



**Project Report on
Mutation Testing**

Authors

Prabin Kumar Shrestha, *pkshrestha@mun.ca*

Daniel Manasseh Ofei Kwafo, *dmokwafo@mun.ca*

ENGI 9839 - SOFTWARE VERIFICATION AND VALIDATION

MASC. IN SOFTWARE ENGINEERING
MEMORIAL UNIVERSITY OF NEWFOUNDLAND

July 2023

Abstract

Mutation testing is a fault-based testing technique that deliberately introduces defects or mutations into a program to assess the effectiveness of the existing test suite in detecting these faults. This project report explores the concept of mutation testing and its applications in software testing across various domains. It delves into the process of automated mutant generation using specialized tools like Stryker.NET for C# code and its role in identifying potential weaknesses in the program's code. The report discusses fundamental hypotheses in mutation testing, including the Competent Programmer Hypothesis (CPH) and the Coupling Effect, which play a crucial role in mutation testing. Additionally, the evaluation criteria of the "mutation score" are introduced, and challenges encountered in mutation testing are discussed, such as computational burden and human effort. The study concludes by emphasizing the importance of mutation testing in enhancing software reliability and quality by identifying and addressing potential faults and defects in diverse testing domains.

1 Introduction

Mutation testing is a fault-based testing technique that intentionally introduces defects or mutations into a program under evaluation. By simulating different types of faults, mutation testing aims to uncover potential weaknesses in the program's code and evaluate the effectiveness of the existing test suite in detecting these faults. This technique has gained maturity over time and is increasingly recognized and utilized in both academic and industrial contexts.

Mutation testing has been proven better than All-Uses Data Flow Criteria. Due to the numerous potential faults present in a program, it is impractical to generate mutants that cover all of them. As a result, the process of selecting only a subset of these faults, particularly those that closely resemble the correct program version, is implemented.

Two fundamental hypotheses in Mutation testing are introduced: the Competent Programmer Hypothesis (CPH) and the Coupling Effect. The CPH suggests that competent programmers tend to produce programs close to correctness, leading to simple and easily correctable faults; thus simple mutants. The Coupling Effect states that test data capable of detecting simple mutants can also effectively identify complex mutants, making the mutation testing practical. The report further explores the process of Mutation testing, where mutants are created by applying syntactic changes to the original code based on specific mutation operators. The goal is to design test cases capable of "killing" the mutants and identifying potential faults.

The evaluation is typically done using a criterion called the "mutation adequacy score," obtained by executing a set of mutated programs known as mutants, which are created by making simple syntactic changes to the original program. It is the ratio of killed mutants to the total number of non-equivalent mutants.

Despite its effectiveness in evaluating test set quality, mutation testing encounters several challenges that hinder its practicality as a testing technique. One notable challenge is the significant computational burden imposed by the execution of a vast number of mutants against a given test set,

making it a resource-intensive process. Additionally, other issues arise in relation to the human effort required for utilizing mutation testing. Several cost reduction techniques, both mutation reduction and execution cost reduction, have been devised to address the challenges associated with mutation testing.

Mutation testing is a flexible and widely acknowledged technique used in software testing across various domains. Originally applied mainly to unit testing, it has evolved to encompass specification, design, integration, and system-level testing. This method has shown successful results in popular programming languages such as C, C++, C#, Java, JavaScript, and Ruby, as well as in specification and modeling languages. Program Mutation, involving the introduction of faults into source code, and Specification Mutation, which inserts faults into program specifications without source code access, are the two main types of mutation testing. At the unit level, Program Mutation generates mutants to represent potential faults within a software unit, while at the integration level, mutants imitate integration faults between different units, also known as Interface Mutation.

In Java and C#, specialized mutation operators were developed to address the challenges posed by their Object-Oriented nature, and in C, a comprehensive set of mutation operators was introduced. Mutation testing has also found applications in formal specifications, web services, and network protocols, enabling the rigorous assessment of the quality and correctness of formal specifications and ensuring the robustness of web-based systems and network protocols. Furthermore, besides evaluating test suite effectiveness, mutation testing has been utilized in software quality assessment, fault detection improvement, test case prioritization, benchmarking test suites, evaluating test oracles, regression testing, safety-critical systems, complementing code coverage metrics, and continuous integration. Overall, mutation testing continues to be an invaluable tool in enhancing software reliability and quality by identifying and addressing potential faults and defects across diverse testing domains.

Mutation testing is a valuable technique for software testing, but manual generation of mutants can be impractical and time-consuming, especially for large applications. To address this, automated mutant generation is essential. We implemented the specialized tool Stryker.NET for C# code. The project applied Test-Driven Development (TDD) and mutation testing to demonstrate its application in .NET projects. Through the use of mutation testing, it identified surviving mutants, highlighting the need for additional test cases to achieve comprehensive test suite coverage and improve the program's effectiveness and reliability.

2 Theory of Mutation Testing

Mutation testing is a powerful and computationally intensive method for assessing the effectiveness of test cases in detecting faults. It requires repeated execution of multiple faulty versions of the program (mutants) under test. Mutation Testing holds the potential to effectively identify suitable test data capable of detecting genuine faults [1].

The All-Uses Data Flow Criteria is a traditional method, used in software testing, to thoroughly validate the data flow within a program. Its purpose is to ensure that all possible paths in the code are tested, which helps detect potential errors and vulnerabilities. This involves checking the flow of

data, including input, processing, and output, across various paths and branches within the software. More information about this criteria, work done by [2], can be found in the work of Mathur and Wang [3].

There exists an extensive array of potential faults for a given program, making it unfeasible to create mutants that encompass all of them. Consequently, conventional Mutation Testing focuses solely on a portion of these faults, specifically those closely resembling the correct program version, under the assumption that they adequately represent all possible faults [4].

2.1 Hypothesis

Jia and Harman [4] mentioned in their paper that traditional mutation testing focuses on a specific subset of faults, namely those that are in close proximity to the correct version of the program. The underlying assumption is that these selected faults will effectively simulate all possible faults. This theory is built upon two fundamental hypotheses: the Competent Programmer Hypothesis (CPH), and the Coupling Effect [5].

2.1.1 Competent Programmer Hypothesis (CPH)

The Competent Programmer Hypothesis suggests that programmers, in general, are competent and tend to create programs that are close to the correct version. This assumption implies that the code produced by competent programmers might contain faults, but these faults are expected to be simple and easily correctable with minor syntactical changes. In mutation testing, various mutations or changes are applied to the original program to create mutant versions. Each mutant represents a specific fault introduced into the original code. However, according to the CPH, only certain types of faults are considered, which are constructed from simple syntactical changes. These syntactical changes are meant to represent the kinds of mistakes that competent programmers are more likely to make [4].

