awe and joy that the natural order of things brings to the true scientist."

Lise Meitner

The generation of test data is an important and challenging task, as it usually requires knowledge of the problem domain. One of the techniques that stands out in this respect is Dynamic Symbolic Execution (DSE) which, as seen in Chapter 2, produces good results in terms of structural coverage criteria. In view of the industrial situation in Chapter 3 and the development of ASkeleTon in Chapter 4, there is a need to implement a test data generation technique capable of detecting potential faults.

This chapter presents a comprehensive study on the ability of DSE to kill mutants and thus detect possible real bugs in the software. To do so, using a set of utilities, we generate test cases and mutants via DSE and MT tools, respectively. All test cases are then run on the mutants to classify them as surviving or killed. First results indicate that there are still several surviving mutants, so we propose combining both MT and DSE in a new family of techniques known as *MISE*. In order to test the potential usefulness of *MISE*, we perform a first basic implementation, which we call naive *MISE*. After performing similar steps to the first experiment in this study, naive *MISE* applies DSE to generate

new test cases for each surviving mutant. *Naive MISE* offers promising results, as the incorporation of MT elements during the generation of test data with DSE makes the mutation coverage increase significantly. Motivated by these results, we propose different potential implementations to implement *MISE* in a more effective and efficient way than *naive MISE* does.

5.1 Motivation

DSE is a well-established technique for the automatic generation of test cases, particularly for achieving structural coverage criteria such as line or branch coverage. However, DSE has several limitations that must be taken into account, including unsolvable constraints and path explosion, which can require significant computational effort to apply to projects of a certain complexity. Despite its limitations, DSE is still a valuable technique for testing, and combining it with other techniques such as MT can help overcome its limitations and partially solve other problems such as the detection of equivalent mutants or even killing mutants that remain alive after numerous test phases. Overall, the implementation of DSE and other automation techniques can bring significant benefits to software testing, including improved efficiency and reduced manual workload.

The incorporation of DSE as a test data generator in ASkeleTon is a promising approach, given the good results reported in the literature for its use in various domains. However, traditional coverage criteria may not be sufficient for detecting real bugs in programs. Therefore, it is necessary to evaluate the ability of DSE to generate test cases that can effectively detect these real bugs. To do so, we can use MT as a complementary technique. MT is a powerful tool for assessing the quality of test cases because it allows us to measure the mutation score, which indicates the percentage of mutants killed by the tests. This metric provides a clear indication of the effectiveness of the test suite in detecting defects and helps to identify areas where the test coverage may be insufficient. By combining DSE with MT, we can obtain a more comprehensive view of the quality and reliability of the test data generated by DSE.

Evaluating the ability of DSE to generate test cases that can effectively detect potential defects is crucial in order to understand its strengths and weaknesses. If the results of this study reveal any weaknesses, we can analyse these and work towards improving DSE

in a more general way. This is important, as the industrial context often involves a wide range of projects with different requirements. By improving DSE, we can contribute to the automation of software testing, which brings numerous economic and quality benefits to industrial software development projects. In addition, reducing manual workload and improving the overall quality of software applications are important goals that can be achieved through the improvement of DSE.

5.2 Combining MT and DSE

This section begins by outlining the limitations in terms of detecting potential defects in software testing techniques that aim to cover structural coverage criteria. This is followed by a case study, in which we outline the experimental setup (common to all the experiments presented in this Chapter) as well as the results of combining MT and DSE.

5.2.1 Evaluating initial effectiveness of DSE-generated test cases for mutant killing

Structural coverage criteria may not be sufficient to measure the ability of test cases to detect potential defects in the code. For this reason, we will test the mutant-killing capability of the test cases generated with DSE. A good set of test cases should be able to kill a large number of non-equivalent mutants.

The process described in Figure 5.1 is proposed to determine the ability of DSE to kill mutants. This process requires a software under test (SUT) and tools to apply both DSE and MT. Figure 5.1 illustrates the process by generating a set of test cases with DSE and a set of mutants from the SUT. The test cases are then run on the original version of the program and its mutants, independently, to obtain their output. Finally, during the 'Mutant execution and classification' step, we compare the outputs and classify the mutants. If there is any difference between the output of the original program and its mutant, the mutant is classified as killed. Otherwise, it is classified as surviving.

The process in Figure 5.1 is not subject to any particular language or tool, so its implementation will depend on the elements chosen in each case. This chapter shows a use case with programs written in C/C++ (see Section 5.2.2), together with some results that

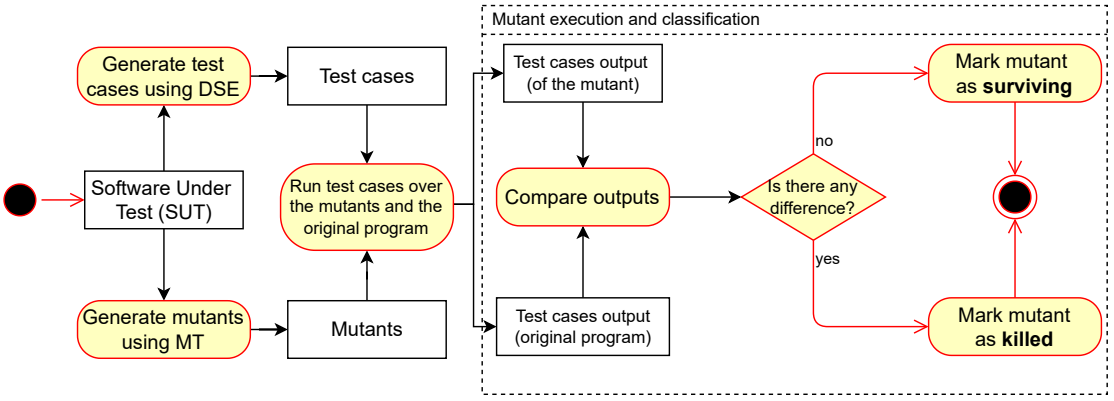

FIGURE 5.1: Evaluating mutation coverage from DSE execution on the original SUT.

have motivated further work along these lines and, therefore, to propose a new family of techniques (described in Section 5.3).

5.2.2 Case study: experimental setup

This study was fully carried out on virtual machines to improve reproducibility, divided between Docker machines [10] and some others hosted in the Google Cloud platform[1]. Mainly, the different utilities analysed have been distributed among different machines, as on some occasions the execution of an utility can take several days (this is discussed in further depth below). All the machines used are dual-core with 4GB of RAM and Ubuntu 18.04 LTS.

SUT: GNU Coreutils

GNU Coreutils[2] is a set of utilities available by default on most GNU operating systems. These utilities, written in C, are open source and handle system properties, shell, the file system and so on. This is why they are known as Coreutils, as they represent the main core of the tools used by millions of users. This means that they have been in active development for decades, and have been extensively tested.

We can find code corresponding to this set of utilities since 2001. GNU Coreutils is a well-tested and widely used tool in several research works [13, 18, 56]. The reader may find works with references to *Fileutils*, *Shellutils* and *Textutils*, whose utilities are

[1] https://cloud.google.com
[2] https://www.gnu.org/software/coreutils/

included in GNU Coreutils since 2003. We can assume that research works using *Fileutils*, *Shellutils* and/or *Textutils* are using a subset of GNU Coreutils utilities. Some of the most representative GNU Coreutils utilities are presented in Table 5.1, while the full list of utilities can be found online in the documentation[3].

Utility	**Description**
ls	list a directory
cat	copy file content to stdout
mkdir	create directory
rmdir	remove empty directory
rm	remove files or directories
cp	copy files or directories
mv	move files or directories
ln	make links
chown	change file owner and group
chmod	change file or directory permissions
dd	convert and copy a file
df	show file system disk space usage

TABLE 5.1: Some of the most representative utilities of GNU Coreutils

One of the main advantages of using this set of utilities as a case study is that they are varied in length and complexity. For example, there are small utilities with less than 100 lines of code (see *sync* or *whoami*), medium-sized utilities with between 200 and 500 lines of code (see *sum* or *touch*) or large utilities with more than 500 lines of code (see *wc* or *numfmt*).

For this case study we have selected 30 utilities from the total of those available in GNU Coreutils, which can be seen in Table 5.3. In the experiments, the output of the original program is stored together with that of the mutants for later cleaning (elements that differ in each execution, such as the ID of the processes or the available disk space, are filtered out), analysis and comparison. This allows the subsequent classification of the mutants as surviving or killed. It is this output processing that makes it reasonable to limit the utilities to a subset of them, in order to make the study more manageable and illustrative. These 30 utilities have been selected during a brief preliminary experimentation, where the rest of the utilities in the set were discarded because of the following situations:

- **Overly long output:** the execution of some of these utilities produce a considerable long output. Combined with each mutant, this results in the generation of files

[3] https://www.gnu.org/software/coreutils/manual/html_node/index.html

larger than 1GB, quickly exhausting the available space on the virtual machines. Some of these utilities are *base64*, *tee*, *users*, *printf* and *factor*.

- **Strongly variable-dependent utilities:** utilities whose output depends almost exclusively on external variables which change according to time or system circumstances (this includes disk space or the current system date and time). Utilities that have been discarded for this reason include *uptime*, *df* and *date*.

- **Excessive number of mutants:** the generation of mutants per se is a resource-light process. However, compiling and executing them is not a simple task and sometimes takes an unacceptable amount of time for experiments in this context. In Coreutils, there are tools that take about an hour to run, which is multiplied by the number of mutants, sometimes in the thousands. Utilities such as *join* and *ls*. have been discarded for this reason.

- **Alteration of vital system elements:** some utilities are designed to modify system permissions and properties, such as *chmod*, *chgrp*, *rm*, *cp* and *mv*. Modifying these parameters leaves the virtual machine in an unstable state, causing invalid results in many mutants and requiring manual reconfiguration of the machine to continue experiments.

- **Special cases:** some utilities such as *sleep* and *chgrp* have been discarded due to their behaviour not being carried over to the output in most cases. For example, to evaluate the utility *sleep* we have to measure the execution time as well as the output. However, this utility pauses program execution depending on the parameter received, causing mutants to take days or even weeks to finish testing simply because a parameter has paused the execution this long. Due to the nature of these experiments and, having 30 tools available for exhaustive testing, these tools have been discarded during the *naive MISE* experiments.

DSE Tool: KLEE

FIGURE 5.2: KLEE logo

KLEE [13] is a popular DSE tool commonly used as a testing tool and code analysis framework. According to its public repository[4], it implements symbolic execution engine, which runs LLVM bitcode modules with support for symbolic values. It works by replacing certain parts of the code with symbolic values, which are then systematically explored to generate test cases. For example, to generate test cases for a C program called "example.c", the following command can be used: `klee example.c`

KLEE will then execute the program symbolically, generating test cases that cover different paths through the program's code. These test cases can be used to validate the program's behaviour and detect any potential bugs. In addition to generating test cases, KLEE can also be used to perform code analysis and find potential defects or vulnerabilities in a program.

Additionally, it includes several auxiliary modules that benefit software testing. For example, there is a simple library for replaying tests previously generated, as well as a more complicated library that includes entries generated by the POSIX/Linux emulation layer. In this way, both closed and native programs (those which rely on the operating system) can be handled in an environment that simulates test inputs, files, environment variables, command line arguments, etc. One of the tools included as open-source with KLEE is Kleaver, a custom constraint solver that uses a series of mathematical simplifications to group constraints into independent subsets for later invoking an internal SMT (satisfiability modulo theories) solver, which will determine whether the contraints are satisfiable [13, 14, 66].

Both KLEE and Kleaver provide support for queries expressed in the KQuery language, which is fundamental later in this work (KLEE uses them internally, while Kleaver uses them as input). The KQuery language[5] is a textual representation of constraint expressions and queries resulting from symbolic execution, used as input to Kleaver and

[4] https://github.com/klee/klee
[5] https://klee.github.io/docs/kquery/

as an intermediate step in KLEE. This language is able to represent formulas on vectors and arrays, offering support for all standard operations. Its design, compact and easy to read and write, is strongly related to the C++ API for *Expr*, which can be consulted for more information in [2].

