Mutation Testing of Simulink Models using Large Language Models

by

Jingfan Zhang

Thesis submitted to the University of Ottawa
in partial fulfillment of the requirements for the degree of
Master of Computer Science
in
School of Electrical Engineering and Computer Science
University of Ottawa
Ottawa, Ontario, Canada

Abstract

With the advancements and innovations in the Deep Learning (DL) and the Natural Language Processing (NLP) domain, Large Language Models (LLMs) have been widely applied in various domains including search engines, medicine, finance, and so on. Recently, thanks to the powerful semantic understanding capabilities of LLMs, they have been used to automate and improve software testing activities, including mutation testing — a well-established technique for verifying software. While this effort mainly targets software represented in text-based programming languages like Java and Python, it is unclear how LLMs perform in testing block-based or graphical modeling languages such as Simulink, which is widely used in the Cyber-Physical System (CPS) domain. To analyze Simulink models, one has to either convert the Simulink models into logical representations or employ heuristic algorithms for interpretation, and apply patterns or rules to generate mutations. Therefore, it would be benefitial to find ways to harness the power of LLMs to automatically generate high-quality mutations for Simulink models.

This thesis presents a **BERT**-ass**i**sted approach to **Mu**tant generation for **S**imulink (BERTiMuS) that uses a pre-trained large language model, CodeBERT, to generate mutants for Simulink models. BERTiMuS transforms Simulink models into textual representations by masking specific tokens from the converted text. It utilizes a variant of CodeBERT that has been pre-trained on an extensive corpus derived from Simulink models. This model predicts the tokens that have been masked. Subsequently, mutants are generated by substituting the masked tokens with predictions from the pre-trained CodeBERT. BERTiMuS generates more 'natural' mutants, akin to those generated by humans instead

of by simple patterns, which are less likely to introduce simple errors, such as incorrect variable names or missing blocks, that could directly interrupt the testing program.

We propose two research questions to validate the effectiveness of our approach: (1) Can the mutants generated by BERTiMuS generate test cases that state-of-the-art mutant generation tools for Simulink cannot? (2) How do the mutants generated by BERTiMuS compare with those generated by existing fault patterns for Simulink models? We evaluate BERTiMuS by applying it to public-domain Simulink model benchmarks from the CPS industry with two baseline methods. Our results show that (1) BERTiMuS can select test cases that cannot be selected by previous Simulink mutation methods, indicating that the mutants generated by BERTiMuS are complementary to previous Simulink mutation methods; (2) a large proportion of the mutants generated by BERTiMuS are relevant in terms of requirements violations, which are more meaningful compared to classical mutation testing in terms of real applications; (3) BERTiMuS can automatically generate complex mutants that originally required human effort and knowledge, and BERTiMuS also has certain advantages compared to traditional fault patterns, like expanding expressions and replacing variable names with compatible and alternative names.

Keywords Mutation testing · Simulink models · Large Language Models · CodeBERT

Acknowledgements

If my contributions have made any impact towards bridging Software Engineering and Natural Language Processing, the credit is entirely due to my mentors: Prof. Mehrdad Sabetzadeh, Prof. Shiva Nejati, and our lab member Delaram Ghobari. Their collaborative dialogues, insightful critiques, detailed experiments, and consistent guidance over these years have not only been instrumental in sculpting my research but have also profoundly influenced my personal growth. These professors not only taught me a lot about valuable knowledge of software engineering but also the ability to conduct comprehensive experiments as well as writing skills. Delaram helped me a lot in conducting experiments to test Simulink models.

Lastly, I am immensely grateful to my parents for providing a nurturing environment and sending me to study at the University of Ottawa, which has been vital for my development. Their unwavering support and encouragement have been fundamental in shaping my educational journey and personal growth.

Table of Contents

List of Tables

List of Figures

Abbreviations

AST Abstract Syntax Tree 79

BERT Bidirectional Encoder Representations from Transformers 23, 26–29

CPS Cyber-Physical System ii, iii, 9

DL Deep Learning ii, 1, 79

DNN Deep Neural Network 21

FFN Feed-forward Neural Network 21

FOM First Order Mutant 14, 41

GAs Genetic Algorithms 17–19

GCN Graph Convolutional Network 6

GPT Generative Pre-trained Transformer 23

HOM Higher Order Mutant 14

LLMs Large Language Models ii, 1, 2, 78, 79, 81

MHA Multi-Head Self-Attention 23

MLM Mask Language Modeling 28, 29, 36, 40, 78, 81

MLP multi-layer Perceptron 21

NLP Natural Language Processing ii, 1, 22, 23

NSP Next Sentence Prediction 28

PLMs Pre-trained Language Models 40

RNN Recurrent Neural Network 23, 26, 29

SA Simulated Annealing 17–19

SBSE Search-based Software Engineering 17

SE Software Engineering 2

Chapter 1

Introduction

1.1 Context

The current development of DL (Deep Learning) and NLP (Natural Language Processing) is marked by a series of groundbreaking advancements that have broadened the horizons of computational linguistics and artificial intelligence. Central to this evolution are LLMs (Large Language Models), which have emerged with powerful language understanding abilities and have been widely applied in various fields such as search engines, healthcare,and notably, software engineering.

LLMs are designed based on DL algorithms and NLP techniques, trained on extensive textual corpora. This process equips them with exceptional abilities to understand intricate language patterns and semantics, as well as vast and varied knowledge stored in the parameters of the models. These abilities enable LLMs to excel in specialized tasks such

as predictive text generation and in-depth semantic analysis, applicable across a spectrum of professional domains.

In the field of Software Engineering (SE), LLMs are distinguished by their abilities to interpret and analyze programming languages [13, 15, 16, 18, 41]. These models are trained on extensive code corpora, enabling them to understand programming semantics and structures. Their application in software testing is especially significant. LLMs are utilized to automate the generation and verification of test cases, greatly enhancing the efficiency and accuracy of these processes. The most widely used model, CodeBERT [21], is trained on a substantial dataset that includes both programming and natural languages. CodeBERT is specifically designed for understanding and generating code, making it well-suited for generating meaningful mutations. Its ability to comprehend both natural language and programming language semantics allows for more accurate and contextually relevant changes. CodeBERT has already been successfully applied for mutation testing of code [16,24,34,61].

It is specifically designed to comprehend and generate programming languages. Its capabilities extend to code comprehension, generation, search, and automated documentation, effectively bridging the gap between human and programming languages.