For mutation testing to be practical, three factors are crucial [5]:

1. The set of simple mutants should be small.
2. Errors must be reliably detectable by mutation analysis tools.
3. The issue of equivalence between mutants and the original program should be manageable as a small subproblem.

The validity of the competent programmer hypothesis has been uncertain, with conflicting claims in past research [6]. The researchers have offered a new perspective on the competent programmer hypothesis and its connection to mutation testing. The approach involves attempting to recreate real-world bugs by applying chains of mutations to the code. By analyzing the lengths of these mutation paths, they aim to determine whether the source code is indeed almost correct, or if substantial variations are necessary to induce bugs. They found that the competent programmer hypothesis holds true to some extent. However, it appears that the current set of mutation operators may not be sufficient to generate representative real-world bugs accurately.

2.1.2 Coupling Effect

The program may contain complex errors that are not explicit mutants and cannot be detected by the existing test data. In contrast to the CPH (Code-Programmer Hypothesis) which focuses on a programmer's behavior, the Coupling Effect is concerned with the types of faults utilized in mutation analysis [4]. It states that "Test data on which all simple mutants fail is so sensitive to changes in the program that it is likely that all complex mutants also fail." [5]

Offcut [7] defines simple fault as correctable with a single change to a source statement, while a complex fault requires multiple changes. Mutations introduce simple faults into a program, and higher-order mutants represent complex faults that result from multiple mutations, forming a subset of the overall complex faults present in the program. Offcut states the coupling effect hypothesis as "Complex faults are coupled to simple faults in such a way that a test data set that detects all simple faults in a program will detect a high percentage of the complex faults." Offcut in his paper [7], has introduced Mutation Coupling Effect Hypothesis concept which states "Complex mutants are coupled to simple mutants in such a way that a test data set that detects all simple mutants in a program will detect a large percentage of the complex mutants."

The study [7] focused on comparing the effectiveness of test data designed to detect basic mutations (1-order mutants) with their ability to identify more complex mutations (2-order and 3-order mutants). Surprisingly, the test data developed for basic mutations successfully killed a significant number of 2-order mutants, indicating their effectiveness in detecting more complex faults. The surviving 2-order mutants did not exhibit specific characteristics making them difficult to detect, further supporting the reliability of the test data. Moreover, the test data for basic mutations killed a higher percentage of 2-order mutants compared to specific 2-order test data, and a similar trend was observed with 3-order mutants.

2.2 Process of Mutation Testing

Mutation testing involves generating a set of faulty programs, denoted as mutants p' , from a given program p by making a few syntactic changes to the original code.

For instance, Table 1 shows the creation of mutant p' by changing logical operator, from $\&\&$ (and) to $\|\|$ (or).

Program p	Mutant p'
...	...
$if(a + b > 100 \&\& c > 0)$	$if(a + b > 100 \ \ c > 0)$
$return 1;$	$return 1;$
...	...

Table 1: Example of mutant obtained by changing logical operator

Similarly, another example Table 2 shows that mutant p' is created by changing arithmetic operator, from $+$ (plus) to $-$ (minus).

Program p	Mutant p'
...	...
$if(a + b > 100 \ \&\& \ c > 0)$	$if(a - b > 100 \ \&\& \ c > 0)$
$return \ 1;$	$return \ 1;$
...	...

Table 2: Example of mutant obtained by changing arithmetic operator

The system is equipped with a test set, denoted as T . Prior to initiating the mutation testing, it is essential to ensure the successful execution of this test set against the original program, referred to as p , in order to validate its correctness for the given test case. If any inconsistencies or errors are identified in p , they must be resolved before proceeding to evaluate other mutants. Conversely, if p is deemed to be error-free, each mutant, represented as p' , will be subjected to execution using the same test set T . If the output obtained from executing p' differs from the output obtained from executing p for any test case within T , the mutant p' is classified as "killed". On the other hand, if the obtained results are identical, the mutant is considered to have "survived" [4]. Certain mutants may be considered syntactically invalid, leading to unsuccessful compilation. These non-viable mutants are commonly known as "stillborn" mutants and must be excluded from subsequent analysis [8]. The process is shown in the Figure 1.

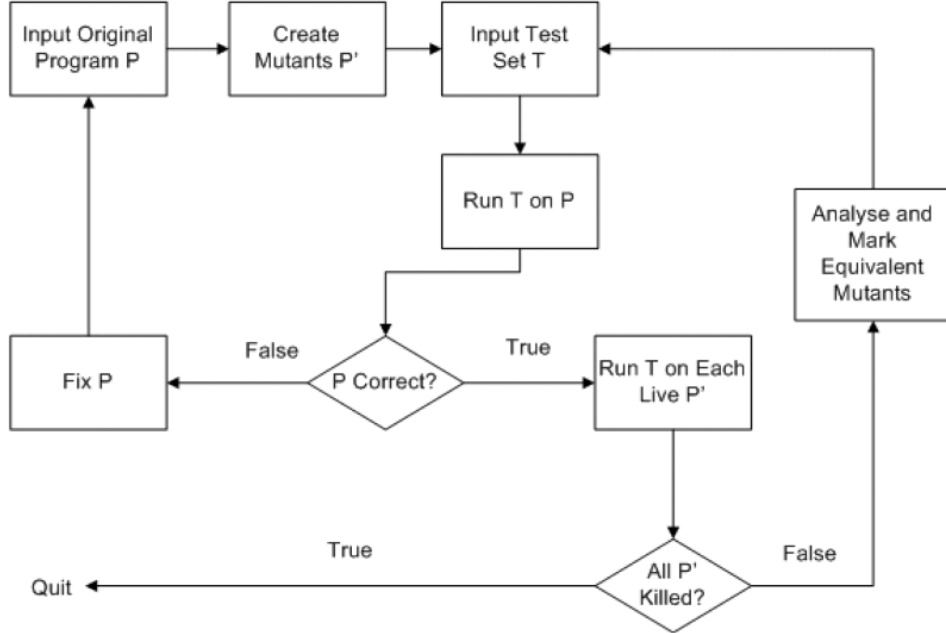


Figure 1: Generic process of mutation testing [4]

Mutation operators are specific rules or transformations that introduce small changes (mutations) in the original code to create mutants. These typical mutation operators are specifically designed to alter variables and expressions through replacement, insertion, or deletion operations [4]. Among

numerous mutation operator, Tables 1 and 2 are two examples of the mutation operator.