The following is an example of how a query written in KQuery works. Listing 5.1 shows a small function with three possible paths, while Listing 5.2 shows a query specifically designed to traverse Path 2. In line 1, we see that an array of 4 elements is declared, which will serve as a symbolic value. The query, in a simple way, asks the solver to return a value whose restrictions are that the value must not be 0 (*Eq false (Eq 0...)*) and also, the value must be signed and less than or equal to 0 (Slt N0 0). Combining both restrictions, the solver will return a value strictly less than 0. We can see that both constraints can exist individually, but their combination returns the expected value.

```
1 int get_sign(int x) {
2   if (x == 0) return 0;     //Path 1
3
4   if (x < 0) return -1;     //Path 2
5   else return 1;            //Path 3
6 }
```

LISTING 5.1: Test function

```
1 array a[4] : w32 -> w8 = symbolic
2 (query [(Eq false
3               (Eq 0
4                   N0:(ReadLSB w32 0 a)))
5         (Slt N0 0)]
6         false [] [a])
```

LISTING 5.2: KQuery for path 2

In this research, we have used KLEE with a configuration similar to that of its authors in previous work with GNU Coreutils [13], with some minor adjustments to adapt it to our environment. The available memory has been increased to 4GB, although the applications rarely exceed this limit. This memory also works well with the configuration of the virtual machines. For the symbolic execution time, we have conducted a small experiment to check the time consumed by each GNU Coreutils utility. In this way, we have launched the utilities for 30 minutes, 1, 2 and 3 hours respectively, finding that most of the applications in our subset consume all the time granted. Even so, the increase in time rarely produced a significant improvement in the results, so it has been decided to maintain the 1 hour limit of symbolic execution for all applications (similar to the previous study mentioned at the beginning of this paragraph). Finally, regarding the search strategy, we have opted for BFS (Breadth-First Search). This strategy has been

used previously in similar studies [18] with good results, and consists of focusing search efforts on those paths directly affected by the inputs. As our objective is the generation of test cases, it fits well with our study.

MT Tool: MuCPP

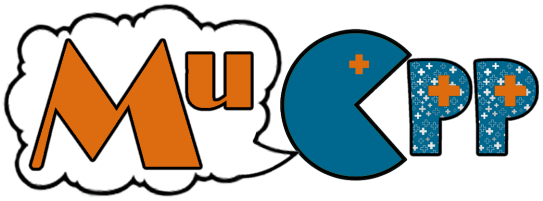

FIGURE 5.3: MuCPP logo

MuCPP [24] is a tool for applying MT in C and C++ programs. While there are other options for applying MT depending on the language or code environment [17, 20], it should be noted that our SUT is GNU Coreutils (written in C) and our tool for applying DSE is KLEE (dedicated to C and C++ code), so MuCPP strongly fits our needs. MuCPP implements mutation operators in a robust way and for different levels of abstraction, incorporating from the most traditional mutation operators to class level operators. This tool incorporates the changes directly into the source code. More specifically, MuCPP generates mutants as it traverses the abstract syntax tree of the code with the Clang API, while saving the mutants in branches of the Git version control. This allows the user to save space, classify mutants, explore them and query for changes efficiently. In addition, MuCPP incorporates some mechanisms to avoid the generation of some duplicate mutants and mutants in system-specific headers, and facilitates the entire process of compiling and running mutants and tests, without being tied to any specific testing framework.

For this case study, and considering the characteristics of the GNU Coreutils utilities, it has been applied a set of 12 traditional mutation operators implemented in MuCPP (see Table 5.2, where these operators are described together with their internal ID for better identification). More information about the inner workings of these and other operators can be found in [24]. Furthermore, as a subsequent step after the application of MuCPP to the utilities, we applied TCE [70] to the generated mutants with the aim of eliminating as many equivalent and duplicate mutants as possible. TCE (Trivial Compiler Equivalence) is a technique that determines whether two pieces of code are equivalent

in terms of their behaviour. This can be used to optimise the test generation process by identifying and eliminating redundant test cases that do not provide any additional coverage or reveal new defects. It is also possible to identify equivalent mutants, as their behaviour should be exactly the same as the original program. TCE can be implemented by comparing the output of the two pieces of code for a given input, or by analysing the intermediate representation of the code to identify any differences in the way that the code is executed. This intermediate representation comparison is actually the approach we take to detect equivalent mutants using TCE. Specifically, we generate and compile all mutants and compare their intermediate representation, discarding any equivalent or duplicated mutants.

Operator (ID)	**Description**
ARB (31) ARU (32) ARS (33)	Arithmetic Operator Replacement (Binary, Unary and Short-cut)
AIU (34) AIS (35)	Arithmetic Operator Insertion (Unary and Short-cut)
ADS (36)	Arithmetic Operator Deletion (Short-cut)
ROR (37)	Relational Operator Replacement
COR (38) COI (39) COD (40)	Conditional Operators (Replacement, Insertion and Deletion)
LOR (41)	Logical Operator Replacement
ASR (42)	Short-cut Assignment Operator Replacement

TABLE 5.2: Traditional mutation operators included in MuCPP

5.2.3 Evaluation results

This section presents the results obtained from applying the procedure shown in Section 5.2, whereby we evaluate the initial ability of DSE-generated test cases to kill mutants.

Table 5.3 is a complete outline of all the results obtained after the runs at this point. Each row represents a different utility, whose name corresponds to the first column and its size in the second column (LOC for Lines of Code). The third and fourth columns correspond to the mutants before and after applying TCE, where as many duplicate and equivalent mutants as possible are removed. This does not mean that there are no duplicate and

equivalent mutants in the fourth column, as TCE is not an exact technique, but a large number of them have been removed, giving more confidence to the results obtained. The last two columns show the killed mutants and the mutation score respectively, which is the objective data sought. Finally, it is important to emphasise that the last row, separated from the rest in Table 5.3, corresponds to the total sum of the data (except for the percentage of killed and alive mutants, which is the average of the values) to give the reader an overall idea of the magnitude of the experiment and the results obtained at a glance.

Program	**LOC**	**Mutants before TCE**	**Mutants after TCE**	**Mutants killed**	**Mutants killed (%)**	**Mutants alive (%)**
basename	132	36	29	14	48.2%	51.8%
chcon	446	41	18	16	88.8%	11.2%
chgrp	249	23	22	22	100.0%	0.0%
chown	258	12	12	9	75.0%	25.0%
chroot	197	49	35	8	22.8%	77.2%
cksum	225	120	110	67	60.9%	39.1%
dirname	98	12	11	8	72.7%	27.3%
echo	213	29	21	7	33.3%	66.7%
expr	790	257	254	135	53.2%	46.8%
false	2	5	5	3	60.0%	40.0%
link	60	12	11	8	72.7%	27.3%
logname	56	12	11	8	72.7%	27.3%
md5sum	657	280	216	25	11.6%	88.4%
mkdir	224	111	63	27	42.9%	57.1%
nproc	94	12	11	8	72.7%	27.3%
numfmt	1,110	1,071	795	254	31.9%	68.1%
pathchk	297	38	21	18	85.7%	14.3%
pwd	263	268	260	18	6.9%	93.1%
realpath	221	25	24	17	70.8%	29.2%
rmdir	171	70	43	24	55.8%	44.2%
sleep	105	33	23	17	73.9%	26.1%
stdbuf	278	103	68	39	56.5%	43.5%
sum	200	157	146	83	56.9%	43.1%
sync	45	12	11	8	72.7%	27.3%
touch	313	111	78	49	62.8%	37.2%
truncate	335	165	164	65	53.2%	46.8%
tty	80	12	11	8	72.7%	27.3%
uname	281	116	57	42	73.7%	26.3%
wc	624	688	546	338	61.9%	38.1%
whoami	63	12	11	8	72.7%	27.3%
Total	8,087	3,892	3,087	1,353	59.9%	40.1%

TABLE 5.3: Killed mutants per program in the first evaluation

Some of the results shown in Table 5.3 are worth highlighting, as they are in line with what was previously presented in Section 5.2.2. Firstly, we can see the variety of the size of the utilities within GNU Coreutils in the LOC column. For example, utilities like *sync*

(45 LOC), *whoami* (63 LOC) or *false* (2 LOC) represent small sized programs. Other utilities such as *chgrp* (249 LOC), *chown* (258 LOC) or *uname* (281 LOC) represent medium-sized programs. Finally, utilities like *numfmt* (1110 LOC) or *expr* (790 LOC) represent larger programs. Note also that the number of mutants is not necessarily proportional to the number of LOCs. For example, MuCPP generated 5 mutants in *false* (2 LOCs), 280 mutants in *md5sum* (657 LOCs) and 268 mutants in *pwd* (263 LOCs). In general, the number of mutants generated in a program depend on the features of its source code. The latter is best understood by looking at the killed mutant results together with Table 5.4.

Operator	**Mutants**	% **Surviving**
AIS	1,093	46.11%
AIU	343	53.94%
LOR	15	46.67%
ARS	38	57.89%
ARB	52	71.15%
ADS	22	59.09%
ROR	157	43.31%
COD	30	56.67%
COI	239	30.54%

TABLE 5.4: Mutants generated per mutation operator (excluding numfmt)

Table 5.4 shows the number of mutants generated by each mutation operator. Please note that the *numfmt* utility has been removed from this table, as it presents a high number of mutants (+1000) with a relatively homogeneous distribution of mutation operators, altering the overall percentages and giving a misleading picture concerning the impact of the mutation operators. The number of mutants is not uniformly distributed among the set of mutation operators. For example, the AIS operator produces more than 1000 mutants while other operators generate hardly any mutants (LOR with 15 or ADS with 22, respectively). Note that this also depends on the behaviour of the mutation operators, as insertion operators usually find more possibilities to incorporate a mutation into the source code, while replacement or deletion operators need certain arithmetic or logical operators to be present in the initial code.

Going back to the results in Table 5.3, the results are mixed. Some tools such as *chcon*, *chgrp* or *uname* reach 70% mutation coverage, while others do not even reach 50% coverage. On average, DSE is able to kill 59.9% of the set of mutants after applying TCE. A manual analysis reveals that, with stronger test data, part of those mutants

would have been killed. For example, many mutants, especially those generated by the AIS operator, are still alive because their mutation affects a threshold value. These are the values used to determine whether a condition is true or false, so a small variation in these values will cause the program to take one path or the other. And it is in this small variation that DSE has a weakness, as it uses values for the tests that are far from the threshold values. A good example of this is a mutant that modifies the condition of a loop whose condition "sp > suffix" becomes "sp >= suffix" (with our configuration, the mutant is *m37_3_1_basename*). With the generated test data, it is the *suffix* variable that acts as the threshold value. A value of *suffix* strictly less than *sp* will always evaluate to *true*, both in the mutant and in the original program, so the mutant will remain alive. The only test data that will detect the mutant in this case will be that where "sp == suffix".

Therefore, the conclusion is that there is ample room for improvement in terms of DSE's ability to kill mutants and, consequently, detect potential defects. Although the percentage of killed mutants is 59.9%, as DSE focuses its efforts on covering traditional coverage criteria, we can say that there are many surviving mutants left to be killed (40.1% in our case).

5.3 Defining Mutation-Inspired Symbolic Execution (*MISE*)

From the extensive work carried out to this point, it is evident that there is ample room for improvement in terms of mutation coverage when using DSE. For this reason, we propose the integration of DSE and MT to generate high quality test data for detecting mutants. This proposal leads us to present *Mutation-Inspired Symbolic Execution (MISE)*, a new and sophisticated family of techniques relying on the combination of both techniques. Thus, their integration can be undertaken in many ways depending on the programming language in use, the tools, or the case studies. To evaluate the potential of MISE, we present a technique whose implementation does not require the alteration of any of the techniques or tools that compose it, which we will call *naive MISE*.

5.3.1 *Naive MISE*: an initial combination of DSE and MT

Naive MISE incorporates MT into the process by applying DSE to each non-equivalent surviving mutant separately (see Figure 5.4). This technique goes one step further compared to what was shown in Section 5.2, so that DSE is re-applied to each of the surviving mutants with the goal of generating new test cases capable of killing new mutants. Eventually, new test cases should kill the mutant from which they are generated. The process ends when there are no surviving mutants left, or all surviving mutants have been analysed with DSE. At this point, it is conceivable that the remaining surviving mutants may be equivalent mutants or may require further study to see why they have not been killed. Applying naive MISE will hopefully result on a set of test cases that will have a higher potential mutation coverage than the initial results obtained from Sections 5.2.3.

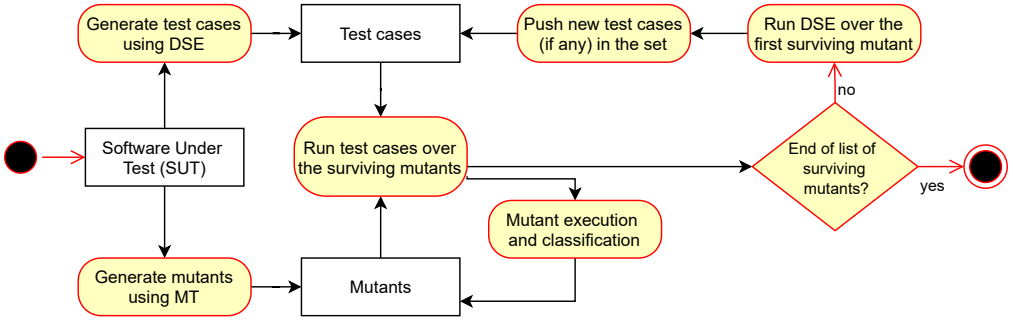

FIGURE 5.4: Naive MISE: combining DSE and MT for improved mutation coverage.

Naturally, despite being a mainly illustrative technique, we cannot overlook the high implementation cost involved. Because of the need to run DSE on each of the surviving mutants, the computational cost will be multiplied by every surviving mutant. Therefore, in Section 5.4 we propose a more refined set of ideas for its implementation.