Mutation testing is a technique in software testing that guides the construction of test cases by making small, random changes (mutations) to the program's source code. The central idea of this method is that if the test cases can detect these artificially introduced errors (mutations), then the test cases are considered effective. Thus, mutation testing aids in enhancing the quality and effectiveness of testing in software engineering and is an important component of the field of software testing.

Previous work has made significant progress in improving mutation testing for common programming languages like Java [14,16,32] and C [24,61]. Papadakis et al. [16,34] employ CodeBERT to generate high-quality mutants as well as effective test cases. CodeBERT takes the context of the code as input and produces 'natural' code snippets that are similar to those created by human developers. For instance, compared to pattern-based or rule-based methods [9,23], the code generated by CodeBERT is more likely to change variables to other variables that appeared before, rather than variables that do not exist and thus leading to a compile error. The code is also less likely to cause an infinite loop or an assertion error [16]. Given the successful application of CodeBERT to code, we would like to investigate whether CodeBERT can also be applied to mutation testing of Simulink models if these models are converted into a textual representation.

Many research papers [7,45,47] have also focused on the mutation of Simulink. Simulink is a block-based (graphical) programming language where each block contains code-like structures or expressions, and the blocks are connected in various ways. To analyze Simulink models, we can either translate them into logical representations or use heuristic algorithms. However, these methods typically rely heavily on feature engineering or patterns and cannot effectively capture the complex structure of Simulink models. Moreover, although CodeBERT has been applied successfully with programming languages like Java, it cannot be applied directly to Simulink because Simulink is a graphical programming language, while CodeBERT can only process sequence-format data as input. To adapt the graphical structure of Simulink models for CodeBERT, we transform them into sequence-format data, referred to as Simulink sequences. In detail, we parse the XML file of Simulink models, which contains the textual information of each block, such as block

type and its property value, and the connections of blocks. Blocks that are associated are placed in order. Due to the length limit of CodeBERT's input, we only keep the essential information of blocks, and we gather the textual information of blocks and group blocks under the same system level together to form Simulink sequences. The Simulink sequences contain not only the information of each block but also the information of nearby associated blocks. Since CodeBERT is not pre-trained on Simulink sequences, as we are going to discuss in Section 3.2, we collect 2,611 Simulink models and obtain their sequence corpus to further train CodeBERT. We propose a **BERT**-ass**i**sted approach to **Mu**tant generation for **S**imulink (BERTiMuS).

The approach of utilizing BERTiMuS to conduct mutation testing for Simulink models involves the following process: (1) Parse the XML file of the Simulink models into sequence format, and mask parts of the block's textual information; (2) Input the partially masked sequence into CodeBERT to obtain predictions for the masked tokens; (3) Create mutants by replacing the masked tokens with the predictions; (4) Discard mutants that cannot be compiled.

In this thesis, we propose two research questions to guide our experiments and validate the efficacy of our approach. (1) Can the mutants generated by BERTiMuS generate test cases that state-of-the-art mutant generation tools for Simulink cannot? (2) How do the mutants generated by BERTiMuS compare with those generated by existing fault patterns for Simulink models? The first research question entails a quantitative comparative analysis: we evaluate the test suites chosen to kill the mutants generated by BERTiMuS against those selected for a state-of-the-art mutation testing technique, considering both classical and property-based mutation testing approaches [8]. The sec-

ond question contrasts our technique against a naive baseline: mutations conducted by traditional mutation operators with human effort. Our methodology for killing mutants includes three distinct mechanisms: output-based, fitness-based, and requirement-based assessments. These mechanisms are designed to gauge the impact of mutations on model behavior to varying degrees. Our experimental framework encompasses the evaluation of five Simulink models, where we quantitatively compare the effectiveness of test cases and the killed mutants of our approach against the baseline approach to Simulink mutations. Furthermore, we perform a qualitative comparison to compare the merits and disadvantages of our approach to human-induced mutations. Our experiments indicate that our method surpasses other Simulink mutation techniques in detecting requirement violations, and the mutants of BERTiMuS generated by BERTiMuS are complementary to previous Simulink mutation methods. Moreover, our methodology can automatically generate complex mutants that originally required human effort and software engineering knowledge. Our methodology also exhibits certain advantages compared to traditional mutation operators, like expanding expressions and replacing variable names with compatible and alternative names.

1.2 Challenges

The main challenges in designing BERTiMuS are as follows:

1. The first challenge is how to adapt Simulink models for CodeBERT. We need to design a method to extract essential information from Simulink while making the

information suitable for CodeBERT input. As noted earlier, Simulink models consist of blocks and connections among blocks. Inside each block is the block type and block value, such as expressions or numerals. CodeBERT only accepts sequence-format data as input, such as natural languages and code lines; it cannot directly take a graph as input. Although there are graph neural networks such as Graph Convolutional Network (GCN) [36] that can learn graph structures, and we can view each block as a node, GCN focuses more on the connections between nodes rather than on node representations. However, the information inside blocks of Simulink models is much more important than the information among block connections. CodeBERT also has a more powerful ability to understand text compared to GCN. Therefore, we parse the XML file of Simulink, where blocks under the same system are clustered together; for the blocks of each system, we extract the block type, ID, and property value of the blocks and form them into sequence-format data. The sequence-format data not only contains essential information about the blocks but also groups surrounding and related blocks together to provide context. Although our method does not consider all connections among blocks, the context provides some connection relations among nearby blocks. This sequence-format data can be used as the input of CodeBERT, and we can mask some parts of the data (tokens) to ask CodeBERT to generate new predictions as mutations.

2. The second challenge involves using CodeBERT to generate high-quality mutants for Simulink models. Since CodeBERT is not pre-trained on Simulink sequence-format data, we collect 2,611 Simulink models, convert them into Simulink sequence-format data, and further pre-train it on CodeBERT; we call the newly-trained model

BERTiMuS. This additional pre-training step provides Simulink-related knowledge to CodeBERT and enables it to mutate the sequence-format data of Simulink. Although BERTiMuS can mask and mutate words (usually called tokens in NLP) in text, it cannot expand the original tokens into more tokens. For example, the token 'a=1' might be replaced with three masked tokens, and BERTiMuS would then provide predictions for these three tokens. Therefore, we adopt pre-processing strategies that can expand the expression 'a=1' to 'a=1 && b=3'. We also use post-pre-processing strategies such as applying filters to discard useless predictions.