Offutt et al. [9] proposed the five-operator set as shown in Table 3. Defining mutant operators is relatively simple as it involves specifying syntactic changes. However, creating useful operators is generally a complex task. Past research has typically taken two approaches: initially defining a comprehensive set of mutant operators based on the language’s grammar, and then through empirical studies, selecting specific subsets to enhance practicality and scalability. Both defining the operators and selecting representative subsets are interconnected, as they involve choosing from an extensive, potentially infinite pool of possibilities. Thus, forming small sets of operators can be seen as a subset selection from the vast range of potential options [8].

Names	Description
ABS	Absolute value insertion
AOR	Arithmetic operator replacement
LCR	Logical connector replacement
ROR	Relational operator replacement
UOI	Unary operator insertion

Table 3: Offutt et al. [9] proposed five operator set [8]

In accordance with the selected set of mutant operators, a set of mutant instances are generated to conduct our analysis. Test objectives revolve around “killing” these mutants, which entails designing test cases capable of detecting and eliminating all the mutants from the program under evaluation [8]. Once all the test cases have been executed, there might still be a small number of surviving mutants. To enhance the effectiveness of the test suite (denoted as T), the program tester has the option to introduce additional test inputs aimed at eliminating these surviving mutants.

Mathur and Wang (1995) [3] discuss a method for decreasing the number of mutants that need to be tested. One prevalent approach for achieving this is by randomly selecting a subset of mutants for evaluation. However, this random selection method might not adequately consider the specific fault detection capabilities of different mutant types. To address this limitation, the researchers proposed a technique called “constrained mutation” or “selective mutation.” This approach involves limiting the generated mutants to two specific types: absolute value mutants (ABS) and mutants obtained by replacing relational operators (ROR). The primary goal of this method is to streamline the testing process, thereby reducing the testing workload, while still maintaining the effectiveness of mutation testing in identifying faults. The researchers put forward two versions of the constrained mutation technique. The first variant entails restricting the mutant operators exclusively to ABS and ROR, while the second variant involves random selection of mutants from each mutant type.

To compare the effectiveness of different testing approaches, the researchers conducted experiments. They specifically compared mutation testing and its variations with the all-uses data flow criteria. In contrast to certain previous studies where human testers created the test sets, Mathur and Wang randomly generated the test sets for their experiments. The results of their experiments confirmed the hypothesis that a testing criterion based on one of the two mutation variants performs better than the all-uses criterion in terms of fault detection effectiveness. Consequently, this suggests that

mutation-based approaches are more adept at identifying faults in software systems compared to the all-uses data flow criteria.

3 Challenges of Mutation Analysis

While Mutation Testing proves to be a valuable method for evaluating the adequacy of a test set, it encounters several challenges. Following are the major problems of mutation analysis [4]:

1. *Computationally Expensive*: Mutation Testing faces a significant hurdle due to its computational complexity. The process demands the execution of a large number of mutants against the test set, leading to considerable computational burden.
2. *Substantial Human Effort*: Mutation testing requires a substantial commitment of human effort. It involves intricate tasks that demand careful attention and meticulous execution from human testers.
 - (a) *Human Oracle Problem*: The procedure of validating the output of the original program against each test case is not exclusive to Mutation Testing; it is a common practice in all forms of software testing. However, Mutation Testing stands out due to its rigorous nature, which involves a greater number of test cases and, consequently, a higher cost associated with conducting these validations (referred to as the oracle cost). This expense often constitutes the most significant aspect of the entire testing process [10].
 - (b) *Equivalent Mutant Problem*: It is important to note that there are certain mutants, known as "Equivalent Mutants," which cannot be eliminated since they always produce identical output to that of the original program. Although these mutants exhibit syntactic differences, they functionally behave the same as the original program. Due to the inherent challenge of determining whether two mutants are equivalent or not, identifying equivalent mutants typically requires additional human effort [8] [11].

While achieving an absolute resolution to these challenges may remain unattainable, noteworthy progress in Mutation Testing has paved the way for automation of the process and improved runtime, enabling a level of practical scalability as demonstrated in this survey. Prior research has predominantly concentrated on devising methodologies to mitigate computational expenses, a subject that will be explored in the subsequent discussion.

4 Cost Reduction Techniques

Mutation Testing is often regarded as a computationally intensive testing method, mainly because of the outdated assumption that all mutants in the set must be thoroughly examined. Nevertheless, to enhance the practicality of Mutation Testing, various approaches have been suggested to decrease its associated costs. Offutt and Untch's research [12] classifies these cost reduction techniques into

three categories: "do fewer," "do faster," and "do smarter." In the study by Jia and Harman [4], these techniques are reorganized into two types: first, reducing the number of generated mutants (termed "do fewer"), and second, minimizing the execution cost (combining "do faster" and "do smarter").

4.1 Reduction of Mutants

Executing a vast number of mutants against the test set is one of the reason why mutation testing requires considerable computation. Consequently, a prominent research focus has emerged on reducing the quantity of generated mutants while maintaining a high level of test effectiveness. Jia and Harman [4] has defined it as: Given a set of mutants M and a set of test data T , the mutation score of the test set T applied to mutants M is denoted as $MS_T(M)$. The mutant reduction problem entails identifying a subset of mutants M' from M , where $MS_T(M') \approx MS_T(M)$.

1. **Mutant Sampling:** Mutant Sampling is a method where a small subset of mutants from the entire set of possible mutants is randomly chosen instead of using all of them for analysis. A certain percentage of mutants, 'x percent,' are randomly selected and use them for the testing process. The remaining mutants are ignored, reducing the computational burden. Different studies [13][14][15] have supported the effectiveness of Mutant Sampling with a higher value of 'x percent.' Alternatively, Sahinoglu and Spafford [16] proposed a different approach to fix the sample rate by using the Bayesian sequential probability ratio test (SPRT). This method randomly selects mutants until a statistically significant sample size is reached. The Bayesian SPRT-based approach is more adaptive and efficient compared to simple random selection, as it adjusts the sample size based on the available test set.
2. **Mutant Clustering:** Mutant Clustering is an alternative approach to Mutation Testing, introduced by Hussain [17]. Unlike random mutant selection, Mutant Clustering employs clustering algorithms to group first order mutants based on their killable test cases. Each cluster contains mutants that can be killed by a similar set of test cases. From each cluster, only a small number of representative mutants are selected for Mutation Testing, while the rest are discarded.
3. **Selective Mutation:** The fundamental concept behind Selective Mutation is that it aims to identify a concise set of mutation operators that can produce a subset of all potential mutants while maintaining high test effectiveness. Mutation operators exhibit varying mutation rates, with some operators generating significantly more mutants than others. Consequently, certain mutation operators may produce redundant mutants that do not contribute substantially to the fault-detection process. Hence, it is feasible to omit such mutation operators from the mutation testing process. Offcutt et al. [9] presented only five mutation operators as shown in Table 3. Likewise, Wong and Mathur [3] employed a streamlined approach, utilizing only two mutation operators: ABS and RAR. Remarkably, their method achieved an impressive 80 percent reduction in the number of mutants generated while experiencing only a minor 5 percent reduction in the mutation score during practical implementation.