5.3.2 Implementing *naive MISE*

Section 5.2.3 shows how there is still a considerable percentage of surviving mutants when using exclusively DSE-generated test cases. What follows, therefore, is an experiment to see if the percentage of killed mutants increases when DSE is applied on the surviving mutants to generate new test cases. In this way, the effectiveness and feasibility of combining MT with DSE can be proven straightforwardly.

This section shows the results obtained after implementing the steps described in Section 5.3, using the same experimental setup shown in Section 5.2.2. Table 5.5 summarises

the results, reflecting the changes in the killed mutants as a result of the newly generated test cases (it does not include the utilities where no improvement was observed when applying DSE on their surviving mutants). Naive MISE increases the mutation score in 9 utilities. While in some utilities like *realpath* or *basename* only one more mutant is killed, other utilities like *cksum* or *md5sum* increase the percentage of killed mutants considerably.

Program	**Mutants killed (initial tests)**	**New mutants killed**	**Mutants killed (%)**
basename	48.2%	+1	51.7%
cksum	60.9%	+18	77.3%
expr	53.2%	+6	55.5%
md5sum	11.6%	+35	27.7%
numfmt	31.9%	+3	32.3%
realpath	70.8%	+1	75.0%
stdbuf	56.5%	+4	63.2%
sum	56.9%	+14	66.4%
touch	62.8%	+10	75.6%

TABLE 5.5: Percentage increment of killed mutants

These results require to be contextualised to be interpreted properly. Looking at Table 5.3 again, we see how some of the utilities already had little room for improvement with only 2 or 3 surviving mutants (*chcon*, *chown*, *dirname*, *link*, *logname*, *nproc*, *pathchk*, *sync*, *tty* and *whoami*). We also have to consider the possibility that some of the surviving mutants in these utilities turn out to be equivalent mutants. However, the sheer number of mutants makes it impractical the manual analysis of all of them to determine this condition with certainty.

A manual analysis of the new test cases that kill mutants reveals the existence of crossfire mutants, which are those mutants killed by test cases initially designed to kill other mutants [86] (i.e., those mutants killed by test cases generated by DSE from other mutants). This has the side effect that the initial *naive MISE* time estimates (up to one hour per mutant) may be subject to variations, as it is not necessary to apply DSE on already killed crossfire mutants. Table 5.6 shows an illustrative example of crossfire mutants present in the *cksum* and *sum* utilities, although this phenomenon is also present in other utilities such as *md5sum*. The table shows the utility (first column), followed by the name of the mutant that generates the test cases (second column) and ending with a list of its crossfire mutants.

Program	Mutant	Crossfire Mutants
cksum	m34_1_2_cksum	m33_1_2_cksum m33_2_1_cksum m33_2_2_cksum m34_10_1_cksum m34_16_1_cksum m34_17_1_cksum
sum	m35_11_2_sum	m34_11_1_sum m34_24_1_sum m34_30_1_sum

TABLE 5.6: Example of detected crossfire mutants

Examining the different mutants has allowed us to identify several reasons why the test cases generated from DSE are limited in terms of mutant detection (see Table 5.3). For example, while the *basename* utility presents a low number of equivalent mutants, *naive MISE* is able to kill only one more mutant. These reasons are shown below with an acronym, along with some illustrative examples in Table 5.7.

R1 - *The mutation is not analysed during the first symbolic execution.* The fact that KLEE is focused on traditional coverage criteria allows us to use line coverage in an auxiliary and simple way thanks to the advances made by its authors[6]. We have found some points where mutations are applied but which, however, are not covered by KLEE when analysing the original program. In our case study, this phenomenon is usually observed when the symbolic execution ends without evaluating concrete conditions (either because the runtime is over or because the generated constraint is too complex to be solved by the solver).

R2 - *The initially generated values are too far from the conditional thresholds.* It is common to have mutation operators that insert a change in the conditions of the source code, such as those present in conditional or loop statements. If we look at the test values generated by DSE, these values are far away from the threshold values that affect these conditions. This means that the mutants generated by operators like `ROR` —which change relational operators, e.g., $>$ to $>=$—, will hardly be detected by those values.

R3 - *The mutation does not affect the specific point where the condition checks are performed.* That is, it is applied at an earlier point in the code where it alters the result

[6] https://klee.github.io/tutorials/testing-coreutils/

of one or more variables that are used in a later condition, and this changes the initial outcome. This is why this change is not always accounted when generating test data.

All the cases described (R1-3) could be alleviated considerably if, instead of focusing the generation of DSE test cases on traditional coverage criteria, mutation coverage was targeted as well.

Reason	**Program**	**Mutant**	**Operator**
R1	numfmt	m37_10_1_numfmt	ROR
	cksum	m34_18_1_cksum	AIU
R2	basename	m37_3_1_basename	ROR
	touch	m35_4_3_touch	AIS
		m35_6_1_touch	AIS
R3	sum	m31_3_1_sum	ARB

TABLE 5.7: Illustrative set of mutants with the reasons identified for the limited mutant detection of DSE.

In cases in which the point where the mutation occurs is not analysed during the first symbolic execution (see R1), it seems that certain variables are not accounted for during the pre-analysis performed by the symbolic execution engine. This is primarily because other variables manage to alter the paths with sufficient weight so that it is not necessary to change others to traverse paths, thus maintaining fixed values for the variables with less weight in the decision. When applying a mutation operator, the weight of the variables changes, and there are situations in which the paths change drastically depending on the value assigned to the variables affected by the mutation. For this reason, we can observe that, when DSE is applied again on the mutants, this time this variable is taken into account to generate test cases and, consequently, the number of killed mutants increases, as shown in Table 5.5. On the other hand, the reasons identified as R2 and R3 are related, as they both affect the conditions. However, looking at how symbolic execution engines currently work, R2 is more solvable. This stems from the fact that it is easier to know which variables are present in the conditions, as well as the values that affect the paths rather than those that affect them indirectly. R3 presents a challenge that deserves future in-depth analysis, as it needs to break down the code beyond how current DSE solutions do it.

As for the mutation operators that generate most of the new killed mutants, these belong to the operators `AIU`, `AIS` and `ROR`. This is not surprising, since looking at Table 5.4, we can see that these three operators generate a high number of mutants (`AIS` the first, `AIU` is the second and `ROR` the fourth). Not only that, but the nature of these operators (insertion and replacement) lead to the generation of mutants prone to cause the situations described in R1-3. Specifically, the `AIS` and `AIU` operators introduce signed and unsigned arithmetic operators (`-`, `++` and `--`). This often causes small changes that are difficult to detect if the test values have not been thought with them in mind, so it is common to find the situations described in R2 and R3. The case of the `ROR` operator is slightly different, since it is in charge of replacing the relational operators present in the code. These replacements significantly affect the conditions, so we can easily find situations like the ones described in R1 and R2.

If we explore a subset of surviving mutants, the `AIU`, `AIS` and `ROR` operators again stand out due to their high incidence. Even so, this time the emergence of the `COI` (condition negation) mutation operator is noteworthy, as the test cases generated by DSE barely kill mutants of this type. Sometimes, when the symbolic execution engine scans the paths, the negation of the condition is not affected, in the sense that it will still be able to scan both paths with the same data set. In this case, the result will be the same test suite, only changing the order of the test cases. For this reason, new test cases are hardly generated when re-applying DSE on such mutants. This is again a consequence of the fact that DSE focuses test case generation on structural coverage criteria and not on mutation coverage.

Overall, *naive MISE* manages to generate new test cases capable of killing more mutants, demonstrating that the combination of DSE with MT has potential benefits. These new test cases not only kill the mutants from which they originate, but also cause the appearance of crossfire mutants, thus increasing the percentage of killed mutants. However, it is important to note that *naive MISE* comes at a high cost, so it serves as an inspiration and guide for the creation of this new family of techniques (MISE).

5.4 Future *MISE* implementations

The results achieved when applying the combination of DSE and MT to the case study described in Section 5.2.2 are promising, as there is a substantial improvement in the mutation score. However, both DSE and MT are very expensive techniques in themselves, making the implementation described in Section 5.3 very costly. As such, the cost of applying DSE is multiplied approximately as many times as the number of surviving mutants after the first symbolic execution. To overcome this high cost, we propose to improve the inner workings of DSE in order to generate test cases whose main objective is mutation coverage rather than structural coverage criteria. This is possible by incorporating elements specific to MT into the entire internal DSE process, with the aim of obtaining a better set of test cases in terms of mutation coverage without the need to apply DSE to each of the surviving mutants individually.

Considering the reasons outlined in Section 5.3.2, we have identified three clear opportunities for improvement in DSE to generate test cases without significantly compromising the cost of the technique: reinforcing the threshold values approach, modifying constraints, and considering variables that directly affect the program output.

These three areas of improvement are described in detail below. The reader will note that we do not use GNU Coreutils for the examples, but simplified source code fragments. Even though the three cases described have been found in the utilities used for the experiments up to this point, the intention of this section is to be as clear and descriptive as possible, and we deem that the used examples will help to better illustrate these situations.

5.4.1 Reinforcing the threshold values approach

A manual analysis of the test cases in the above case study reveals that commonly the values generated by DSE are far away from the threshold values. For example, in the case of *int* variables within a condition, we can see that sometimes they are assigned values like 16843009 or -2147483648. If we bear in mind the branch coverage criterion, these values are rather adequate, as they allow us to evaluate such condition as *true* or *false*. Mutation testing, however, is a stronger criterion and the selected values should be more specific to be able to kill the mutants. It requires more knowledge regarding the

domain or values close to the boundary of the conditions [27]. A real situation can be found in a mutant generated by the mutation operator `ROR` in the utility *numfmt* (line 477), where a < operator is changed to <=. As the test data is -2147483648, the mutant is not killed by the initial test suite.

To better understand the scope of this approach, let's look at the example of Listing 5.3.

```
1 int exampleSE (int b) {
2     int a;
3     a = b * 10;
4     if ( a == 20 ) {
5         throw Exception( );      //Path 1
6     } else return a;             //Path 2
7 }
```

LISTING 5.3: Illustrative method

To traverse the two possible paths of Listing 5.3, the variable `b` must take two different values, which the solver will produce as a result of two constraints: '`b` * 10 == 20' and '`b` * 10 != 20'. Solving both constraints manually, we could say that '`b` = 2' and '`b` = 3' are sufficient to cover both paths. Now, suppose we apply the ROR mutation operator at this point so that the condition '==' is changed to '<=' in line 4. With the two proposed values, the mutant would still be alive, and it is clear that it is not an equivalent mutant. If we choose another value, such as '`b` = 1', the change would be noticeable, as it would result in a test case that explores the first path in the original program, while it explores the second path in the mutant, thus making a difference in the program output and detecting the mutant. Table 5.8 shows the paths that the execution would follow according to the test values entered in the original program and the mutant.

	Test cases		
Version	**'b = 2'**	**'b = 3'**	**'b = 1'**
Original	Path 1	Path 2	Path 2
Mutant (== by <=)	Path 1	Path 2	Path 1

TABLE 5.8: Paths explored in Listing 5.3 with different values for the test cases.

The proposed value (`b` = 1) corresponds to what is expected for killing mutants, as it is a threshold value of the condition that allows for easy detection of changes in the

condition. Therefore, incorporating the knowledge derived from the MT elements for test case generation, rather than values far from the thresholds, allows this situation to be solved. For this reason, it would be beneficial if DSE could be extended to generate at least three different values in these terms: a threshold value, a smaller one and a higher one. This is just one example of all the variations that can be introduced by mutation operators, and there are other situations that would be beneficial, such as generating small variations in text strings. Therefore, it would be necessary to generate more test data for the test cases generated by DSE in its current state, which could help to kill more mutants without the need to apply DSE several times as done in *naive MISE*.

5.4.2 Modifying constraints

It is possible to generate new test data from the constraints as they are being generated in the symbolic execution, or after it is finished. In KLEE, two fundamental elements allow this: the KQuery language and the Kleaver solver. In this section we propose the modification of KQuery constraints for the generation of new test data. Specifically, the approach consists of using a KQuery constraint initially generated by DSE and adding small modifications based on the characteristic mutations of MT. These new constraints will be sent as input to Kleaver, which will solve them and then produce potentially new test data.

To show the feasibility of the proposal, observe the code snippet of Listing 5.4, which is one of the functions distributed as an example with KLEE[7]. Two conditions apply: '$x == 0$' and '$x < 0$'. After using KLEE with this function, three KQuery files are generated with constraints that will return three different values: '$x = 0$', '$x < 0$' and '$x > 0$'.

```
1 int get_sign(int x) {
2   if (x == 0) return 0;     //Path 1
3
4   if (x < 0) return -1;     //Path 2
5   else return 1;            //Path 3
6 }
```

Listing 5.4: Get Sign example

[7] https://github.com/klee/klee/blob/master/examples/get_sign/get_sign.c

Listing 5.5 shows one of the KQuery queries that KLEE generates during the symbolic execution of the code presented in Listing 5.4. This query generates a value specifically intended to cover path 3. In this example, `N0` is a symbolic value representing the variable `x` in the code. In first place, this query restricts the value of `N0` to be different from 0 (lines 2 and 3). Then, another restriction is added so that `N0` is strictly greater than 0 (*Slt* stands for *Signed less than*, but it is negated). After executing this query in the solver, the result is 16843009, a value that meets the two aforementioned conditions.