1.3 Research Contribution

The following research contributions are made through this research:

- To the best of our knowledge, we are the first to treat Simulink models as text for the purpose of mutation testing. To achieve our objective, we first pre-train CodeBERT on a Simulink corpus to equip it with Simulink knowledge. Then, we use the CodeBERT pre-trained on the Simulink corpus to generate mutations for Simulink models. We also propose some pre-processing and post-processing strategies to improve the diversity and quality of the mutants generated by BERTiMuS.

- Existing work [8] only considers one requirement per Simulink model. However, in practice, Simulink models often have multiple requirements, and some mutants may not be relevant to all requirements. Therefore, we use property-based mutation testing [8], where one can attempt to kill mutants in three ways: (1) based on output

differences, (2) based on differences in the fitness values of optimization criteria, and (3) based on requirement violations. These methods exhibit varying degrees of how mutations impact the model, thus offering a more detailed analysis.

- We design comprehensive evaluation strategies to conduct our experiments and validate the effectiveness of our proposed approach. We analyze and compare the mutants generated by BERTiMuS and other approaches, design a workflow to obtain the test cases that can kill mutants, and then compare the test cases and the killed mutants. We also outline some key advantages of the mutants of our approach compared to existing fault patterns for Simulink models.

1.4 Organization

The remainder of this thesis paper is organized as follows: Chapter 2 covers the background concepts that provide the reader with basic knowledge of Simulink, mutation testing, and pre-trained language models. Chapter 3 discusses the main methods of how we process Simulink using CodeBERT, train BERTiMuS, and adopt BERTiMuS to perform mutation testing. We also introduce the mutation operators in our method. Chapter 4 introduces the evaluation strategy of this thesis, discussing the research questions, evaluation setup, study subjects, and evaluation metrics to evaluate the proposed methods in the thesis. Chapter 5 reports on our experimental results, answers the research questions, and compares with baseline methods. Chapter 6 discusses related work in the field of mutation testing for Simulink. Finally, Chapter 7 presents the conclusion, future work, and concluding remarks.

Chapter 2

Background

This chapter presents background on Simulink, mutation testing, and natural language processing basis for language models.

2.1 MATLAB-based Simulink and Stateflow

Simulink [43] is a development and simulation language widely used by the CPS industry to model dynamic systems. CPS are systems that integrate physical and computational components, enabling them to interact and collaborate. They combine aspects of computing, networking, and physical processes to achieve intelligent control, monitoring, and decision-making. Central to Simulink is the block diagram, a graphical representation that serves as the core of system modeling. Block diagrams consist of interconnected blocks drawn from an extensive library, each encapsulating specific mathematical or logical functions. Simulink is commonly used for simulation before developing hardware and deployment

without writing code. Simulink is extensively used for engineering complex systems, such as control systems, signal processing, and communication systems. Its graphical interface and model-based design capabilities make it ideal for multidisciplinary projects in aerospace, automotive, electronics, and industrial automation [29, 30].

The block diagrams of Simulink are characterized by several key attributes, as shown in Figure 2.1:

(1) *Blocks:* Fundamental entities drawn from a large library, encapsulating mathematical, logical, and algorithmic functions. As shown in Figure 2.1, the subsystem contains blocks for comparison and logical operations.

(2) *Interconnections:* Lines connecting blocks represent data, signal, and control flow, capturing dynamic dependencies and forming the basis for simulation. In Figure 2.1, arrows connect the blocks, indicating the flow of signals and data between the 'TL' and 'BL' inputs and their respective outputs 'TLc' and 'BLc'.

(3) *Hierarchical Structure:* Simulink's hierarchical structure allows for the organization of complex system models by breaking them down into smaller subsystems, each containing related blocks. This is illustrated in Figure 2.1, where the main system is broken down into a detailed subsystem, showing how subsystems can encapsulate functionality to enhance model readability, maintainability, and reusability.

Stateflow [44] is a hierarchical state machine language integrated into Simulink, designed for modeling and simulating the event-driven behavior of reactive systems. These systems transition from one operational mode to another in response to various input events and system conditions. Compared to normal Simulink, Stateflow is closer to code.

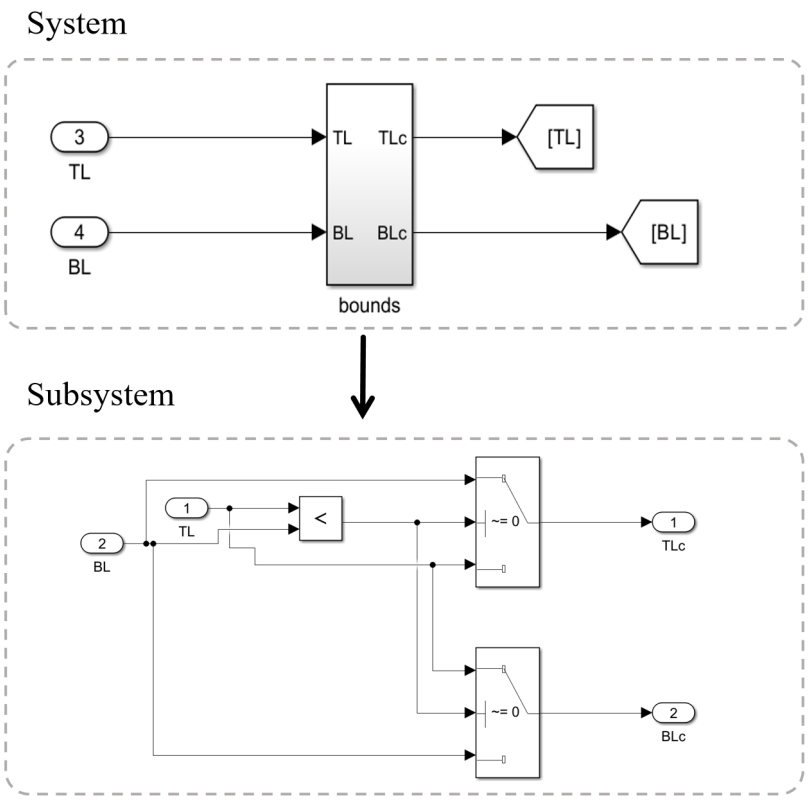
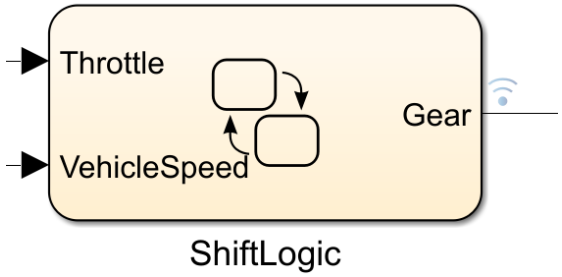
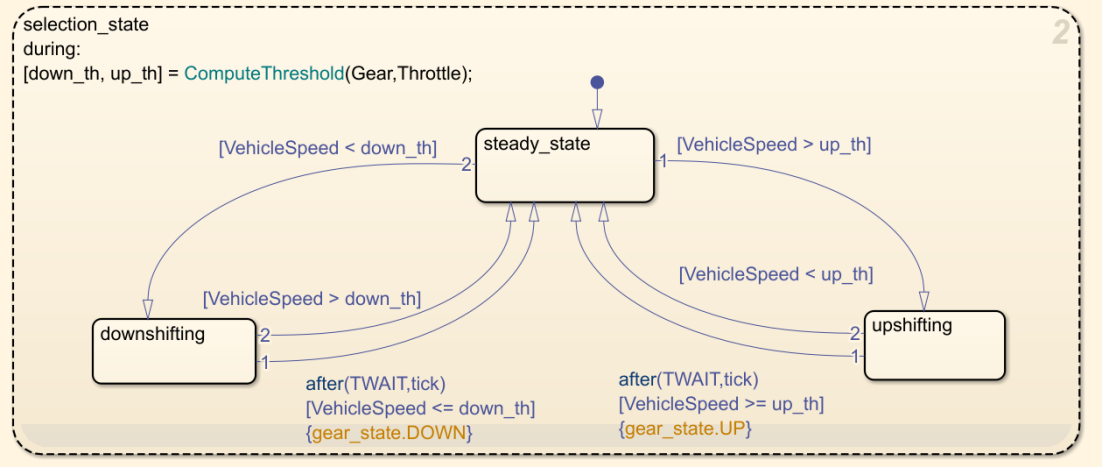