4. **Higher Order Mutation:** Mutations can be differentiated into two categories: First Order Mutants (FOMs) and Higher Order Mutants (HOMs). While FOMs arise from single mutation operator applications and represent straightforward changes in the code, HOMs involve complex, multi-step mutations, capturing more subtle defects [4]. The underlying motivation behind HOM lies in the quest to identify those rare yet invaluable higher order mutants that point to latent faults often missed by traditional FOMs. The set of test cases capable of detecting a subsuming HOM includes all the test cases that can identify the FOMs used to construct it. Moreover, the concept of "strongly subsuming HOMs" (SSHOMs) represents an even more challenging class of mutants that can only be detected by a subset of the intersection of test cases capable of killing each contributing FOM. Consequently, SSHOMs exhibit a reduced reliance on test cases while maintaining superior fault-detection capabilities.

4.2 Reduction of Execution Cost

Mutation Testing for Networks presents various techniques aimed at reducing the computational cost associated with the mutation execution process. These optimization strategies play a crucial role in making Mutation Testing more feasible for larger and complex software systems. The following three types of techniques have been explored in the report.

1. **Strong, Weak, and Firm Mutation:** Strong Mutation, also known as traditional Mutation Testing, was originally proposed by DeMillo et al. In this approach, a mutant is considered killed only if it produces a different output from the original program. To optimize the execution of Strong Mutation, Howden [18] introduced Weak Mutation. Weak Mutation focuses on checking mutants immediately after the execution point of the mutated component, rather than the entire program. This reduces the computational cost as each mutant does not require a complete execution process. Firm Mutation, proposed by Woodward and Halewood [19], bridges the gap between Weak and Strong Mutations by providing a continuum of intermediate possibilities in the 'compare state'. However, the availability of publicly available Firm Mutation tools remains limited.
2. **Run-time Optimization Techniques:**
 - (a) *Interpreter-Based Technique:* The traditional Interpreter-Based Technique interprets the result of a mutant directly from its source code. It offers flexibility but can become slower as program scale increases.
 - (b) *Compiler-Based Technique:* The most common approach, where each mutant is compiled into an executable program, providing faster execution than interpretation. However, there is a compilation bottleneck for programs with longer run-times than compilation/link time.
 - (c) *Compiler-Integrated Technique:* Designed to reduce redundant compilation costs, it generates and compiles mutants using an instrumented compiler, reducing individual compilation overhead [20].

(d) *Mutant Schema Generation and Bytecode Translation*: Both techniques aim to reduce compilation costs by generating metaprograms or bytecode mutants directly from compiled object code.

3. **Advanced Platforms Support for Mutation Testing**: Mutation Testing has been extended to advanced computer architectures to distribute the computational cost among multiple processors. Various approaches have been explored for vector processors, SIMD machines, MIMD machines, and network environments. These optimization techniques address the challenge of reducing computational overhead in Mutation Testing, making it more practical and scalable for real-world software systems. While each technique has its advantages and limitations, they collectively contribute to enhancing the efficiency and effectiveness of Mutation Testing processes.

5 Evaluation Metrics

One of the assessment of the test suite's effectiveness is quantified by the mutation score. A mutant that remains unaffected by any test case is considered an equivalent mutant, and mutation score is represented the ratio of killed mutants to the total number of non-equivalent mutants [21].

If K is the total number of killed mutants, N is the total number of mutants, and E is the total number of equivalent mutants, mutation score MS is given by the following equation:

$$MS = \frac{K}{N - E}$$

6 Application of Mutation Testing

Mutation testing is widely recognized for its flexibility, making it applicable across various testing domains. While primarily used in unit testing, advancements have extended its reach to specification, design, integration, and system levels. The method has been successfully applied to popular programming languages including C, C++, C#, Java, JavaScript, and Ruby, as well as specification and modeling languages. For instance, in the design, Mutation Testing has been employed in various contexts such as Finite State Machines. These examples illustrate the diverse range of applications where mutation testing has been successfully utilized.

Since its proposal in the 1970s, Mutation Testing has been utilized to test both program source code, known as Program Mutation [22], and program specifications, referred to as Specification Mutation [23]. Program Mutation falls under the category of white-box testing, where faults are introduced into the source code, while Specification Mutation belongs to black-box testing, where faults are inserted into program specifications without access to the source code during testing.

6.1 Program Mutation

Program Mutation in Mutation Testing has been effectively applied at both the unit level and the integration level of testing. At the unit level, mutants are generated to represent the potential faults that programmers might have made within a software unit. On the other hand, at the integration level, mutants are designed to mimic the integration faults that can arise due to the connection or interaction between different software units [24]. Applying Program Mutation at the integration level is also referred to as Interface Mutation, which was first introduced by Delamaro et al. [25] in 1996.

6.1.1 Mutation Testing for C

Mutation Testing for C has a significant history dating back to 1989 when Agrawal et al. [26] introduced a comprehensive set of 77 mutation operators specifically designed to comply with the ANSI C programming language. These operators were categorized into variable mutation, operator mutation, constant mutation, and statement mutation.

Delamaro et al. [27], further explored the application of Mutation Testing at the integration level. They selected a subset of 10 mutation operators from Agrawal et al.'s set of 77 to test the interfaces of C programs. These operators focused on injecting faults into the signature of public functions, assessing the robustness of the program's interface. In recent times, Higher Order Mutation Testing has also been applied to C programs by Jia and Harman [4], indicating the ongoing evolution and innovation in the field of Mutation Testing for C.