One of the most expensive parts of symbolic execution is the exploration of paths before the generation of constraints like the one in Listing 5.5. Modifying already generated KQuery constraints makes it possible to obtain new test data without the need to re-apply DSE and thus dispense with the high cost of re-exploring paths. The following is an example of how a modified KQuery constraint is able to generate test data to execute other parts of the source code. Notice line 5 of Listing 5.5, where the value of N0 is restricted to be strictly greater than 0 (technically, it could be equal to 0, but this possibility is covered by the previous constraint). Removing the negation of this constraint, as we see in Listing 5.6, now the value must be strictly less than 0. Solving the KQuery of Listing 5.6 with Kleaver, results in the value -2147483648, which fulfils the conditions for exploring path 2.

```
1 array a[4] : w32 -> w8 = symbolic
2 (query [(Eq false
3              (Eq 0
4                  N0:(ReadLSB w32 0 a)))
5          (Eq false (Slt N0 0))]
6         false [] [a])
```

LISTING 5.5: KQuery of a test case

```
1 array a[4] : w32 -> w8 = symbolic
2 (query [(Eq false
3              (Eq 0
4                  N0:(ReadLSB w32 0 a)))
5          (Slt N0 0)]
6         false [] [a])
```

LISTING 5.6: Modified KQuery

This example shows that it is possible to introduce small modifications to constraints in a similar way as MT does in the source code in order to obtain new test data, resulting in new test cases. The cost of solving these constraints is usually much lower than the initial cost of generating them. Hence, we can generate new constraints in a single symbolic execution, resulting in a larger set of test cases with a higher potential to kill

mutants. This would avoid having to run DSE more than once, reducing the total cost considerably.

5.4.3 Considering variables that directly affect the program output

As stated so far, DSE is designed to cover the structural coverage criteria, so the test values are intended to cover all execution paths. Therefore, a variable will only be addressed by DSE when the paths taken depend on its value. This implies that the rest of the variables will not be taken into account for the constraints, even if they affect the program output. This is a weakness of DSE and the reason why some mutants remain alive.

This situation can be a little more difficult to understand, so we show a real situation where it occurs. Listing 5.7 represents a small fragment of the *sum* utility. Specifically, it is part of the *m31_3_1_sum* mutant, where the `ARB` mutation operator has replaced the `+` operator with the `-` operator in line 2. This mutant survives during the experiments conducted in the case study.

```
...
checksum = (r & 0xffff) /* ARB */ - (r >> 16);

printf("%d %s", checksum,
    human_readable (total_bytes, hbuf,
                    human_ceiling, 1, 512));
...
```

LISTING 5.7: Sum example of variable

Looking closely at the code fragment of Listing 5.7, we can see how the value of the variable `r` affects the output of the program through the variable `checksum`. While it does affect the output, it never affects the program's paths, since it is a line that is always executed regardless of the value of its variables. The execution of DSE at this point always assigns the same value to the variable `r`. The value assigned, although fixed, will depend on the search strategy implemented. Since it is a subtraction in the case of the mutant (or an addition in the case of the original program), it is logical to assign it the neutral value of these operations `r = 0`, enough to cover the lines of Listing 5.7.

However, a value that does not alter the output in any case, so the mutant will remain alive. In fact, it is necessary for `r >> 16` to produce a value other than 0 in order to distinguish the mutant from the original program.

In their current state, tools such as KLEE do not have the ability to detect variables that affect the output in such a way that certain different values are proposed. It is possible to manually indicate variables, but this requires extensive manual analysis of each of the mutants. Thus, we propose to extend KLEE to deal with variables in the context of mutations, which poses two main challenges. The first challenge is to identify variables that should be made symbolic, while the second challenge is to generate values that help detect mutants by their output.

Identification of symbolic variables

The automatic identification of such variables is an important challenge that can be tackled on several fronts. One option is to modify KLEE to be aware of functions that affect the output, either because they return a value or because they display something via any of the output streams. As a first approximation, in addition to the values returned by the functions, it is possible to trace functions from the standard C library, such as `printf` (C) or `std::cout` (C++). This would be useful in the Listing 5.7 example, since `checksum` is used as an argument to `printf`; then, following its dependencies, we would find `r`, a variable that affects the output of the program. This new functionality would still need some limits imposed by the user, in order to keep the computational cost at reasonable levels. Still, it would allow more mutants to be killed in a single symbolic execution.

Generation of relevant values regarding mutants

As with the identification of variables, the generation of relevant values to be able to detect more mutants can be done from several approaches. A first approach would be for the solver to propose several random values for the identified variables, rather than a single fixed value as is currently done. While this does not guarantee killing mutants, it does increase the likelihood of finding them as there are more test values. Another more sophisticated approach is to explore the subexpressions that affect the

variables. Returning to the example of Listing 5.7, we can see that `r` is affected by the subexpressions `r & 0xffff` and `r >> 16`. As we have seen before, a test case where `r >> 16` produces a value other than 0 would be enough to kill the Listing 5.7 mutant. Evaluating these subexpressions would help in finding new values, especially if we had mechanisms that took into account the changes that mutation operators normally introduce.

5.5 Chapter conclusions

This chapter contributes a comprehensive study on the ability of DSE to generate test data capable of detecting potential defects in real software. It also studies the effect of combining DSE with MT, resulting in a new family of testing techniques called *mutation-inspired symbolic execution (MISE)*. This family of techniques focuses on incorporating elements of MT within DSE, specifically, in the automatic process of test case generation. The study is divided into three parts:

1. An analysis of the initial mutant-killing capacity of test cases generated with DSE based on a case study on a set of real utilities.

2. A case study where we apply DSE on the surviving mutants resulting from the first part of the study, with the aim of obtaining new test cases capable of killing more mutants. This is the study that results in MISE, thanks to the findings made after the execution of a first approach to the proposal (naive MISE).

3. An analysis based on the results of the second experiment, where opportunities for improvement are identified and three more sophisticated implementations of MISE are proposed in the hope of obtaining better results.

In our case study, after eliminating a good number of equivalent mutants through TCE, DSE (applied through the KLEE tool) is able to kill on average 59.9% of the mutants. These results indicate that there is ample room for improvement, as there are still around 40% of surviving mutants. The most basic implementation of MISE is able to increase the number of mutants killed by up to 16% in some of the utilities. While this is a promising result, the combination of DSE and MT comes at a very high cost, as the

cost of applying DSE is multiplied by the number of surviving mutants in the worst-case scenario. In order to reduce this high cost, we propose three more sophisticated implementations of MISE, which aim to increase the number of mutants killed while maintaining a reasonable cost, ideally in the same order as running DSE once.

To conclude, we have identified a number of weaknesses of DSE when it comes to generating test cases to detect real defects. The proposed solutions involve modifying or extending tools such as KLEE specifically for this purpose. The modification of this type of tools and the elements that compose it is not something new, as other authors have also considered or implemented modifications in KLEE to make it smarter in other areas, such as the generation of more useful and readable strings in test cases (Yoshida et al. [97]), or the generation of more complex values such as arrays and bit vectors (Dustmann et al. [31]). Our proposal is to extend KLEE (or any DSE tool) to be able to generate more constraints by including the improvements described throughout this chapter. In this way, DSE tools would incorporate MT knowledge and would be able to achieve similar results to those obtained with *naive MISE* in Section 5.3.2, but reducing costs significantly by not needing more than one symbolic execution.

All MISE implementations are applicable to any kind of software system, as long as the DSE and MT tools compatible with the environment are available. In particular, it could be combined with ASkeleTon to obtain high quality test cases in terms of mutant detection. A full design and implementation of the solutions proposed in this chapter is part of the proposed future work, as they involve several promising but different lines of research (research on tools, solvers, constraints, etc.). Even so, significant progress has already been made in this regard, as shown below in Chapter 6.

Chapter 6

Results

"We're having an information explosion, and it's certainly obvious that information is of no use unless it's available."

Mary Kenneth Keller

This chapter summarises the results derived from the research lines of this PhD thesis. It concludes with a discussion of the results as well as of the integration and interaction between each of the different lines of research.

6.1 Industrial experience

In the first line of research of this PhD work, we took advantage of our proximity to the industrial environment of the Bay of Cadiz to understand its needs beyond what is discussed in the literature. In an effort to reduce the gap between industry and academia, we carried out a close collaborative work in which we identified two fundamental issues.

- The cost of software testing can account for up to 40% of the total cost of projects. In large-scale projects, which typically exceed millions of euros, this is a considerable amount.
- Commercial solutions do not adapt to the changing needs of individual software projects, which can take years to develop and involve different technologies.

During an industrial immersion experience with constant site visits and staff interviews, we were able to identify up to four differentiated stages during the development of software testing: source code analysis, test harness generation, test data generation and industrial validation. Each of these phases is described in detail in Chapter 3. It is worth noting that they are usually carried out by different teams working on different projects, hence the effort and communication work is often substantial. Test automation, even if not complete, has a number of potential benefits associated with it, from economic benefits (estimated savings of up to 13% in this context) to those associated with quality, both in terms of product and workplace, where a more robust product can be produced while employees work in a more relaxed environment.

It is clear from the knowledge acquired in this industrial experience that a solution for the complete automation of the testing process is needed. Based on the need for cost reduction, the development of modular tools, and the focus on separate testing stages for increased flexibility, the rest of this PhD work presents a solution that addresses these industrial needs and aims to improve the testing process through automation

6.2 ASkeleTon

As a result of the second line of research of this PhD work, we developed ASkeleTon, a tool for automating the generation of test harnesses from a SUT. This tool is able to automate the first two stages of test development identified during the industrial experience: source code analysis and test harness generation (see Chapter 4). The source code analysis stage allows for scalable search patterns to obtain information about the code, while the test generation stage uses the results of this analysis to automatically generate the test harness. Together, these stages form ASkeleTon, a tool for automating the early stages of the testing process. Even though it is not its main goal, this framework is also able to partially cover the third stage by generating initial test data for the test harnesses.

ASkeleTon arised from the field work done during Chapter 3, so its main functions are inspired by the observations and industrial feedback received. It stands out for its high modularity, both in its source code and in the products generated, as it allows each of

the parts to be easily maintained and reused. In this way, its adaptation to the changing needs of the industry could be more easily achieved.

This tool parses the source code into an AST out of which an analysis of its elements is performed, resulting in a complete test harness ready for execution. As long as the SUT does not require special configuration, it is possible to compile the test harnesses right after their generation. The generation of test data can be delegated to an external source, allowing the modification of the used test data at any time without the need to recompile the tests or the SUT. This feature is essential for the maintenance of test data and the incorporation of the latest state-of-the-art techniques for test data generation.

All the details of ASkeleTon, including its design, development and generated products are described in detail in Chapter 4.

6.3 Mutation-Inspired Symbolic Execution (MISE)

The third line of this PhD work presents a comprehensive study that starts from evaluating the effectiveness of the test cases generated with DSE to the presentation of a new family of testing techniques. Consequently, we have obtained a variety of encouraging results which motivate future work in this research line.

Firstly, and after highlighting the good results of DSE tools for test case generation in terms of structural coverage criteria, we propose to assess the effectiveness of this technique in terms of mutation coverage. Specifically, we set up a study with three main elements: KLEE as a symbolic execution tool, MuCPP as a mutation testing tool and a set of GNU CoreUtils utilities as SUT. The results show that the detection of mutants is relatively low, suggesting that the quality of the tests generated with DSE is not as good when using other more complex coverage criteria like MT as when using traditional coverage criteria.

Secondly, we manually study the surviving mutants, finding that many of them are not equivalent and can still be killed with the correct tests. After studying the possible reasons why DSE failed to detect them in the first place, we propose to re-apply DSE on the surviving mutants to see if new test cases capable of killing more mutants are generated. The results indicate that up to 16% more of the total mutants are killed. This

confirms that combining the two techniques in the test generation process significantly improves the initial results. However, this approach comes at a very high cost, as DSE, a costly technique in itself, has to be applied to each surviving mutant independently. The combination of the two techniques gives rise to a new family of test generation techniques that we call MISE (Mutation-Inspired Symbolic Execution), with 'naive MISE' being the basic implementation for the second experiment.

Finally, we study the surviving mutants resulting from the second experiment to identify opportunities for improvement in the test generation process. In this regard, we present three possible implementations of MISE that aim to obtain better results without compromising the already high cost of DSE. Among the three possible implementations, we highlight the approach of modifying the constraints that DSE generates and whose resolution may result in new test cases. For this reason, we carry out an extensive study on how to translate mutation operators implemented in MuCPP to the KQuery language. This study arises as a result of the experiments for the combination of DSE with MT and is presented in Section 6.3.1 below.