Figure 2.1: Simulink model structure: the figure shows a system-level Simulink model, and the details of a block (subsystem); blocks are connected through arrows, which imply the flows of data, signal, and control.

In Stateflow, these mode shifts are represented as transitions within a transition system in charts. Further, Stateflow facilitates the handling of input and output data, events that can initiate other Stateflow charts, as well as actions and conditions that are linked to specific states and transitions. Figure 2.2 shows an example of a Stateflow model. The ShiftLogic Stateflow subsystem, shown in Figure 2.2(a), is part of an Automatic Transmission Simulink controller. Figure 2.2(b) shows the details of the ShiftLogic Stateflow model consisting of states, transitions, and conditions.



(a) ShiftLogic Stateflow subsystem



(b) ShiftLogic Stateflow model

Figure 2.2: A Stateflow model example: (a) ShitfLogic Stateflow subsystem, and (b) ShitfLogic Stateflow model.

2.2 Mutation Testing

In this section, we introduce the background and fundamental concepts of mutation testing.

Mutation testing [6, 31, 53] is a technique in software testing that modifies a program's source code using predefined operations, known as mutation operators, to create mutants. These mutants are slightly altered versions of the original program, designed to evaluate the effectiveness of existing test cases rather than create new ones. The purpose of mutation testing is to check if the existing test cases can detect these mutants, thereby identifying weaknesses in the test suite. This process highlights gaps in the test suite, guiding the development of more comprehensive test scenarios. The effectiveness of mutation testing relies heavily on the quality of the test suite created. Mutation score provides a measure of test quality and effectiveness, enhancing software reliability and quality assurance.

2.2.1 Basic Concepts

We referenced Bartocci et al. [8] to provide the definitions.

Definition II. 1 (*Mutation Operator*): A mutation operator in software testing is a method used to modify certain aspects of the source code, with the intention of creating faults or variations. These modifications help in evaluating the effectiveness of test cases at identifying errors.

Definition II. 2 (*Test Case/Suite*): A test case is a set of inputs, execution conditions, and expected results developed for a specific objective, such as to exercise a particular program path or verify compliance with a requirement. A test suite is a collection of such

test cases used to validate the behavior of a program or software product. Here, we use $\mathcal{T}$ to denote a test suite, and $t \in \mathcal{T}$ a test case in $\mathcal{T}$.

Definition II. 3 (*Mutant*): A mutant is the altered version of the original program code, generated by applying a mutation operator. For a given program $\mathcal{P}$ and a set $\mathcal{O} = \{op_1, op_2, \ldots, op_n\}$ of mutation operators, a mutant p is the result of applying a mutation operator $op \in \mathcal{O}$ to the program $\mathcal{P}$ at certain locations. A mutant that is created by applying only one mutation operator to $\mathcal{P}$ is called a First Order Mutant (FOM). Applying multiple mutation operators to $\mathcal{P}$ results in a Higher Order Mutant (HOM).

Given a test suite $\mathcal{T}$ and a test case $t \in \mathcal{T}$, we use $t \models p$ to denote when the test passes on p, and $t \not\models p$ to denote when the test fails on p. We denote $O(t, p)$ as the output generated by p with t, and denote $F(t, p)$ as the fitness value generated by p with t.

Definition II. 4 (*Killed Mutant*). A killed mutant is a term used when a test case successfully detects an error in a mutant, implying that the mutant's behavior differs from the original program. This indicates that the test case is effective in revealing faults. A mutant p is said to be killed by $\mathcal{T}$ if some test case $t \in \mathcal{T}$ fails when executing p, i.e., $\exists t \in \mathcal{T} : t \not\models p$.

Definition II. 5 (*Live Mutant*): A live-survived mutant refers to a scenario where a mutant p passes all the test cases in the test suite $\mathcal{T}$, suggesting that the testing was unable to detect the introduced change. It either indicates a need for more robust testing or that the mutation does not impact the program's external behavior. Formally, p is said to be live if $\forall t \in \mathcal{T}, t \models p$.

Definition II. 6 (*Equivalent Mutant*): An equivalent mutant is a mutant whose behav-

ior is indistinguishable from the original program, despite the code changes. This happens when the mutation does not affect the program's output, making it impossible for any test case to detect it as faulty. More precisely, a mutant p is viewed as equivalent if the original program $\mathcal{P}$ and the mutant p generate the same output for any possible input. This definition implies that no matter how extensive the test suite $\mathcal{T}$ is, it might not be able to distinguish between the mutant p and the original program $\mathcal{P}$, highlighting the challenge in detecting equivalent mutants due to the potential incompleteness of any given test suite.

Definition II. 7 (*Invalid Mutant*): An invalid mutant is a mutated version of the program that is syntactically incorrect or results in unexecutable code. Such mutants are usually discarded in the mutation testing process.