6.1.2 Mutation Testing for Java

Mutation Testing for Java poses unique challenges compared to traditional programming languages due to its Object-Oriented nature. The faults represented by traditional mutation operators do not entirely cover the specific issues that can arise in an OO environment, especially those related to inheritance and polymorphism [28]. As a result, the design of Java mutation operators was not heavily influenced by previous work.

The pioneering work in Java mutation operators was done by Kim et al. [29]. They proposed 20 mutation operators for Java, employing the HAZOP (Hazard and Operability Studies) technique, which is commonly used in safety analysis. These Java mutation operators were classified into six groups, including Types/Variables, Names, Classes/interface declarations, Blocks, Expressions, and others.

Building upon their previous work, Kim et al. [30] introduced Class Mutation, which applies mutation to Java programs with a focus on faults related to OO-specific features. Initially, three mutation operators representing Java OO-features were selected from the 20 Java mutation operators. Later, in 2000, Kim et al. [31] added another 10 mutation operators for Class Mutation, and by 2001, the number of Class mutation operators was extended to 15, classified into four types: polymorphic types, method overloading types, information hiding, and exception handling types [32].

Ma et al. [33], [34] proposed 24 comprehensive Java mutation operators based on empirical evaluation of the effectiveness of all mutation operators. These new mutation operators were classified

into six groups: Information Hiding group, Inheritance group, Polymorphism group, Overloading group, Java Specific Features group, and Common Programming Mistakes group.

6.1.3 Mutation Testing for C#

Mutation Testing for C# involves the extension of mutation operators initially proposed for Java to suit the specific features and characteristics of the C# programming language. Derezinska [35], [36] introduced a set of specialized mutation operators for C#, building upon the concepts from Java mutation operators. These new C# mutation operators were implemented in a mutation testing tool called CREAM.

The extension of Java mutation operators to C# was driven by the need to address the language-specific features and concepts introduced by C#. C# is an object-oriented language similar to Java, but it also includes additional features like properties, delegates, events, and LINQ (Language-Integrated Query). The specialized C# mutation operators introduced by Derezinska likely cover aspects such as inheritance, polymorphism, LINQ expressions, event handling, and other language-specific constructs. These operators aim to represent common faults and programming errors that can occur in C# code.

Empirical results obtained by applying the C# mutation operators using the CREAM tool provide insights into the effectiveness of the test suite in detecting faults specific to C# code. The evaluation may include information on the mutation score, which indicates the ability of the test suite to detect injected faults. Additionally, mutation testing for C# can be applied in various domains, including web development, enterprise applications, game development, and more. Mutation testing in C# can help identify subtle defects and corner cases in complex C# codebases, contributing to enhanced software reliability and quality.[2]

Furthermore, mutation testing for C# can be valuable in security-focused applications, as C# is commonly used in the development of web applications and services. Identifying and addressing security vulnerabilities is crucial, and mutation testing can aid in assessing the robustness of the test suite in detecting security-related faults.

6.2 Specification Mutation

Although initially proposed as a white-box testing technique for implementation-level testing, Mutation Testing has also found applications at the software design level. At the design level, Mutation Testing is commonly referred to as Specification Mutation. In Specification Mutation, faults are injected into a state machine or logic expressions. The aim of Specification Mutation is to identify faults related to missing functions in the implementation or misinterpretation of the specification [37].

6.2.1 Mutation Testing for Formal Specifications

Mutation Testing for Formal Specifications is a specialized area where formal specification languages are subjected to mutation operators to evaluate the effectiveness of test suites in detecting faults and

ensuring the quality of formal specifications. Various forms of formal specifications, such as calculus expressions [23] [38], Finite State Machines (FSM) [39] [40], and Statecharts [41] [42], have been considered for mutation testing.

The application of Mutation Testing to formal specifications allows for rigorous assessment of the quality of specifications, ensuring their correctness and effectiveness in defining system behavior. Researchers have proposed specialized mutation operators for various formal specification languages, expanding the scope of Mutation Testing to diverse domains. Empirical evaluations have been conducted to validate the effectiveness of these mutation operators and provide insights into improving formal specification and testing practices.

6.2.2 Mutation Testing for Web Services

Mutation Testing for Web Services has been explored in various contexts, particularly focusing on XML data models and XML-based language features. Lee and Offutt [43] were pioneers in applying Mutation Testing to Web Services. They introduced an Interaction Specification Model to formalize web component interactions and proposed generic mutation operators to mutate the XML data model based on this specification model. Li and Miller [44] proposed another set of XML schema mutation operators, further enhancing the scope of Mutation Testing for XML data models.

The application of Mutation Testing to Web Services and XML data models offers valuable insights into the correctness and robustness of web-based systems. By injecting faults and evaluating the ability of test suites to detect them, researchers can ensure the reliability of web services and address potential defects in XML data handling and processing.

6.2.3 Mutation Testing for Networks

Mutation Testing for Networks is a critical aspect of ensuring the robustness and reliability of network protocols and systems. Researchers have explored the application of Mutation Testing to various network-related components, including network protocols, intrusion detection signatures, and state-based protocols. Sidhu and Leung [45] investigated the fault coverage of network protocols, paving the way for further research in this domain. These mutation operators aim to evaluate the ability of network protocols to handle different fault scenarios, ensuring their resilience in real-world networking environments. Building on this work, Probert and Guo [46] proposed a set of mutation operators specifically designed to test network protocols.

Vigna et al. [47] applied Mutation Testing to network-based intrusion detection signatures. These signatures are essential for identifying and detecting malicious traffic and potential cyber threats in network communications. The use of Mutation Testing in this context helps assess the effectiveness and coverage of intrusion detection mechanisms.

6.3 Other Testing Applications

In addition to assessing the quality of test sets, Mutation Testing has also been used to support other testing activities, such as software quality assessment, fault detection and many other areas.