The full details of this new family of techniques, the tools and procedures involved, as well as the final results are presented in Chapter 5.

6.3.1 First steps in the constraint modification approach

One of the MISE implementations proposed in Section 5.4 is the modification of symbolic execution constraints. To do this, we focus on the language used by KLEE to express the constraints, which is KQuery. A manual analysis of the results obtained during the study in Chapter 5 reveals an interesting fact. In Section 5.3.2, when we used KLEE to apply DSE on a surviving mutant and it generated a new test that killed the mutant, we could observe that such mutant was directly represented in the KQuery constraint associated to the new test case.

Listing 6.1 shows a fragment of one of the example programs distributed with KLEE. To reach line 3 in this program, the input value given to x must be strictly 0. The KQuery constraint shown in Listing 6.2 (which is a result from applying KLEE on Listing 6.1), is responsible for generating the corresponding value, where we see that the symbolic variable will take the value 0 (Eq 0 ...).

```
1 int get_sign(int x) {
2   if (x == 0)
3     return 0;
4   ...
5 }
```

LISTING 6.1: Fragment of get_sign.c example

```
1 array a[4] : w32 -> w8 = symbolic
2 (query [(Eq 0
3             (ReadLSB w32 0 a))]
4         false)
```

LISTING 6.2: Generated KQuery

It is possible to apply the COI mutation operator negating the condition on line 2, as we can see in Listing 6.3. An application of DSE on that mutant generates a new KQuery query where the mutation operator is directly reflected, shown in Listing 6.4. This KQuery constraint is very similar to the one generated by the original program, but this time it looks for the result of the query negation.

```
1 int get_sign(int x) {
2   if (!(x == 0))
3     return 0;
4   ...
5 }
```

LISTING 6.3: Fragment of mutated get_sign.c example

```
1 array a[4] : w32 -> w8 = symbolic
2 (query [(Eq false
3             (Eq 0
4                 (ReadLSB w32 0 a)))]
5         false)
```

LISTING 6.4: Generated KQuery

As part of this PhD work, motivated by the observations presented in the introduction to this section, we have made some first steps that will serve as a foundation for future work in this line. This section presents a study on how the mutation operators that MuCPP incorporates in C and C++ programs are reflected in KQuery queries.

6.3.2 MuCPP and KQuery: equivalences between mutation operators

The KQuery language is a simple language with a reduced grammar over which KLEE performs a series of simplifications when generating the constraints to obtain the test cases. For example, during the test generation process, KLEE automatically eliminates variables that are not used to generate test data, and replaces complex expressions with simpler expressions that produce the same result. This makes it easier to work with

KLEE and speeds up the test generation process, as fewer constraints are needed to achieve the same results.

This approach proposes to modify the constraints at two possible points: during the process of test generation or on the tests once they have been generated. If the constraints are modified during test generation, automation is facilitated, as it allows KLEE to more easily reach the desired testing goals by providing more specific guidance on the test inputs, although it would be necessary to modify the KLEE source code. By contrast, if the KQuery constraints are modified at the end of test generation, an external process would be necessary to introduce the changes on them and send the modified constraints to the corresponding solver. In either case, the expected result is a set of test cases that can kill more mutants with a single run of KLEE. For this initial study, the modifications are included manually at the end of the process, to verify the feasibility of this proposal. However, future work includes modifying KLEE to include these changes while generating the KQuery constraints.

It can be assumed, to a first approximation, that the implementation of changes in the KQuery constraints should map directly to the behaviour of MuCPP mutation operators. As the KQueries used are generated by KLEE, there are some rules that should be considered:

- Constraints do not include negative values, regardless of the data type.
- No logical negation operators are used. Also, the operations 'greater than' and 'greater than or equal to' will not be included either, being replaced by their analogues 'less than' and 'less than or equal to', respectively.
- Compound operations (+=, -=, *=, etc.) and increment and decrement operations (++,- -) will not be included, being replaced by their elementary operations. For example, if an increment (x++) appears in the C code, in KQuery it will be represented by a sum of the value 1 (x = x + 1).

The simplistic nature of the KQuery language, along with the simplifications that KLEE makes to the constraints, results in some overlap among the changes made by the mutation operators. What in C and C++ were two or more distinct mutation operators, in

the KQuery language can be simplified to one. Table 6.1 shows the equivalences observed in this study between the mutation operators implemented in MuCPP, with the left column being a mutation operator that covers the behaviour of the operators listed in the right column.

Mutation operator	**Sub-operator (only in KQuery)**
ARB	ARS
	ASR
AIU	ARU
AIS	ADS
COI	COD
COR	LOR
ROR	*none*

TABLE 6.1: Mutation operator equivalence in the KQuery language

Each of these operators is shown below with their behaviour in both MuCPP and the KQuery language, as well as an explanation of why certain mutation operators cover the functionality of others.

ROR operator

This operator is the only one that has no equivalence with other operators when applied to the KQuery language. Listing 6.5 shows how this operator works in MuCPP on C++ code, while Listing 6.6 shows the changes that this operator produces in KQuery constraints. To summarise, this mutation operator replaces some relational operators with others.

```
> changes to >=
>= changes to >
< changes to <=
<= changes to <
== changes to !=
!= changes to ==
```

LISTING 6.5: ROR - Changes made by MuCPP (C/C++ programs)

```
Sle changes to Slt
Slt changes to Sle
Eq changes to Ne
```

LISTING 6.6: ROR - Changes observed in KQueries

It is striking, however, that MuCPP is able to make changes in up to six operators, while in the KQuery language this is only reflected in three. In our observations we have found that the operation is exactly the same, since only these three relational operators cover the operation of all of them.

This is the meaning of the operators that appear in Listing 6.6:

- `Sle`: Signed less or equal.
- `Slt`: Signed less than.
- `Eq`: Equal.
- `Ne`: Not equal.

The rest of the relational operators implemented by the KQuery language, according to the `Klee::Expr`[1] documentation, are not used, so we will never find other operators of this type in the constraints that result in the tests. Likewise, the only Boolean operation that can involve a constant is the boolean `not (== false)`, which is reflected in the mutation operation as follows:

- Neither $>$ nor $>=$ operators (`Sgt` and `Sge`, respectively) are implemented. Instead, it performs an optimisation corresponding to the following example:

```
C++ ( a < 10 ) -> KQuery ( Slt a 10 )
C++ ( a > 10 ) -> KQuery ( Slt 10 a )
```

- The != operator (Ne) is not implemented, therefore the Eq operator is negated when it appears (regardless of whether its origin is == or !=).

```
C++ ( a == 10 ) -> KQuery ( Eq a 10 )
C++ ( a != 10 ) -> KQuery ( Eq false ( Eq a 10 ) )
```

The behaviour of this mutation operator is unique and no other mutation operator has been observed to work in a similar way in this context.

[1] https://klee.github.io/doxygen/html/classklee_1_1Expr.html

ARB operator

The ARB mutation operator, as seen in Listings 6.7 and 6.8, has a straightforward application in its transition from C++ to KQuery. The only difference is that, in this case, the KQuery language operators differentiate between signed and unsigned division. Please note that `Rem` refers to the "remainder" operator, which returns the remainder after division of one number by another.

```
+ changes to −
− changes to +
∗ changes to /
/ changes to ∗
Rem changes to /
```

LISTING 6.7: ARB - Changes made by MuCPP (C/C++ programs)

```
Add changes to Sub
Sub changes to Add
Mul changes to SDiv
SDiv changes to Mul
UDiv changes to Mul
SRem changes to SDiv
URem changes to UDiv
```

LISTING 6.8: ARB - Changes observed in KQueries

An example of direct application is shown below.

```
C++ (a = a * 4) => KQuery (Mul 4 a)
```

The operation is simple, but we are dealing with a mutation operator capable of modifying 7 different arithmetic operators. As a consequence of the simplifications made in the KQuery language, this operator covers both of the ARS and ASR mutation operators, which we will call sub-operators in this context. Below is a description of the changes these operators produce and why they are covered by the ARB mutation operator.

ARB sub-operators (ARS and ASR)

Operator	**Description**	**Behaviour**
ARS	Arithmetic Operator Replacement (Short-cut)	Replaces ++ and - -
ASR	Short-cut Assignment Operator Replacement	Replaces compound operators

TABLE 6.2: ARS and ARS behaviour in C/C++

The ARS mutation operator, in C++, swaps the increment (++) and decrement (- -) operators when they appear. These operators are not implemented in KQuery and the following simplification is made instead:

```
C++ (a++) -> KQuery (Add 1 a)
C++ (a--) -> KQuery (Sub 1 a)
```

This means that the increment and decrement operators are converted to arithmetic additions and subtractions with the value 1 in the KQuery language. Therefore, the operation of the ARS mutation operator translates into swapping the arithmetic operators Add and Sub when they appear. This change has already been made by the operator, which covers this case since, in the end, we are adding or subtracting a value to the variable. Similarly, the ASR mutation operator, in C++, swaps compound operators such as -= and +=, making the same swaps as the ARB operator. Since these operators are not supported in the KQuery language, the code is simplified in a similar way as shown below.

```
C++ (a += 10) -> KLEE (Add a 10)
```

In this respect, we have two differentiated situations. The ARS mutation operator can be considered as a subset of the ARB mutation operator, since it only performs two changes out of the seven possible with ARB. The ASR operator, however, has no place in the KQuery language, since its implementation would be exactly the same as that of the ARB operator. In any case, it is concluded that the ARB mutation operator covers the operation of both ARS and ASR.

AIU operator

The AIU mutation operator introduces a negation operator on any positive variable or constant where possible. None of the KQuery constraints generated during the experiments in Chapter 5 had a negative value, so we did a small test to see how this language behaves in this situation. We negated the value of a variable in any function within a condition and the result was as follows:

```
C++ (a == 100) -> KQuery (Eq 100 a)
C++ (-a == 100) -> KQuery (Eq 4294967196 a)
```

The alteration in the constraint is not in the variable `a`, but in the constant 100. Apparently, this value is transformed into another large value, mainly because it is not the `int` type used in C++, but a `32-bit unsigned int` type. By taking the value 4294967196 and transforming it from a `32-bit unsigned int` to an `int`, we get the value -100, which is exactly the value that appears in C++ and is therefore directly reflected in the KQuery constraint.

Therefore, the application of the AIU mutation operator in the KQuery language is a direct transformation of the behaviour of the same operator applied by the mutation tool. To do this, it is sufficient to detect all those situations where variables and constants appear (assignments, conditional statements, etc.) and negate their numerical value, keeping in mind that it is a `32-bit unsigned int`, applying the corresponding data transformation.

```
Original KQuery (Eq 100 a) -> Mutated KQuery (Eq 4294967196 a)
```

AIU sub-operator (ARU)

Operator	**Description**	**Behaviour**
ARU	Arithmetic Operator Replacement (Unary)	Replaces + and -

TABLE 6.3: ARU behaviour in C/C++

In MuCPP, the ARU mutation operator replaces the unary operators + and - when they appear in any variable or constant. Due to simplifications in the KQuery constraints, neither of these arithmetic operators will be replaced, but the variable or constant will simply be negated when it appears. In practice, the negative value (with the - operator) is replaced by a positive value and vice versa. For this reason, the AIU mutation operator in KLEE completely covers the functionality of the ARU mutation operator.

AIS operator

The AIS mutation operator inserts an increment (++) or decrement (- -) operator on those variables where possible. As we have seen in previous examples, the KQuery language does not implement these operators, but they are replaced by an addition or subtraction of the value 1. This does not prevent the AIS mutation operator from having a direct application in KQuery constraints, adding or subtracting 1 to variables within the constraint tree. For example, with `a` being any variable:

```
Original KQuery (a) => Mutated KQuery ((Add 1 a))
Original KQuery (a) => Mutated KQuery ((Sub 1 a))
```

AIS sub-operator (ADS)

Operator	Description	Behaviour
ADS	Arithmetic Operator Deletion (Short-cut)	Removes ++ and - -

TABLE 6.4: ADS behaviour in C/C++

The application of the ADS mutation operator becomes more complex when it comes to introducing it into the KQuery language. This operator, in MuCPP, removes the increment (++) and decrement (- -) operators when they appear. We have seen that the KQuery language does not implement these operators, but they are reflected by an addition or subtraction of the value 1. In this way, it is impossible to differentiate when an increment or decrement is being performed, or adding or subtracting the value 1 in the C++ code. A small test has also been performed, showing that KLEE performs a simplification on the KQuery constraint, as seen below.

```
Original C++ (a == 100) -> Original KQuery (Eq 100 a)
Mutated C++ (++a == 100) -> Mutated KQuery (Eq 99 a)
```

In a global application, where we apply all the operators as in the experiments in Chapter 5, we can consider that the ADS mutation operator is subsumed by the AIS mutation operator. This is because when the value 1 is added or subtracted to an operation using the increment or decrement operator, it is overridden, so it is effectively removed in practice.