Definition II. 8 (*Redundant Mutant*): A redundant mutant is one that, while different from the original program, does not bring new information to the testing process because its behavior has already been tested by other mutants. Identifying and eliminating redundant mutants can make mutation testing more efficient.

To determine whether a test case $t \in \mathcal{T}$ can kill a mutant p, the following conditions must be satisfied [49, 50]: 1) Reachability: t has to reach the mutated statement in p. 2) Necessity: t must impact the program state by affecting the program states for p and $\mathcal{P}$. 3) Sufficiency: The incorrect program states must affect the output of p and be detected by t, i.e., observable differences between the outputs of p and $\mathcal{P}$ for t.

Definition II. 9 (*Mutation Score*): Mutation score is a metric used to assess the effectiveness of a test suite by measuring the percentage of mutants that are killed. High

mutation score indicates that the test suite is robust and capable of detecting a wide range of faults. It is computed using the formula:

$$Mutation\ score = \frac{\#Kill}{\#Valid - \#Equivalent}$$

where $\#Kill$ denotes the number of killed mutants, $\#Valid$ denotes the number of valid mutants, $\#Equivalent$ denotes the number of equivalent mutants.

2.2.2 Fitness Functions

Fitness functions are quantitative measures employed to evaluate how close a given design solution is to achieve set objectives. These functions play a pivotal role in various computational algorithms, especially in the fields of optimization and evolutionary computing. Their primary purpose is to guide algorithms towards more optimal solutions by providing a method to assess the quality or "fitness" of potential solutions against predefined criteria.

The definition of a fitness function is inherently problem-specific, often tailored to the particular goals of the system under consideration. For example, in engineering design optimization, a fitness function might quantify aspects such as efficiency, cost, weight, or performance characteristics. In machine learning applications, it might measure accuracy, error rates, or predictive capabilities.

Computationally, a fitness function takes a candidate solution as input and produces a fitness score as output. This score is a numerical value representing how well the solution meets the objectives. The calculation methodology varies based on the problem domain

but generally involves assessing key performance indicators relevant to the specific goals.

In terms of applications, fitness functions are extensively used in Genetic Algorithms (GAs) [20, 56], Simulated Annealing (SA) [22, 46], and other heuristic or metaheuristic optimization techniques. In these contexts, the fitness function assesses each potential solution generated by the algorithm, guiding the search process towards regions in the solution space where the most promising solutions are likely to be found. This is crucial for problems where traditional optimization methods are inefficient or infeasible due to the complexity of the solution space or the nature of the objective functions.

2.2.3 Mutations in Simulink

The alterations that can be made to a Simulink model are mainly in two ways:

1) Line mutations: changing the behavior of the signals that go through lines from one block to other blocks.

2) Block mutations: changing the behavior of a block, for example, by making changes to the value of a property in a block.

2.3 Search-based Software Testing

Search-based software testing is a sub-field of Search-based Software Engineering (SBSE). SBSE applies search algorithms, such as GAs and SA, to solve optimization problems in software engineering. It explores the potential solution space of software engineering tasks in an automated manner to find optimal or near-optimal solutions.

Search-based software testing focuses on the automatic generation and optimization of test cases using search techniques. Employing algorithms like GAs [20,56] and SA [22,46], search-based software testing aims to maximize code coverage and detect faults efficiently. This approach has been shown to significantly enhance the effectiveness of the testing process by navigating through the vast search space of possible test inputs and configurations.

In addition to maximizing code coverage and fault detection, search-based software testing aims to enhance the overall quality and reliability of software by identifying test suites that can reveal requirement violations or have a significant impact on the requirements such as changing the fitness values of requirements. This is particularly important in complex systems modeled in environments like Simulink, where the interaction between components can be intricate, and the potential for requirements violations is substantial. By selecting optimal test suites, search-based software testing not only ensures that the software behaves as intended under a wide range of conditions but also aids in the early detection of issues, facilitating a more efficient development cycle and contributing to the delivery of high-quality software products.

GAs [20, 56] and SA [22, 46] are among the most prominent techniques used within search-based software testing. Genetic Algorithms are inspired by the process of natural selection, where the fittest individuals are chosen to reproduce, leading to the generation of offspring that are potentially more adapted to their environment. In the context of search-based software testing, GAs can be applied to evolve test suites over successive generations, with each generation of test cases being evaluated based on predefined objectives such as code coverage, fault detection, or execution time. Through selection, crossover, and mutation operations, GAs search the space of possible test cases to find those that best

meet the testing objectives.

Simulated Annealing, on the other hand, is inspired by the physical process of heating and then slowly cooling a material to decrease defects, thus minimizing the system's energy. In search-based software testing, SA is used to iteratively explore the space of test cases, gradually improving the quality of the test suite by making small, random changes and accepting these changes based on a probability that decreases over time. This approach helps in escaping local optima and moving towards a globally optimal set of test cases that maximize the testing objectives.

The integration of techniques such as GAs and SA into the testing process allows for a more automated and objective-driven approach to selecting and evaluating test suites. By employing these algorithms, testers can automatically generate test suites that are not only comprehensive in terms of code coverage but also effective in uncovering subtle, high-impact faults that might not be detected through manual testing methods. Moreover, these algorithms can optimize various aspects of the testing process, such as reducing the redundancy of test cases, balancing the trade-offs between different testing objectives (e.g., thoroughness vs. execution time), and adapting the testing effort to focus on the most critical components of the system under test.

The searching approach is particularly effective in dealing with complex testing scenarios, offering a more efficient and comprehensive testing solution compared to traditional manual methods.

2.4 Language Models

2.4.1 Statistical Language Models

Statistical language models are probabilistic models that calculate the probability of a sentence based on statistical and mathematical models, typically constructed from a corpus. Consider a sentence W composed of T words: $W = (w_1, w_2, \ldots, w_T)$. The joint probability $p(W) = p(w_1, w_2, \ldots, w_T)$ is known as the language model.

According to Bayes' theorem, the above can be represented as:

$$p(W) = p(w_1)p(w_2|w_1)\ldots p(w_T|w_1, w_2, \ldots, w_{T-1}) \tag{2.1}$$

Each term of conditional probability in the above formula represents a parameter of the language model. If these parameters are obtained, then given a sentence W, the corresponding probability of that sentence can be computed. The primary statistical and mathematical models for calculating these parameters include n-gram models, decision tree models, and neural networks. Training language models often involves predicting the T^{th} word from the previous $T-1$ words in a sentence, allowing the model to learn the semantic relationships between words. Once the language model is trained, it can obtain the vector representation of sentences or words, and we can use it for some NLP downstream tasks. The approach based on neural network models is referred to as neural network language models.