1. **Software Quality Assessment:** Mutation testing serves as a rigorous measure of test suite effectiveness. By evaluating how well a test suite can detect injected faults (mutants), developers gain insights into the quality of their testing process. A high mutation score indicates a robust test suite that can identify potential defects in the codebase.
2. **Fault Detection Improvement:** Mutation testing highlights weaknesses in test cases by revealing areas where the test suite fails to identify injected faults. This insight allows developers to focus their efforts on writing additional test cases or improving existing ones to enhance fault detection capabilities. [24]
3. **Test Case Prioritization:** Mutation testing can aid in test case prioritization, where more critical test cases, i.e., those that cover areas prone to faults, are executed earlier in the testing process. This approach allows for more efficient testing, as faults are likely to be detected sooner.
4. **Benchmarking Test Suites:** Mutation testing can be used to benchmark different test suites by comparing their mutation scores. This comparison helps identify which test suites are more effective in detecting faults, making it easier to select the most appropriate test suite for specific projects.
5. **Evaluating Test Oracles:** Mutation testing not only verifies the effectiveness of test cases but also helps evaluate the correctness of test oracles. It ensures that test cases produce the correct outcomes when executed against both the original program and mutants.
6. **Regression Testing:** Mutation testing can be applied to assess the impact of code changes during regression testing. It ensures that new code modifications do not inadvertently introduce faults and that existing test cases continue to detect existing faults.
7. **Safety-Critical Systems:** For safety-critical systems, where software failures can lead to catastrophic consequences, mutation testing is invaluable. It helps validate the effectiveness of test cases to detect potential faults in critical domains such as avionics, medical devices, and nuclear systems.
8. **Complementing Code Coverage Metrics:** Mutation testing complements traditional code coverage metrics like statement coverage and branch coverage. While code coverage indicates how much of the code is executed by test cases, mutation testing evaluates how well the test cases can distinguish correct code behavior from faulty behavior [4]
9. **Continuous Integration:** Mutation testing can be integrated into the continuous integration process, providing early feedback on the quality of code changes. It helps ensure that only high-quality code is integrated into the main codebase.

7 Implementation

The process of manually generating mutants can be extremely time-consuming and impractical, especially when dealing with large and complex applications, as it could result in the creation of thousands of mutants. To overcome this challenge, automated mutant generation is essential, along with the capability to test test suites against these mutants. Specialized tools play a crucial role in achieving this automation effectively. For C# codes, we have chosen to use Stryker Mutator, specifically Stryker.NET, as our preferred tool for automating mutation testing and streamlining the process efficiently. Stryker.NET also provides a Mutation Report, which is highly commendable for its user-friendly interface and the valuable insights it offers.

The project was developed using the Test-Driven Development (TDD) approach, which involves writing comprehensive test cases based on the expected behavior of the code. This ensures that each unit of code is thoroughly tested before implementing the corresponding functionality. Furthermore, mutation testing was employed by re-running the tests to identify and address potential bugs or shortcomings.

The objective of this project was not to create a single application but rather to demonstrate the concept of mutation testing and its application in the context of .NET projects. Two projects were created in .NET Core: one named "MutationApp," which contains the application code, and the other named "MutationApp.Test," which is an NUnit test project. The "MutationApp" consists of a service class called "ShoppingCartService," which intentionally remains incomplete to focus solely on the implementation details required for mutation testing. On the other hand, "MutationApp.Test" contains the testing class for "ShoppingCartService" named "ShoppingCartServiceTest." Initially, only six unit tests were conducted.

The original program should be tested to ensure that all the initial tests pass. As depicted in Figure 2, once the tests passed, mutants were generated using a set of mutant operators, with the default configuration provided by the Stryker Mutator being used in this case. After running the mutation testing tool, six mutants were generated, out of which five were killed and one survived, resulting in a mutation score of 83.33%.

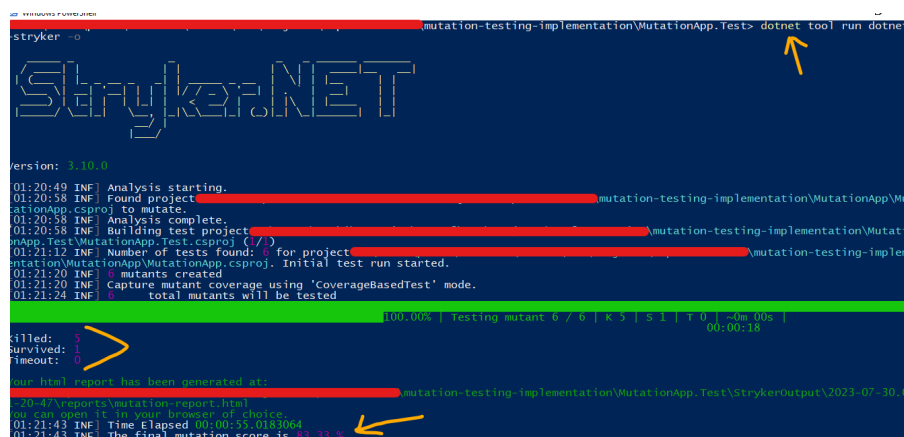
The image is a screenshot of a terminal window showing the execution of Stryker.NET. At the top, the command 'stryker -o' is entered. The Stryker.NET logo is displayed in a large, stylized font. Below the logo, the version '3.10.0' is shown. The terminal output consists of several lines of log messages with timestamps and log levels (INF). Key messages include 'Analysis starting.', 'Found project', 'Analysis complete.', 'Building test project', 'Number of tests found: 6 for project', 'mutants created', 'Capture mutant coverage using 'CoverageBasedTest' mode.', and 'total mutants will be tested'. A progress bar shows '100.00% | Testing mutant 6 / 6 | K 5 | S 1 | F 0 | -4m 00s | 00:00:18'. At the bottom, a summary shows 'Killed: 5', 'Survived: 1', and 'Timeout: 0'. A message states 'Your html report has been generated at: mutation-testing-implementation\MutationApp.Test\StrykerOutput\2023-07-30-01-20-47-reports\mutation-report.html'. The final log message at the bottom states 'The final mutation score is 83.33 %'. Two yellow arrows are present: one pointing to the command line at the top right, and another pointing to the final mutation score at the bottom.

Figure 2: Mutation testing using Stryker.NET tool in unmodified code

The mutation report, shown in Figure 3 indicated that the surviving mutant was produced by changing the specific method "IsDiscountEligible," where \geq was replaced with $>$ using the logical mutation operator.

```

10
17  /* Constructor */
18  public ShoppingCartService() { items = new List<CartItem>(); } ●
19
20  /* Add Item to the List */
21  public void AddItem(CartItem item) => items.Add(item);
22
23  /* Remove Item from the List */
24  public void RemoveItem(CartItem item) => items.Remove(item);
25
26  /* Check if shopping cart is empty */
27  public bool IsEmpty() => items.Count == 0; ●
28
29  /* Check if cart is eligible for discount */
30 - public bool IsDiscountEligible() => items.Count >= DISCOUNT_THRESHOLD_QUANTITY; ● ●
31 + public bool IsDiscountEligible() => items.Count > DISCOUNT_THRESHOLD_QUANTITY;
32
33  /* Calculate and returns the total price without discount */
34  public decimal CalculateTotalPriceWithoutDiscount() => items.Select(item => item.Price * item.Quantity).Sum(); ● ●
35  }
36
37  /* Model Class to represent the Cart Item */
38  public class CartItem
39  {
40      public string Name { get; set; }
41      public decimal Price { get; set; }
42      public int Quantity { get; set; }
43  }
44

```

Figure 3: Mutation report showing the survived mutants

Following the mutation testing, three possible cases could arise:

1. The program is incorrect, in which case it should be corrected.
2. The tests are incomplete, resulting in insufficient test coverage. In this scenario, additional test cases should be added to achieve comprehensive coverage.
3. Equivalent mutants, which do not require any changes, as they do not affect the program's behavior.