COI operator

This mutation operator inserts the corresponding negation operator ('!' in C and C++) into the conditions to negate them. This change also carries over to the KQuery language with ease, as we can insert the 'Not' operator to any condition. Its application is therefore straightforward.

```
Original C++ (a == 100) -> Original KQuery (Eq 100 a)
Mutated C++ (!(a == 100)) -> Mutated KQuery (Not (Eq 100 a))
```

Please note that this change is not exactly the one reflected in the KQuery constraints obtained during the experiments in Chapter 5. Instead, the following modification is introduced:

```
Original C++ (a == 100) -> Original KQuery (Eq 100 a)
Mutated C++ (!(a == 100)) -> Mutated KQuery (Eq false (Eq 100 a))
```

The operation is exactly the same, since negating a condition or forcing the result of a condition to be false makes no difference to the behaviour. Both changes are valid and work well for generating test data, but we consider that the first option (inserting *Not*) leads to a more readable constraint.

COI sub-operator (COD)

Operator	**Description**	**Behaviour**
COD	Conditional Operator Deletion	Removes `!` and `not`

TABLE 6.5: COD behaviour in C/C++

The COD mutation operator, again, is tricky to incorporate into the KQuery language. Simplifications in relational operators lead to confusing situations. The following example shows a situation of a KQuery constraint from any C++ code. This has been manually written for illustrative purposes.

```
C++ ( a < 10 ) -> KQuery ( Slt a 10 )
C++ ( a > 10 ) -> KQuery ( Eq false ( Sle 10 a ))
```

If we just focus on the second KQuery constraint, we have no way of telling whether the original condition was $a > 10$ or $!(a <= 10)$. Although arithmetically it is the same situation, this is not the case in the source code, as it is impossible to know if the negation operator is being used.

Again, in a context such as the experiments in Chapter 5, where we apply all the mutation operators, we can say that the operation of this mutation operator is covered by the COI mutation operator. In this sense, all conditions appearing in the KQuery constraint are negated indiscriminately. As a result, in case the negation operator had been applied in the source code, it would be negated and, in practice, eliminated.

COR operator

```
&& changes to ||
|| changes to &&
Xor changes to ||
```

LISTING 6.9: COR - Changes made by MuCPP (C/C++ programs)

```
And changes to Or
Or changes to And
Xor changes to Or
```

LISTING 6.10: COR - Changes observed in KQueries

Listings 6.9 and 6.10 show that this mutation operator has a straightforward application, so it is only necessary to translate this behaviour to the KQuery language without any additional changes or considerations.

COR sub-operator (LOR)

Operator	**Description**	**Behaviour**
LOR	Logical Operator Replacement	Remplacement of (&, | and ^)

TABLE 6.6: LOR behaviour in C/C++

This mutation operator, in C and C++, replaces the unary logical operators. The KQuery language does not support this type of operators. Instead, these operators are translated as follows:

```
C++ (&) -> KQuery (And)
```

```
C++ (|) -> KQuery (Or)
C++ (^) -> KQuery (Xor)
```

Again, if we only look at the KQuery constraints, we have no way of knowing whether the original code uses the binary or unary logical operators. The implementation of these changes is therefore covered by the COR mutation operator.

6.3.3 Discussion

In this section, we have seen a first approximation of how mutation operators are reflected in the KQuery language and we have defined a number of mutation sub-operators, whose behaviour is fully covered without the need for an explicit implementation. However, some of them only make sense if all traditional mutation operators are applied. From the set of traditional operators, according to our observations, only the ARS mutation operator could be implemented independently, since its operation is a small part of the ARB operator. However, if we wanted to implement an individual version of the rest of the sub-operators, further information would be required. Specifically, the ASR, ARU, ADS and COD mutation operators would need additional information from the original code to know if the changes would take place. The case of the LOR mutation operator is special because it modifies operators nos supported by the KQuery language and, therefore, its operation will always be equivalent to that of the COR mutation operator.

In any case, the application of all mutation operators makes sense in the context of MISE, so these limitations would only exist in the case of focusing the study on a single mutation operator. This study of mutation operators, which is a further result obtained from the work done in Chapter 5, lays the foundation for future implementation during the test generation process with KLEE. The simple nature of the KQuery language and simplifications made during test generation make it easy in most cases to incorporate mutation operators, which are designed with consideration of their behaviour in MuCPP and the changes observed in KQuery constraints resulting from experiments in Chapter 5. An initial contact test reveals that changes are easy to introduce, as only the plain-text constraint needs to be modified. Resolution of the mutated KQuery constraints is trivial for the solvers implemented by KLEE, allowing for quick generation of new test data without the need to reapply DSE on each surviving mutant. Some tests have been

performed and the results are promising, as they allow for the generation of test data that could previously only be obtained by applying DSE on surviving mutants. In this case, only a single run of KLEE is needed, bringing us closer to the goal of generating stronger test data while reducing run time considerably.

Chapter 7

Conclusions and Future Work

"I'm as proud of many of the things we haven't done as the things we have done. Innovation is saying no to a thousand things."

Steve Jobs

This last chapter compiles the conclusions of the lines of research presented in this PhD work. After discussing them, we propose a series of points that could be expanded as future work. Finally, we show the set of scientific publications produced during the doctoral period that complement the work presented in this document.

7.1 Conclusions

The overall goal of this PhD work is to identify the needs of the industry in terms of the development of the software testing stage and to provide a flexible solution to reduce the final cost without compromising the quality of the product. To achieve this goal, we have worked closely with part of the business fabric of the Bay of Cadiz while studying the most up-to-date work on the state of the art. This industrial experience has allowed us to identify the development stages of software testing in this environment, as well as the total cost of applying them. This cost can be considerably reduced by applying test automation techniques, which has motivated us to develop two fundamental elements in the generation of software tests: ASkeleTon, a tool for the generation of test harnesses,

and MISE, a new family of techniques that combines two well-known state-of-the-art techniques to generate test data with a view to detecting possible more potential faults.

The good relationships with the companies in the Bay of Cadiz environment have allowed us to carry out immersion experiences in the manufacturing plants, where we have seen first-hand how the industry works. Thanks to this experience and the first-hand feedback from industrial personnel, we identified four differentiated stages during the testing of industrial software: source code analysis, test harness generation, test data generation and industrial validation. While not much different from what is seen in traditional software development, it is necessary to shift the perspective to large projects, with hundreds of thousands of lines of code and dependencies, so it is common to find several teams working on the different stages. With the exception of the industrial validation stage, which inevitably requires manual intervention by expert personnel, the remaining stages seem automatable from a state-of-the-art perspective. From the automation point of view, it is mainly possible to help teams working in the two intermediate stages, i.e. test harness generation and test data generation. Although source code analysis is an initial stage, we can treat it as a cross-cutting stage in the two stages mentioned above, as we will always need to take into account the original code when generating test cases or test data sets.

As a result of this study, we have developed ASkeleTon, a tool for the complete generation of test harnesses based on the AST. Given a SUT written in C or C++, the output after executing this tool is an initial set of test cases ready for compilation and execution. With the help of industrial feedback, the tool has been designed in a completely modular way, both the modules of the tool itself and the resulting test harnesses. This modularity has several advantages inspired by the needs of industry. As for ASkeleTon, this makes maintenance, revision and extension of the tool a simpler task, adapting to the changing needs of the industry and its projects. It could incorporate, for example, another testing framework, a different code analysis technique or even extend the languages on which test cases can be generated. In terms of the test harnesses, it facilitates teamwork and encourages the reuse and expansion of tests, making it easier to manually incorporate new tests. It also allows the inclusion of new techniques for the generation of test data without the need to modify the source code of ASkeleTon or the test harnesses themselves.

In this PhD thesis we have presented a comprehensive study on the ability of DSE –a

well-known state-of-the-art technique for test generation– to produce test data able to detect bugs in the code following the MT criterion, which measures the quality of the tests with respect to this very purpose. The results show that DSE is not able to kill many mutants, so we set up a second experiment where we combine both techniques, showing that the results improve notably. The implementation of this combination consists in executing DSE on each of the mutants remaining alive, a technique known as *naive MISE*. However, the improvement in results is associated with a considerable increase in cost, as the application of DSE on each of the surviving mutants can take up to an hour to generate the tests. Motivated by this, we propose up to three different lines that try to implement this combination of techniques in a more sophisticated way, giving rise to a new family of techniques that we call MISE. In order to demonstrate the feasibility of these new lines, mutation operators have been defined for the KQuery constraint, the resolution of which gives rise to new test cases. The first results suggest that it is possible to obtain test cases similar to those obtained with *naive MISE* by simply applying the mutation operators to the constraints. In this way, we avoid having to apply DSE on each of the mutants and, therefore, we can reduce the final cost significantly.

Overall, this work makes three important contributions to the literature: a comprehensive industry study based on the state of the art and real industrial experience, a modular, robust and flexible tool that implements a novel process for test harness generation, and a new family of techniques for test data generation. Together, the three elements represent a complete, flexible process for automatic test generation based on industry needs.

The seamless integration of all parts of this study has been severely affected by the emergency situation caused by COVID-19. In addition to the inconvenience to the scientific community as a whole, it has been an obstacle regarding integration with industry. New safety protocols, changes in regulations and the strict confinement have meant that industrial validation and final testing with this kind of projects could not be compatible with the limited time available for the completion of a PhD thesis. Industrial projects, due to their confidential nature, require strict access protocols and can only be accessed in person. Despite all the challenges and limitations mentioned, the research and collaboration with the business community in the Bay of Cadiz has continued. The close collaboration and feedback from industry experts is expected to yield positive results in the future. In fact, this research is expected to be part of a larger project [76] that is specifically designed to address these issues and improve the software testing process in

the industrial environment. Even so, both ASkeleTon and MISE represent advances in their respective research fields, being validated through open source case studies available to everyone.

7.2 Future work

Promising lines of future work generated by this PhD thesis are described below.

Full integration of the presented lines as a whole

In Chapters 4 and 5 we introduce ASkeleTon and MISE, respectively. In their initial state, one technique generates test harnesses and the other one generates test data, but currently there is no interaction between them. Thanks to the modular design of the outputs produced by ASkeleTon, the test data obtained with MISE can easily be used as input for the test harnesses. However, the injection of those test data has to be done manually. Work is underway to integrate the two solutions so that ASkeleTon offers the test data generated by MISE by default, beyond random or manual data. This line not only offers value to test engineers, who will be able to get a set of test harnesses including high quality test data in a fully automated way, but also highlights the convenience and relevance of the modular design of the ASkeleTon tool.

Industrial validation of the complete test generation process

Chapter 3 has described in detail the outcome of the industrial experience. As discussed in Section 7.1, the COVID-19 pandemic has caused a number of difficulties that have prevented the industrial validation of all lines as a whole against current large-scale industrial projects. As the sanitary situation is returning to normal and the constraints produced by the new regulations are addressed, it is planned to integrate the results of this PhD work in an industrial context. In fact, we are currently working on a recently granted proof-of-concept project (ASSENTER) [76], where one of the objectives is to integrate tools like this directly into industry. Thanks to this, it will be possible to study the real benefits and improve the procedure on the basis of the obtained results.

Integration of mutation operators in DSE tools

In Chapter 6, a set of mutation operators has been defined for the KQuery language with the aim of generating test data that will potentially be able to kill more mutants than the original KQuery contraints. These mutation operators are based on the traditional mutation operators implemented by MuCPP. A first manual check allowed us to confirm that it is possible to introduce changes to the KQuery constraints and obtain new results, making the integration of mutation operators feasible. Open source DSE tools such as KLEE are ideal for incorporating these mutation operators automatically into the mechanics of the symbolic execution. A module is being developed to apply the mutation operators during the test case generation process, so that we get a set of test cases capable of killing more mutants in a single DSE run.

This improvement is not only about incorporating these changes into the test case generation process, but also about refining the process. For example, there are cases in which mutation operators generate different constraints that produce the same result (hence equivalent mutants). These are easier to remove than classical equivalent mutants, since, when generating test data, it would only be necessary to discard duplicate data produced by those constraints. This and other difficulties that could arise will be taken into account as the solution is implemented.

Design of mutation operators specific to the KQuery constraint language

In addition to traditional mutation operators, it is common to find in the literature mutation operators designed for specific elements of programming languages (e.g., class mutation operators for C++). The KQuery constraint language, as well as other constraint languages compatible with typical DSE solvers, pose a new challenge for the design and implementation of new mutation operators. In this sense, work is being done on new mutation operators that take into account different characteristics of this language, focusing on its particular notation, its structure or elements that do not fit in more complex programming languages and other KQuery-specific features. The implementation of this new set of mutation operators will make it possible to generate more test cases, capable of detecting more mutants and, therefore, capable of detecting more possible real faults.

Implementation of other ways to apply MISE

The three more sophisticated MISE implementations proposed in this document pose a considerable challenge. Chapter 6 shows the initial development of one of them, as the other two require in-depth study and development, leading to further work on a larger scale. It is envisaged that MISE will be implemented in all three ways proposed in this work, leading to new studies to compare these techniques, to combine them or directly to create new ones based on this experience.