2.4.2 Feed-forward Neural Network Language Model

A Feed-forward Neural Network (FFN) consists of multiple neurons and employs a multi-layer Perceptron (MLP) structure where the propagation between neurons is calculated using logistic regression. Logistic regression's formula comprises two parts: linear output and non-linear mapping. The linear output is generated by a linear function $y = kx + b$, while the non-linear mapping often involves a non-linear function, referred to as the activation function in neural networks. A common function used is the hyperbolic tangent (tanh) function. The presence of the activation function makes the neural network model more flexible, capable of solving problems that are not linearly separable.

In neural networks, there can be many layers, where the output of one layer serves as the input for the next. When the number of layers in a neural network is substantial, it is termed a Deep Neural Network (DNN). For instance, a three-layer neural network is structured as follows:

$$z_1^{[1]} = W_1^{[1]}x+b_1^{[1]}, a_1^{[1]} = g(z_1^{[1]})z_2^{[1]} = W_2^{[1]}a_1^{[1]}+b_2^{[1]}, a_2^{[1]} = g(z_2^{[1]})z_3^{[1]} = W_3^{[1]}a_2^{[1]}+b_3^{[1]}, a_3^{[1]} = g(z_3^{[1]}) \tag{2.2}$$

In this structure, the superscript denotes the layer of the neural network, while the subscript represents the sequence index of neurons. The input vector x has dimensions [m, nx], where m is the number of input samples or neurons, and nx is the dimension of x. W represents the weight matrix, with dimensions [nx, hidden_size]. By multiplying W and x, the dimension of vector x is transformed from nx to hidden_size, where hidden_size is

referred to as the hidden layer dimension. b denotes the bias term, and W, b are parameters of the neural network model.

For text data, to obtain the initial vector x, one-hot encoding is used to convert sentences into vectors. In an NLP task, a vocabulary V is constructed, such as V = [I, love, natural, language, processing]. Then, the one-hot encoded vector for a word corresponds to a 1 at its position in the vocabulary, e.g., the vector for 'love' would be [0, 1, 0, 0, 0]. In text sequences, the number of input samples m is the number of words in the sentence, with each word corresponding to a vector of dimension hidden_size.

Bengio et al. [10] proposed a feedforward neural probabilistic language model in 2003, which uses a neural network model to accomplish the tasks of a language model. Their approach involves converting each token in a sentence into a one-hot vector and inputting it into the neural network model. This allows for better learning of the semantic relationships between words through matrix operations.

2.4.3 Attention Mechanism

Attention mechanism [5] has revolutionized the field of NLP by providing an efficient way to process and understand large sequences of data. The most commonly used attention mechanism is self-attention [60], which allows each position in the sequence to attend to all positions in the previous layer of the model, thus capturing intricate and long-range dependencies in text. Mathematically, the self-attention mechanism can be expressed as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{2.3}$$

Here, Q, K, and V represent the query, key, and value matrices, respectively, derived from the input. The term d_k denotes the dimension of the key. The softmax function is applied to the scaled dot product of Q and K^T (the transpose of K), which then multiplies with V to produce the output. This allows the model to weigh the importance of different words in the sequence, enabling it to focus more on relevant words and less on irrelevant ones. The computing paradigm of self-attention is shown in Figure 2.3.

In practice, Bidirectional Encoder Representations from Transformers (BERT) [19] uses Multi-Head Self-Attention (MHA) [60] mechanism, which is shown in Figure 2.4. MHA applies self-attention multiple times in parallel, allowing the model to jointly attend to information from different representation subspaces at different positions. This mechanism enhances the model's ability to capture various aspects of the data by processing the inputs with multiple independent attention heads and then concatenating their outputs.

Multi-head attention language models have been foundational in developing state-of-the-art models like Transformer [60], BERT [19], Generative Pre-trained Transformer (GPT) [54], and their variants. These models have achieved remarkable success in various NLP tasks such as text classification, machine translation, and question-answering by effectively capturing the context and relationships within text sequences.

The flexibility and efficiency of self-attention make it particularly powerful in handling long sequences, as it eschews the sequential processing inherent in traditional Recurrent Neural Network (RNN)s [28], leading to significant improvements in both performance and parallelization capabilities. This paradigm shift towards self-attention models has opened new avenues in NLP research, pushing the boundaries of what machines can understand

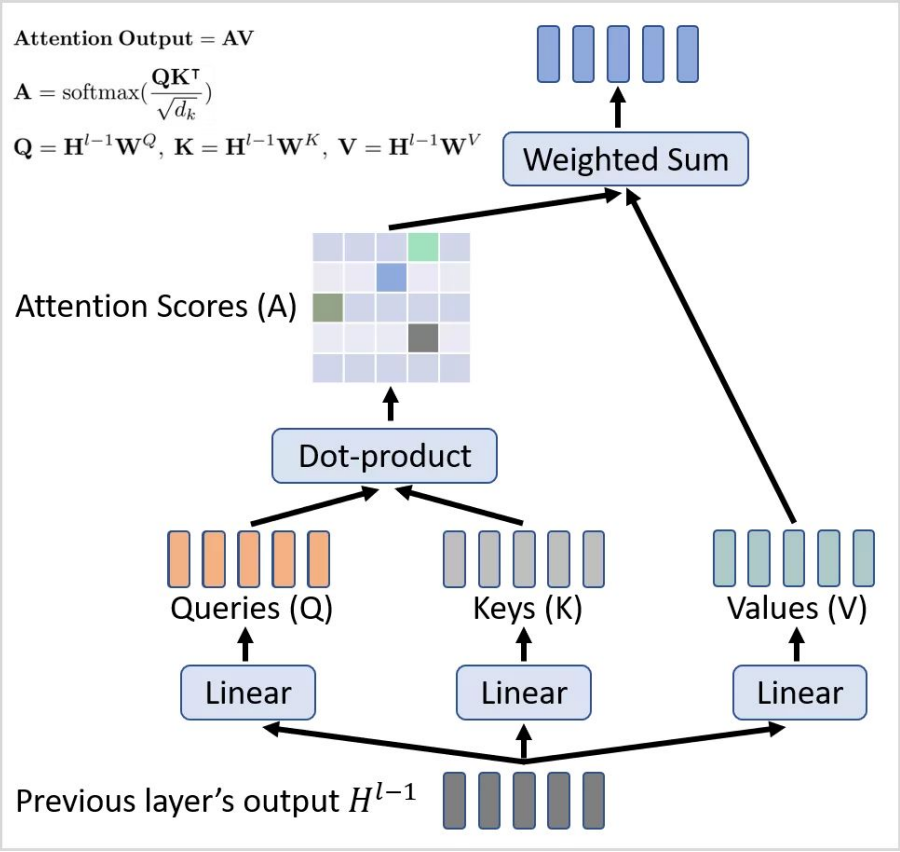


Figure 2.3: Self-attention computing paradigm: Starting with the output of the previous layers H_{l-1}, the queries Q, keys K, and values V are computed through linear transformations. The attention scores A are then derived by performing a dot product between the queries and keys, followed by a Softmax function to obtain a probability distribution. These scores act as weights, which are multiplied with the corresponding values to produce a weighted sum. This weighted sum effectively captures the relevant information from the entire input sequence, weighted by their importance. This figure is originally from [60].

and generate in terms of human language.