In this particular case, it was identified as the second scenario, indicating that the test suite was incomplete and missing boundary conditions. To achieve a mutation score of 100%, a new unit test was added, as shown in the Figure 4. After re-running the tool, all mutants were successfully killed, as illustrated in the Figure 5.

```

[Test]
public void IsDiscountEligible_ShouldReturnTrue_WhenCartItemsIs5()
{
    ShoppingCartService cart = new ShoppingCartService();
    for (int i = 1; i <= ShoppingCartService.DISCOUNT_THRESHOLD_QUANTITY; i++)
    {
        cart.AddItem(new CartItem { Name = $"Item {i}", Price = i * 10, Quantity = 1 });
    }
    bool isEligible = cart.IsDiscountEligible();
    Assert.True(isEligible); // Assert
}

```

Figure 4: Added Unit Test to kill the survived mutant

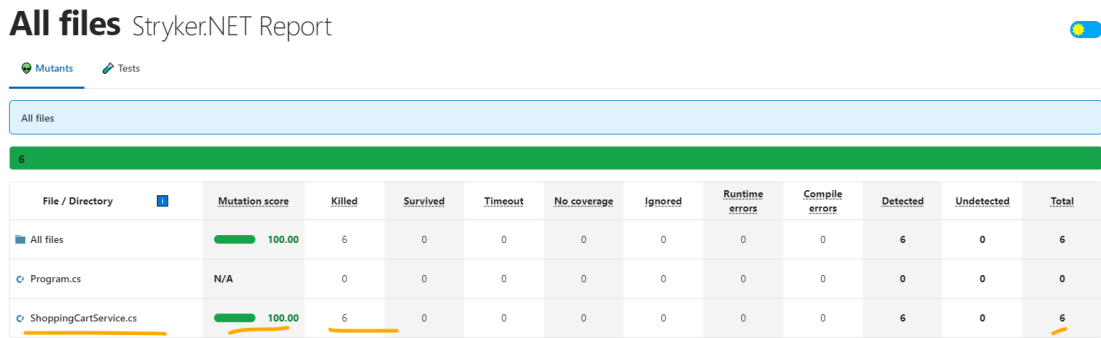


Figure 5: Mutation report showing all mutants killed after the addition of the unit test

From this experiment, it became evident that mutation testing is an effective method for evaluating the quality of test suites and their coverage. If any mutants survive the testing process, it indicates that the above mentioned three cases should be analysed and modified to ensure their effectiveness.

8 Conclusion

In conclusion, mutation testing is a fault-based testing technique that intentionally introduces defects into a program to evaluate the effectiveness of the test suite in detecting these faults. It has proven to be better than All-Uses Data Flow Criteria and is recognized and utilized in both academic and industrial contexts. The evaluation of mutation testing is typically done using a "mutation score," which measures the ratio of killed mutants to the total number of non-equivalent mutants. Despite its effectiveness, mutation testing faces challenges, such as computational burden and human effort, which have led to the development of cost reduction techniques. Overall, mutation testing remains a valuable tool in software testing for various domains, aiding in identifying and addressing potential faults and improving software reliability and quality.

References

- [1] P. Chevalley, "Applying mutation analysis for object-oriented programs using a reflective approach," in *Proceedings Eighth Asia-Pacific Software Engineering Conference*, pp. 267–270, 2001.
- [2] L. Clarke, A. Podgurski, D. Richardson, and S. Zeil, "A formal evaluation of data flow path selection criteria," *IEEE Transactions on Software Engineering*, vol. 15, no. 11, p. 1318–1332, 1989.
- [3] W. E. Wong and A. P. Mathur, "Fault detection effectiveness of mutation and data flow testing," *Software Quality Journal*, vol. 4, no. 1, p. 69–83, 1995.

- [4] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.
- [5] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward *Mutation analysis.*, 1979.
- [6] Z. Ahmed, E. Stein, S. Herbold, F. Trautsch, and J. Grabowski, "A new perspective on the competent programmer hypothesis through the reproduction of bugs with repeated mutations," May 2023.
- [7] A. J. Offutt, "Investigations of the software testing coupling effect," *ACM Transactions on Software Engineering and Methodology*, vol. 1, no. 1, p. 5–20, 1992.
- [8] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman, "Chapter six - mutation testing advances: An analysis and survey," vol. 112 of *Advances in Computers*, pp. 275–378, Elsevier, 2019.
- [9] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 2, p. 99–118, 1996.
- [10] J. S. Bradbury, J. R. Cordy, and J. Dingel, "Exman: A generic and customizable framework for experimental mutation analysis," in *Second Workshop on Mutation Analysis (Mutation 2006 - ISSRE Workshops 2006)*, pp. 4–4, 2006.
- [11] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, "A systematic review of the application and empirical investigation of search-based test case generation," *IEEE Transactions on Software Engineering*, vol. 36, no. 6, pp. 742–762, 2010.
- [12] A. S. Namin and J. H. Andrews, "Finding sufficient mutation operators via variable reduction," in *Second Workshop on Mutation Analysis (Mutation 2006 - ISSRE Workshops 2006)*, pp. 5–5, 2006.
- [13] J. S. Bradbury, J. R. Cordy, and J. Dingel, "Mutation operators for concurrent java (j2se 5.0)," in *Second Workshop on Mutation Analysis (Mutation 2006 - ISSRE Workshops 2006)*, pp. 11–11, 2006.
- [14] A. Mathur and E. Krauser, "Mutant unification for improved vectorization," 1988.
- [15] F. C. Ferrari, J. C. Maldonado, and A. Rashid, "Mutation testing for aspect-oriented programs," in *2008 1st International Conference on Software Testing, Verification, and Validation*, pp. 52–61, 2008.
- [16] "An experimental approach to statistical mutation-based testing."
- [17] S. Hussain, "Mutation clustering," *Ms. Th., Kings College London, Strand, London*, p. 9, 2008.