7.3 Publications

The work described in this document has resulted in several scientific publications in different media, which are listed below.

International Journal

- Valle-Gómez, K. J., García-Domínguez, A., Delgado-Pérez, P., & Medina-Bulo, I. (2022). Mutation-inspired symbolic execution for software testing. *IET Software 16(5)*, 478–492 (2022), doi: 10.1049/sfw2.12063.

Book Chapter

- Valle-Gómez, K. J., Delgado-Pérez, P., Medina-Bulo, I., & Magallanes-Fernández, J. F. (2020). La prueba del software como parte esencial en la industria 4.0. In *Diseño, energía y digitalización en proyectos de I D+ i (pp. 162-197)*. Editorial UCA, ISBN: 9788498288438.

International Conference

- Valle-Gómez, K. J., Delgado-Pérez, P., Medina-Bulo, I., & Magallanes-Fernández, J. (2019, May). Software Testing: Cost Reduction in Industry 4.0, *2019 IEEE/ACM 14th International Workshop on Automation of Software Test (AST)*, Montreal, QC, Canada, 2019, pp. 69-70, doi: 10.1109/AST.2019.00018.

National Conferences

- Valle-Gómez, K. J., Delgado-Pérez, P., Medina-Bulo, I., & Garcıa-Dominguez, A. (2022) Incorporación de mutaciones en la Ejecución Simbólica Dinámica. In *Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2022). Goñi Sarriguren, A.(ed.)* Handle: http://hdl.handle.net/11705/JISBD/2022/7537

- Valle-Gómez, K. J., Delgado-Pérez, P., & Medina-Bulo, I. (2021). Técnicas avanzadas para la mejora de la prueba del software. In *Actas de las Jornadas de Investigación Predoctoral en Ingeniería Informática: Proceedings of the Doctoral Consortium in Computer Science (JIPII 2021) (pp. 6-10)*. Universidad de Cádiz.

- Valle-Gómez, K. J., Delgado-Pérez, P., Medina-Bulo, I., & Garcıa-Dominguez, A. (2021). Ejecución Simbólica y Prueba de Mutaciones: mejora de la generación automática de casos de prueba. In *Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2021). Abrahão. Gonzales, S.(ed.)*
  Handle: http://hdl.handle.net/11705/JISBD/2021/040

- Valle-Gómez, K. J., Delgado-Pérez, P., Medina-Bulo, I., & Fernández, J. M. (2019) Reducción de costes en la Industria 4.0 a través de la prueba del software. In *Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2019)*. Handle: http://hdl.handle.net/11705/JISBD/2019/024.

Other publications

During the PhD thesis period, there has been collaboration with directly related projects and other research work. The following publications, although not a product of this PhD work, are those in which the author has participated and are related to the theme of the thesis. This list is not exhaustive and there are additional publications that are not included here.

- Delgado-Pérez, P., Ramírez, A., **Valle-Gómez, K. J.**, Medina-Bulo, I., & Romero, J. R. (2022). InterEvo-TR: Interactive Evolutionary Test Generation with Readability Assessment. *IEEE Transactions on Software Engineering*.
  doi: 10.1109/TSE.2022.3227418.

- Delgado-Pérez, P., Ramírez, A., **Valle-Gómez, K. J.**, Medina-Bulo, I., & Romero, J. R. (2021). Mejora de la legibilidad en la generación de casos de prueba mediante búsqueda interactiva. In *Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2021). Abrahão. Gonzales, S.(ed.)*
  Handle: http://hdl.handle.net/11705/JISBD/2021/022

- Ramírez, A., Delgado-Pérez, P., **Valle-Gómez, K. J.**, Medina-Bulo, I., & Romero, J. R. (2021). Interactivity in the Generation of Test Cases with Evolutionary Computation, *2021 IEEE Congress on Evolutionary Computation (CEC)*, Kraków, Poland, 2021, pp. 2395-2402, doi: 10.1109/CEC45853.2021.9504786.

- Delgado-Pérez, P., Medina-Bulo, I., Álvarez-García, M. Á., & **Valle-Gómez, K. J.** (2021). Mutation Testing and Self/Peer Assessment: Analyzing their Effect on Students in a Software Testing Course, *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, Madrid, ES, 2021, pp. 231-240,
  doi: 10.1109/ICSE-SEET52601.2021.00033.

7.4 Projects

The work carried out in this thesis is part of numerous projects. In particular, it has taken part in a research grant, in a research network and in a large number of projects co-financed by European funds. The following is a list of them in Spanish in order to keep the names of the different governmental organisations.

- Beca para la realización de tesis doctorales en la industria con referencia 2017-083/PU/EPIF-FPI-NAVANTIA/CP, confinanciada por la Universidad de Cádiz y la empresa Navantia.

- Proyecto ASSENTER con referencia PDC2022-133522-I00, de la convocatoria de proyectos de "prueba de concepto" en el marco del Programa Estatal para impulsar la Investigación Científico-Técnica y su Transferencia, del Plan Estatal de Investigación Científica y Técnica y de Innovación 2021-2023, proyecto cofinanciado con fondos FEDER.

- Proyecto AWESOME con referencia PID2021-122215NB-C33, proyecto de generación de conocimiento (PGC2021) en el marco del Programa Estatal para impulsar la Investigación Científico-Técnica y su Transferencia, del Plan Estatal de Investigación Científica, Técnica y de Innovación 2021-2023, proyecto cofinanciado con fondos FEDER.

- Red de Investigación en Ingeniería de Software basada en Búsqueda con referencia RED2018-102472-T, en el marco del Subprograma Estatal de Generación de Conocimiento correspondiente al Programa Estatal de Generación de Conocimiento y Fortalecimiento Científico y Tecnológico del Sistema de I+D+i, proyecto cofinanciado con fondos FEDER.

- Proyecto FAME con referencia RTI2018-093608-B-C33, de la convocatoria de proyectos de "retos de investigación" en el marco del Programa Estatal orietnada a los retos de la sociedad, en el marco del Plan Estatal de Investigación Científica y Técnica y de Innovación 2017-2020, proyecto cofinanciado con fondos FEDER.

- Proyecto DArDOS con referencia TIN2015-65845-C3-3-R, en el marco del Programa Estatal de Investigación, Desarrollo e Innovación orientada a los retos de la sociedad, convocatoria 2015.

Bibliography

[1] Unidad de innovación conjunta Navantia-UCA - Foro Convergia. http://foroconvergia.com/grupo/unidad-de-innovacion-conjunta-navantia-uca/. [Accessed 01-Aug-2022].

[2] KLEE: KLEE::Expr Class Reference. http://formalverification.cs.utah.edu/gklee_doxy/classklee_1_1Expr.html. [Accessed 18-Jul-2022].

[3] R. Ahmadi, K. Jahed, and J. Dingel. mCUTE: A Model-Level Concolic Unit Testing Engine for UML State Machines. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1182–1185, 2019. doi: 10.1109/ASE.2019.00132.

[4] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, & tools*. Pearson Education India, 2007.

[5] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds. An Industrial Evaluation of Unit Test Generation: Finding Real Faults in a Financial Application. In *Proceedings - IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP*, pages 263–272. Institute of Electrical and Electronics Engineers Inc., jun 2017. ISBN 9781538627174. doi: 10.1109/ICSE-SEIP.2017.27.

[6] A. Arcuri. An experience report on applying software testing academic results in industry: we need usable automated test generation. *Empirical Software Engineering*, 23(4):1959–1981, 2018. doi: 10.1007/s10664-017-9570-9.

[7] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018. doi: 10.1145/3182657.

[8] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering*, 41 (5):507–525, 2015. doi: 10.1109/TSE.2014.2372785.

[9] M. Binkhonain and L. Zhao. A review of machine learning algorithms for identification and classification of non-functional requirements. *Expert Systems with Applications: X*, 1:100001, 2019.

[10] C. Boettiger. An introduction to Docker for reproducible research. *ACM SIGOPS Operating Systems Review*, 49(1):71–79, 2015.

[11] C. Cadar and M. Nowack. KLEE symbolic execution engine in 2019. *International Journal on Software Tools for Technology Transfer*, pages 1–4, 2020.

[12] C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, Feb. 2013. ISSN 0001-0782, 1557-7317. doi: 10.1145/2408776.2408795.

[13] C. Cadar, D. Dunbar, D. R. Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[14] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):1–38, 2008.

[15] R. Casamayor, L. Arcega, F. Pérez, and C. Cetina. Bug Localization in Game Software Engineering: Evolving Simulations to Locate Bugs in Software Models of Video Games. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*, MODELS '22, page 356–366, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450394666. doi: 10.1145/3550355.3552440.

[16] CEPSA. Compromiso de transparencia - Fundación Cepsa — fundacion.cepsa.com. https://fundacion.cepsa.com/es/la-fundacion/transparencia. [Accessed 14-Sep-2022].

[17] T. T. Chekam, M. Papadakis, and Y. Le Traon. Mart: A Mutant Generation Tool for LLVM. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 1080–1084. ACM, Association for Computing Machinery. ISBN 9781450355728. doi: 10.1145/3338906.3341180.

[18] T. T. Chekam, M. Papadakis, M. Cordy, and Y. L. Traon. Killing Stubborn Mutants with Symbolic Execution. *ACM Transactions on Software Engineering and Methodology*, 30(2), 2021. ISSN 1049-331X. doi: 10.1145/3425497.

[19] T. Y. Chen, H. Leung, and I. K. Mak. Adaptive random testing. In *Annual Asian Computing Science Conference*, pages 320–329. Springer, 2004.

[20] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque. Pit: a Practical Mutation Testing Tool for Java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 449–452, 2016.

[21] A. Corallo, M. Lazoi, and M. Lezzi. Cybersecurity in the context of industry 4.0: A structured classification of critical assets and business impacts. *Computers in industry*, 114:103165, 2020.

[22] R. D. Craig and S. P. Jaskiel. *Systematic software testing*. Artech house, 2002.

[23] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[24] P. Delgado-Pérez, I. Medina-Bulo, F. Palomo-Lozano, A. García-Domínguez, and J. J. Domínguez-Jiménez. Assessment of class mutation operators for C++ with the MuCPP mutation system. *Information and Software Technology*, 81:169–184, 2017. ISSN 0950-5849. doi: 10.1016/j.infsof.2016.07.002.

[25] R. A. DeMillo. Test Adequacy and Program Mutation. In *Proceedings of the 11th International Conference on Software Engineering*, pages 355–356, Pittsburg, PA, USA, 1989. IEEE Computer Society / ACM Press. doi: 10.1145/74587.74634.

[26] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Program Mutation: A New Approach to Program Testing. *Infotech State of the Art Report, Software Testing*, pages 107–126, 1979.

[27] R. A. DeMillo, A. J. Offutt, and Others. Constraint-Cased Automatic Test Data Generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, 1991.

[28] E. W. Dijkstra. *Chapter I: Notes on Structured Programming.* Academic Press Ltd., GBR, 1972. ISBN 0122005503.

[29] E. B. Duffy, B. A. Malloy, and S. Schaub. Exploiting the Clang AST for analysis of C++ applications. In *Proceedings of the 52nd annual ACM southeast conference*, 2014.

[30] J. W. Duran and S. C. Ntafos. An evaluation of random testing. *IEEE transactions on Software Engineering*, (4):438–444, 1984.

[31] O. S. Dustmann, K. Wehrle, and C. Cadar. PARTI: a multi-interval theory solver for symbolic execution. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 430–440, 2018.

[32] Eclipse IoT Working Group. Open Source Software for Industry 4.0. 2017 (October):18, 2017. URL https://iot.eclipse.org/resources/white-papers/EclipseIoTWhitePaper-OpenSourceSoftwareforIndustry4.0.pdf.

[33] A. Estero-Botaro, A. García-Domínguez, J. J. Domínguez-Jiménez, F. Palomo-Lozano, and I. Medina-Bulo. A framework for genetic test-case generation for WS-BPEL compositions. In *IFIP International Conference on Testing Software and Systems*, pages 1–16. Springer, 2014.

[34] G. Fraser and A. Arcuri. EvoSuite: Automatic test suite generation for object-oriented software. In *SIGSOFT/FSE 2011 - Proceedings of the 19th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 416–419. ACM Press. ISBN 9781450304436. doi: 10.1145/2025113.2025179.

[35] C. Freeman and F. Louçã. *As time goes by: from the industrial revolutions to the information revolution.* Oxford University Press, 2001.

[36] D. Fu, Y. Xu, H. Yu, and B. Yang. WASTK: a weighted abstract syntax tree kernel method for source code plagiarism detection. *Scientific Programming*, 2017, 2017.

[37] A. S. Ghiduk, M. R. Girgis, and M. H. Shehata. Employing Dynamic Symbolic Execution for Equivalent Mutant Detection. *IEEE Access*, 7:163767–163777, 2019. doi: 10.1109/ACCESS.2019.2952246.