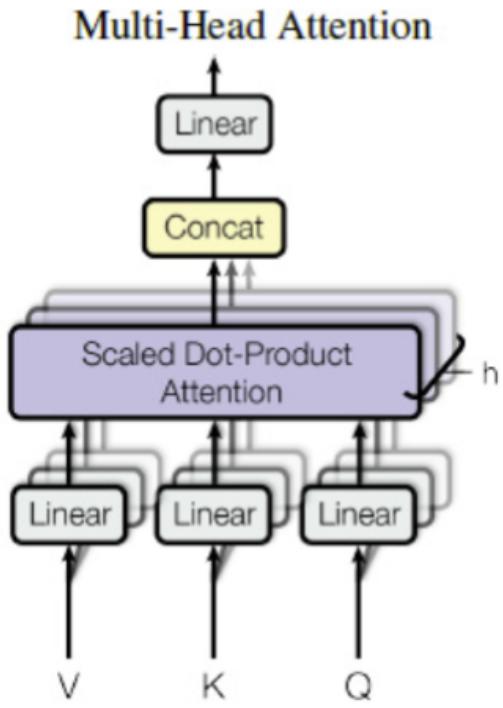


Figure 2.4: Multi-head Self-attention: Multi-head attention splits the input into multiple subspaces, allowing the model to focus on different aspects of the data simultaneously. Each head performs scaled dot-product attention independently, generating attention-weighted representations. These are then concatenated and linearly transformed to produce the final output, enabling the capture of diverse contextual information within sequences. This figure is originally from [60].

The formula for multi-head attention can be represented as:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \text{head}_2, ..., \text{head}_h)W^O \tag{2.4}$$

, where each head_i is an independent attention operation, defined as:

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{2.5}$$

2.4.4 BERT

BERT [19] is a bidirectional language model that employs the encoder portion of the transformer model framework. Due to its large number of parameters, BERT is considered a large language model. The input to the BERT model consists of three components: token embedding, segment embedding, and position embedding. Embedding refers to the vector representation of an object. The token embedding is obtained through one-hot encoding. The role of the position embedding is to inject the sequential information of words in a sentence into the BERT model. Unlike RNN, which transmits information from left to right and inherently contains sequential information, the parallel mechanism of attention lacks the ability to capture sequential information. Therefore, we need to add position encoding to allow the BERT model to learn it. The initialization method assigns values from 0 to the maximum sequence length in order for each token. The function of the segment embedding is to help BERT distinguish between multiple sentences, which is often necessary in sentence relation tasks, such as text semantic matching. Different sentences are distinguished using 0 and 1. The [CLS] token, known as the classifier, is added at the beginning of a sentence, and its corresponding vector can be used to represent the entire sentence vector. The [SEP] token, known as the separator, is used to separate multiple sentences. In cases where there is only one sentence, the corresponding token is represented

entirely by 0. In token embedding, the numbers represent the index of a given token in the vocabulary, used for one-hot encoding.

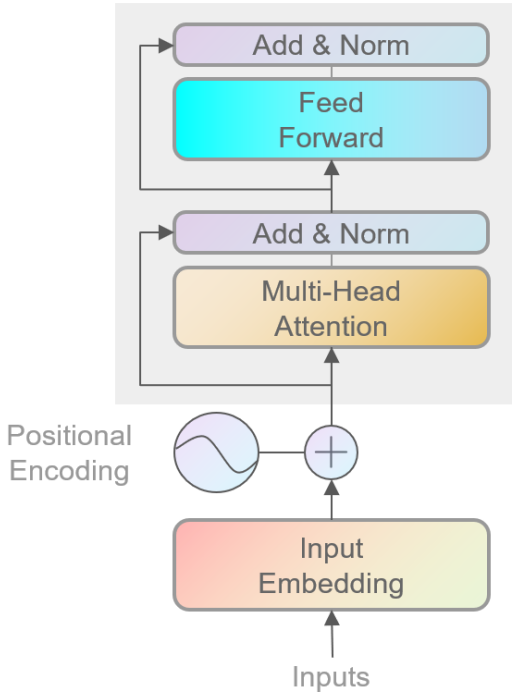


Figure 2.5: BERT encoder architecture: it takes word embeddings and position embeddings as input, then goes through the multi-head attention module with layer normalization and residual connection, and finally goes through the feed forward network with another set of layer normalization and residual connection. This figure is originally from [60]

.

The architecture of BERT contains a set of stacked transformer encoders. BERT-base consists of 12 layers of encoders, while BERT-large consists of 24 layers. As shown in Figure 2.5, the grey box in the diagram represents one part of the BERT encoder, which includes components such as the multi-head self-attention mechanism, residual connections [25] with layer normalization [4], and a feed-forward neural network, consistent with the transformer

model. The attention module enables BERT to learn semantic relations, and the feed-forward neural network is used to store knowledge from its training on the corpus.

BERT training can be divided into two parts: pre-training and fine-tuning. BERT's pre-training refers to the process where BERT learns general semantics from a large amount of text, including two tasks: Mask Language Modeling (MLM) and Next Sentence Prediction (NSP). The MLM can be understood as a cloze task, where the model randomly masks 15% of the words in each sentence and predicts the masked words based on their context. For example, 'I am happy today' becomes 'I am happy today [MASK]', where the word "today" is masked. The BERT model then predicts the word in the masked position. There are three specific ways to mask: in 80% of cases, the [MASK] token is used; in 10% of cases, a random word replaces the masked word; and in the remaining 10%, the word is unchanged. These three [MASK] strategies are designed to prevent overfitting and to avoid the model from simply memorizing that [MASK] corresponds to token 'today'. The MLM learning method enables the BERT model to see both the context before and after the mask, achieving bidirectional learning. Next Sentence Prediction involves selecting sentence pairs A and B, where in 50% of the cases, B is the next sentence of A, and in the remaining 50%, B is randomly selected from the corpus. The BERT model is tasked with determining whether two sentences are sequentially related. In BERT's pre-training tasks, MLM learns the semantic relationships between words within a sentence, while NSP learns the semantic relationships between sentences within a text.