- [18] W. E. Howden, "Weak mutation testing and completeness of test sets," *IEEE Transactions on Software Engineering*, no. 4, pp. 371–379, 1982.
- [19] M. Woodward and K. Halewood, "From weak to strong, dead or alive? an analysis of some mutation testing issues," in *Workshop on software testing, verification, and analysis*, pp. 152–153, IEEE Computer Society, 1988.
- [20] M. Delamaro, "Proteum-a mutation analysis based testing environmen," *master's thesis, Univ. of São Paulo*, 1993.
- [21] A. Derezinska and K. Kowalski, "Object-oriented mutation applied in common intermediate language programs originated from c#," in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pp. 342–350, 2011.
- [22] R. DeMillo, "Program mutation: An approach to software testing," *Georgia Institute of Technology, Technical Report*, 1983.
- [23] T. A. Budd and A. S. Gopal, "Program testing by specification mutation," *Computer languages*, vol. 10, no. 1, pp. 63–73, 1985.
- [24] A. M. R. Vincenzi, J. C. Maldonado, E. F. Barbosa, and M. E. Delamaro, "Unit and integration testing strategies for c programs using mutation," *Software Testing, Verification and Reliability*, vol. 11, no. 4, pp. 249–268, 2001.
- [25] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur, "Integration testing using interface mutation," in *Proceedings of ISSRE'96: 7th International Symposium on Software Reliability Engineering*, pp. 112–121, IEEE, 1996.
- [26] H. Agrawal, R. A. DeMillo, R. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. Spafford, "Design of mutant operators for the c programming language," tech. rep., Technical Report SERC-TR-41-P, Software Engineering Research Center, Purdue . . . , 1989.
- [27] M. E. Delamaro, J. C. Maldonado, A. Pasquini, and A. P. Mathur, "Interface mutation test adequacy criterion: An empirical evaluation," *Empirical Software Engineering*, vol. 6, pp. 111–142, 2001.
- [28] K. Lakhotia, P. McMinn, and M. Harman, "Automated test data generation for coverage: Haven't we solved this problem yet?," in *2009 Testing: Academic and Industrial Conference-Practice and Research Techniques*, pp. 95–104, IEEE, 2009.
- [29] S. Kim, J. A. Clark, and J. McDermid, "The rigorous generation of java mutation operators using hazop," *Informe técnico, The University of York*, 1999.

- [30] S. Kim, J. A. Clark, and J. A. McDermid, "Assessing test set adequacy for object oriented programs using class mutation," in *Proceedings of the 3rd Symposium on Software Technology (SoST'99)*, Citeseer, 1999.
- [31] S. Kim, J. A. Clark, and J. A. McDermid, "Class mutation: Mutation testing for object-oriented programs," in *Proc. Net. ObjectDays*, pp. 9–12, Net. Objects Erfurt, Germany, 2000.
- [32] S.-W. Kim, J. A. Clark, and J. A. McDermid, "Investigating the effectiveness of object-oriented testing strategies using the mutation method," *Software Testing, Verification and Reliability*, vol. 11, no. 4, pp. 207–225, 2001.
- [33] Y.-S. Ma, Y.-R. Kwon, and J. Offutt, "Inter-class mutation operators for java," in *13th International Symposium on Software Reliability Engineering, 2002. Proceedings.*, pp. 352–363, IEEE, 2002.
- [34] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "Mujava: an automated class mutation system," *Software Testing, Verification and Reliability*, vol. 15, no. 2, pp. 97–133, 2005.
- [35] A. Derezińska, "Advanced mutation operators applicable in c# programs," in *Software Engineering Techniques: Design for Quality*, pp. 283–288, Springer, 2007.
- [36] A. Derezińska, "Quality assessment of mutation operators dedicated for c# programs," in *2006 Sixth International Conference on Quality Software (QSIC'06)*, pp. 227–234, IEEE, 2006.
- [37] V. Okun, *Specification mutation for test generation and analysis*. University of Maryland, Baltimore County, 2004.
- [38] B. K. Aichernig, "Mutation testing in the refinement calculus," *Formal Aspects of Computing*, vol. 15, pp. 280–295, 2003.
- [39] S. P. F. Fabbri, M. E. Delamaro, J. C. Maldonado, and P. C. Masiero, "Mutation analysis testing for finite state machines," in *Proceedings of 1994 IEEE international symposium on software reliability engineering*, pp. 220–229, IEEE, 1994.
- [40] R. M. Hierons and M. G. Merayo, "Mutation testing from probabilistic and stochastic finite state machines," *Journal of Systems and Software*, vol. 82, no. 11, pp. 1804–1818, 2009.
- [41] S. C. P. F. Fabbri, J. C. Maldonado, T. Sugeta, and P. C. Masiero, "Mutation testing applied to validate specifications based on statecharts," in *Proceedings 10th International Symposium on Software Reliability Engineering (Cat. No. PR00443)*, pp. 210–219, IEEE, 1999.
- [42] H. Yoon, B. Choi, and J.-O. Jeon, "Mutation-based inter-class testing," in *Proceedings 1998 Asia Pacific Software Engineering Conference (Cat. No. 98EX240)*, pp. 174–181, IEEE, 1998.
- [43] S. C. Lee and J. Offutt, "Generating test cases for xml-based web component interactions using mutation analysis," in *Proceedings 12th International Symposium on Software Reliability Engineering*, pp. 200–209, IEEE, 2001.

- [44] J. B. Li and J. Miller, “Testing the semantics of w3c xml schema,” in *29th Annual International Computer Software and Applications Conference (COMPSAC’05)*, vol. 1, pp. 443–448, IEEE, 2005.
- [45] D. Sidhu and T.-K. Leung, “Fault coverage of protocol test methods,” in *IEEE INFOCOM’88, Seventh Annual Joint Conference of the IEEE Computer and Communications Societies. Networks: Evolution or Revolution?*, pp. 80–85, IEEE, 1988.
- [46] R. L. Probert and F. Guo, “Mutation testing of protocols: Principles and preliminary experimental results,” in *Proceedings of the Workshop on Protocol Test Systems*, pp. 57–76, 1991.
- [47] G. Vigna, W. Robertson, and D. Balzarotti, “Testing network-based intrusion detection signatures using mutant exploits,” in *Proceedings of the 11th ACM conference on Computer and communications security*, pp. 21–30, 2004.