[38] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223.

[39] P. Goodman and A. Groce. DeepState: Symbolic unit testing for C and C++. In *NDSS Workshop on Binary Analysis Research*, 2018. doi: 10.14722/ndss.2018.23xxx.

[40] B. J. Grün, D. Schuler, and A. Zeller. The impact of equivalent mutants. In *International Conference on Software Testing, Verification, and Validation Workshops*, pages 192–199, 2009.

[41] M. Harman and P. O'Hearn. From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 1–23. IEEE, 2018.

[42] W. C. Hetzel. *The complete guide to software testing.* John Wiley & Sons, Inc., 1990.

[43] E. Hofmann and M. Rüsch. Industry 4.0 and the current status as well as future prospects on logistics. *Computers in industry*, 89:23–34, 2017.

[44] ISO/IEC/IEEE 29119-1:2022. Software and systems engineering — Software testing. Standard, International Organization for Standardization, Geneva, CH, 2022.

[45] H. Kagermann, W.-D. Lukas, and W. Wahlster. Industrie 4.0: Mit dem Internet der Dinge auf dem Weg zur 4. industriellen Revolution. *VDI nachrichten*, 13(1): 2–3, 2011.

[46] M. E. Khan and F. Khan. A comparative study of white box, black box and grey box testing techniques. *International Journal of Advanced Computer Science and Applications*, 3(6), 2012.

[47] N. Khan, A. Naim, M. R. Hussain, Q. N. Naveed, N. Ahmad, and S. Qamar. The 51 v's of big data: survey, technologies, characteristics, opportunities, issues and challenges. In *Proceedings of the international conference on omni-layer intelligent systems*, pages 19–24, 2019.

[48] M. Khatibsyarbini, M. A. Isa, D. N. Jawawi, and R. Tumeng. Test case prioritization approaches in regression testing: A systematic literature review. *Information and Software Technology*, 93:74–93, 2018.

[49] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976. ISSN 0001-0782. doi: 10.1145/360248.360252.

[50] M. Kintis, M. Papadakis, A. Papadopoulos, E. Valvis, and N. Malevris. Analysing and comparing the effectiveness of mutation testing tools: A manual study. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 147–156. IEEE, 2016.

[51] H. Lasi, P. Fettke, H.-G. Kemper, T. Feld, and M. Hoffmann. Industry 4.0. *Business & information systems engineering*, 6(4):239–242, 2014.

[52] C. Lattner. LLVM and Clang: Next generation compiler technology. In *The BSD conference*, volume 5, pages 1–20, 2008.

[53] N. Leicht, I. Blohm, and J. M. Leimeister. Leveraging the power of the crowd for software testing. *IEEE Software*, 34(2):62–69, 2017.

[54] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang. Fuzzing: State of the Art. *IEEE Transactions on Reliability*, 67(3):1199–1218, 2018. doi: 10.1109/TR.2018.2834476.

[55] Y. Lu. Industry 4.0: A survey on technologies, applications and open research issues. *Journal of industrial information integration*, 6:1–10, 2017.

[56] K.-K. Ma, K. Yit Phang, J. S. Foster, and M. Hicks. Directed symbolic execution. In *International Static Analysis Symposium*, pages 95–111. Springer, 2011.

[57] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg. Manticore: A user-friendly Symbolic Execution Framework for Binaries and Smart Contracts. pages 1186–1189. 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2019.

[58] G. J. Myers, T. Badgett, T. M. Thomas, and C. Sandler. *The art of software testing*, volume 2. Wiley Online Library, 2004.

[59] NAVANTIA. NAVANTIA signs a contract with Saudi Arabia for the construction of 5 corvettes. https://www.navantia.es/en/news/press-releases/

navantia-signs-a-contract-with-saudi-arabia-for-the-construction-of-5-corvettes/, 2018. [Accessed 14-Sep-2022].

[60] NAVANTIA. Calidad y Medio ambiente - Privado: Responsabilidad social corporativa - Navantia. https://www.navantia.es/es/personas/buen-gobierno/calidad-medio-ambiente/, 2022. [Accessed 01-Sep-2022].

[61] NAVANTIA. Transparencia económica - Contratos. https://www.navantia.es/es/transparencia/economica/contratos/, 2022. [Accessed 14-Sep-2022].

[62] S. Nidhra and J. Dondeti. Black box and white box testing techniques-a literature review. *International Journal of Embedded Systems and Applications (IJESA)*, 2 (2):29–50, 2012.

[63] T. Nivas. Test harness and script design principles for automated testing of non-GUI or web based applications. In *Proceedings of the First International Workshop on End-to-End Test Script Engineering*, pages 30–37, 2011.

[64] S. Ognawala, T. Hutzelmann, E. Psallida, and A. Pretschner. Improving Function Coverage with Munch: a Hybrid Fuzzing and Directed Symbolic Execution Approach. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, pages 1475–1482, 2018.

[65] C. Pacheco, S. K. Lahiri, and T. Ball. Finding errors in .NET with feedback-directed random testing. pages 87–95. ISSTA'08: Proceedings of the 2008 International Symposium on Software Testing and Analysis, ACM Press. ISBN 9781605580500. doi: 10.1145/1390630.1390643.

[66] H. Palikareva and C. Cadar. Multi-solver support in symbolic execution. In *International Conference on Computer Aided Verification*, pages 53–68. Springer, 2013.

[67] A. Panichella, J. Campos, and G. Fraser. EvoSuite at the SBST 2020 Tool Competition. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pages 549–552, 2020.

[68] M. Papadakis and N. Malevris. Automatic Mutation Test Case Generation via Dynamic Symbolic Execution. In *IEEE 21st International Symposium on Software Reliability Engineering*, pages 121–130, 2010. doi: 10.1109/ISSRE.2010.38.

[69] M. Papadakis and N. Malevris. Automatically Performing Weak Mutation with the Aid of Symbolic Execution, Concolic Testing and Search-based Testing. *Software Quality Journal*, 19(4):691, 2011. doi: 10.1007/s11219-011-9142-y.

[70] M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon. Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique. In *IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume Volume 1, pages 936–946, 2015. doi: 10.1109/ICSE.2015.103.

[71] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman. Mutation Testing Advances: An Analysis and Survey. volume Volume 112 of *Advances in Computers*, pages 275 – 378. Elsevier), 2019. doi: 10.1016/bs.adcom.2018.03.015.

[72] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff. What is Keeping My Phone Awake? Characterizing and Detecting No-Sleep Energy Bugs in Smartphone Apps. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, page 267–280, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450313018. doi: 10.1145/2307636.2307661.

[73] S. Peacock, L. Deng, J. Dehlinger, and S. Chakraborty. Automatic equivalent mutants classification using abstract syntax tree neural networks. In *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 13–18. IEEE, 2021.

[74] D. Perez and S. Chiba. Cross-language clone detection by learning over abstract syntax trees. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 518–528. IEEE, 2019.

[75] J. Petke, S. O. Haraldsson, M. Harman, W. B. Langdon, D. R. White, and J. R. Woodward. Genetic improvement of software: a comprehensive survey. *IEEE Transactions on Evolutionary Computation*, 22(3):415–432, 2017.

[76] PROYECTO ASSENTER. Referencia: PDC2022-133522-I00 (proyecto base: FAME RTI2018-093608-B-C33). Título: Aplicación de técnicas avanzadas de procesamiento de datos y prueba en la industria. Convocatoria: "Prueba de concepto" en el marco del Plan Estatal de Investigación Científica y Técnica y de Innovación

2021-2023, proyecto cofinanciado con fondos FEDER. Objetivo: persigue alcanzar un TRL6 en productos software del ámbito de las ciudades Inteligente y de la industria y fomentar una mayor implantación de los nuevos avances de software por parte de las empresas. Se llevarán a cabo 2 pruebas de concepto, una de ellas en la empresa Navantia: Automatización de la fase de prueba de software para disminuir el coste total y aumentar la calidad y fiabilidad de sus proyectos software, 2022.

[77] D. M. Rafi, K. R. K. Moses, K. Petersen, and M. V. Mäntylä. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In *2012 7th International Workshop on Automation of Software Test (AST)*, pages 36–42, 2012. doi: 10.1109/IWAST.2012.6228988.

[78] R. Ramler, G. Buchgeher, and C. Klammer. Adapting automated test generation to GUI testing of industry applications. *Information and Software Technology*, 93: 248–263, 2018.

[79] A. Reynolds and V. Kuncak. Induction for SMT solvers. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 80–98. Springer, 2015.

[80] D. S. Rodrigues, M. E. Delamaro, C. G. Corrêa, and F. L. Nunes. Using genetic algorithms in test data generation: a critical systematic mapping. *ACM Computing Surveys (CSUR)*, 51(2):1–23, 2018.

[81] B. Schäling. *The boost C++ libraries*. Boris Schäling, 2011.

[82] K. Schwab. *The fourth industrial revolution*. Currency, 2017.

[83] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés. A survey on metamorphic testing. *IEEE Transactions on software engineering*, 42(9):805–824, 2016.

[84] S. Shamshiri, J. M. Rojas, J. P. Galeotti, N. Walkinshaw, and G. Fraser. How do automatically generated unit tests influence software maintenance? In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 250–261, 2018. doi: 10.1109/ICST.2018.00033.

[85] M. Singh, E. Fuenmayor, E. P. Hinchy, Y. Qiao, N. Murray, and D. Devine. Digital twin: Origin to future. *Applied System Innovation*, 4(2):36, 2021.

[86] B. H. Smith and L. Williams. An empirical evaluation of the MuJava mutation operators. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION)*, pages 193–202, 2007.

[87] UCA. Cátedra Navantia - José Patiño Rosales. https://catedranavantia.uca.es/. [Accessed 01-Aug-2022].

[88] UCA. Cátedra Acerinox. https://catedraacerinox.uca.es/, 2022. [Accessed 01-Aug-2022].

[89] UCA. Cátedra Fundación CEPSA. https://catedrafundacioncepsa.uca.es/, 2022. [Accessed 01-Aug-2022].

[90] K. J. Valle-Gómez, P. Delgado-Pérez, I. Medina-Bulo, and J. Magallanes-Fernández. Reducción de costes en la Industria 4.0 a través de la prueba del software. In *Jornadas de Ingeniería del Software y Bases de Datos (JISBD)*, 2019. URL http://hdl.handle.net/11705/JISBD/2019/024.

[91] K. J. Valle-Gómez, P. Delgado-Pérez, I. Medina-Bulo, and J. Magallanes-Fernández. La prueba del software como parte esencial en la industria 4.0. In A. P. Fernández, editor, *Diseño, energía y digitalización en proyectos de I+D+i*, chapter 4, pages 166–201. Cádiz: Editorial UCA, Valencia: Asociación Española de Dirección e Ingeniería de Proyectos, 2020. ISBN 9788498288438.

[92] K. J. Valle-Gómez, P. Delgado-Pérez, I. Medina-Bulo, and J. Magallanes-Fernández. Software Testing: Cost Reduction in Industry 4.0. In *2019 IEEE/ACM 14th International Workshop on Automation of Software Test (AST)*, pages 69–70, 2019. doi: 10.1109/AST.2019.00018.

[93] S. Vogl, S. Schweikl, G. Fraser, A. Arcuri, J. Campos, and A. Panichella. EVOSUITE at the SBST 2021 Tool Competition. In *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*, pages 28–29. IEEE, 2021.

[94] M. Wang, J. Liang, Y. Chen, Y. Jiang, X. Jiao, H. Liu, X. Zhao, and J. Sun. SAFL: Increasing and Accelerating Testing Coverage with Symbolic Execution and Guided Fuzzing. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, page 61–64. Association for Computing Machinery, 2018. ISBN 9781450356633. doi: 10.1145/3183440.3183494.

[95] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 261–271. IEEE, 2020.

[96] Y. Wang and H. Li. Code completion by modeling flattened abstract syntax trees as graphs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 14015–14023, 2021.

[97] H. Yoshida, G. Li, T. Kamiya, I. Ghosh, S. Rajan, S. Tokumoto, K. Munakata, and T. Uehara. KLOVER: Automatic Test Generation for C and C++ Programs, Using Symbolic Execution. *IEEE Software*, 34(5):30–37, 2017. doi: 10.1109/MS.2017.3571576.

[98] H. Yu, H. Gong, and Y. Wang. Design and implementation of fault injection based on abstract syntax tree of C Program. In *IOP Conference Series: Materials Science and Engineering*, volume 715, page 012034. IOP Publishing, 2020.

[99] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 783–794. IEEE, 2019.

[100] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. De Halleux, and H. Mei. Test Generation via Dynamic Symbolic Execution for Mutation Testing. In *IEEE International Conference on Software Maintenance*, pages 1–10, 2010. doi: 10.1109/ICSM.2010.5609672.

[101] M. Zhivich and R. K. Cunningham. The Real Cost of Software Errors. *IEEE Security & Privacy*, 7(2):87–90, 2009. doi: 10.1109/MSP.2009.56.