The fine-tuning phase of BERT refers to a training process where BERT continues training on downstream tasks and updates parameters. Since the updated parameters are often in the last few layers of the BERT encoder, this process is referred to as fine-tuning.

Thanks to BERT's excellent model structure and its extensive pre-training process, BERT has achieved far superior results in many NLP tasks compared to traditional RNN models.

2.4.5 CodeBERT

Pre-trained language models such as BERT are pre-trained on large-scale text data to learn the statistical patterns and semantic information of languages. The vast knowledge and understanding ability make it powerful during fine-tuning on downstream tasks, such as text classification, named entity recognition, and so on.

CodeBERT [21], built upon the Transformer architecture, is pre-trained on both natural language and programming languages, including six programming languages such as Java, C, and Python, as well as text annotations in natural language. The volume of data used for pre-training is approximately 2,130,000 entries. It employs a dual-encoder framework to process natural language and programming language separately. This dual-encoder design allows CodeBERT to capture the intricate relationships between these two modalities, making it a powerful tool for tasks at the intersection of natural language and programming language.

The core principle behind CodeBERT lies in its pre-training methodology: Masked Language Modeling (MLM). For a given input sequence, it randomly masks 15% of the tokens with a special token '⟨mask⟩', then it asks CodeBERT to make a probability prediction on the masked token based on a given vocabulary. This process equips CodeBERT with a deep contextual understanding of the nuances present in both natural language and code constructs. This knowledge is then fine-tuned for specific software engineering

tasks, such as code generation, code recommendation, or code comprehension. We employ the vast code knowledge and generative ability of CodeBERT through mask-and-predict strategy [19] to mutate Simulink models.

Chapter 3

CodeBERT-assisted Mutation Testing for Simulink

In this chapter, we present a CodeBERT-assisted approach to mutant generation for Simulink, called BERTiMuS. First, we describe our mutation operators for Simulink models and explain how they are developed. Since not all types of mutation operators for Simulink models can be supported using CodeBERT, we also present a list of the mutation operators that BERTiMuS does not provide. Second, we elaborate on how we obtain BERTiMuS by further pre-training CodeBERT on the textual transformations of Simulink models, and we describe the process of using BERTiMuS to generate mutants. Lastly, we propose some additional processing strategies to further improve the diversity and quality of the mutants generated by BERTiMuS.

In addition to presenting BERTiMuS, this chapter describes three different approaches

to mutation testing: The first is the classical mutation testing [49, 50], where a mutant is killed if it can impact an output. The second is fitness-based mutation testing, which states that a mutant is killed if it can affect the assessment of a requirement (fitness value). The third is requirement-based mutation testing, which asserts that a mutant is killed when its impact on a requirement leads to the violation of the requirement that was originally satisfied by the model (fitness value polarity changed), or vice versa.

3.1 Mutation Operators of BERTiMuS

Table 3.1 shows a list of Simulink fault patterns and their corresponding mutation operators that are supported by BERTiMuS. The fault patterns include those identified from automotive engineers [45] as well as the patterns for Simulink mutation operators based on our survey of the literature [11, 26, 39, 53, 55]. We also present in Table 3.2 the Simulink mutation operators that are not used by BERTiMuS and discuss the reasons why they fall outside the scope of BERTiMuS. The operators presented in Tables 3.1 and 3.2 include all the Simulink-related mutation operators presented in the Mutation Testing Tools repository [52]. In the following paragraph, we introduce the mutation operators listed in Table 3.1.

Table 3.1: Simulink fault patterns and mutation operators used in BERTiMuS

Fault ID	**Fault Patterns**	**Corresponding mutation operators**	**References**
1	Incorrect signal data types in math equations	Changing the data type from double or float to int, and from single to double. Changing the unit of numbers, e.g., converting milliseconds to seconds for signal data. Replacing the MATLAB 'fixdt(0,8,3)' data type with the MATLAB 'fixdt(0,3,8)' data type. Converting arrays to other arrays.	[45]
2	Incorrect 'GoTo' block or 'From' block	Changing the tag value of a 'GoTo' block or the tag value of a 'From' block.	[45]
3	False 'Saturate on integer overflow' in math operations blocks	Changing the value between 'on' and 'off'.	[45]
4	Incorrect constant values	Changing the value of 'Gain' or 'Constant' block, such as changing the constant value from 0.5 to 50. The type of constant values can be array, such as replacing '[1,5,6]' with '[2,5,6]'.	[45], [31], [26], [39], [11]
5	Incorrect Simulink blocks	Changing arithmetic operators, e.g., replacing '-' with '+' or '*' or '/'. Changing relation operators, e.g., replacing '$\leq$' with '$\geq$' or '=' or '$\neq$'. Changing logical operator, e.g., replacing 'AND' with 'OR' or 'NOR'. Changing inputs value of Sum block, e.g., replacing inputs value '++' with '+-' or '–'.	[45], [31], [26], [39], [11]
6	False initial conditions and sample time	Changing the initial value and sample time in 'Integration' and 'Unit Delay' blocks.	[45], [11]
7	Incorrect transition conditions in Stateflow models	Modifying relation and logical operators in expressions, such as 'NOT' to 'AND', '<' to '>'.	[45], [31], [26], [39]
8	Incorrect variable names in Stateflow models	Changing variable names, such as changing 'send(LO)' to 'send(RI)', changing 'Gear=1' to 'Gear=2'.	[45], [39], [11]
9	Incorrect actions in Stateflow models	Changing arithmetic operators, modifying constants. Changing state duration, e.g., changing 'after(10 sec)' to 'after(8 sec)', changing logical operator '&&' to '||'.	[45], [39], [11]
10	Changing keywords in Stateflow models	Changing transition-related keywords, e.g., changing 'before' to 'after' or 'at'. Changing state-related keywords, e.g., changing 'entry' to 'exit' or 'on'. Changing event-related keywords, e.g., changing 'send' to 'broadcast'. Changing time-related keywords, e.g., changing 'every' to 'after'.	[45], [11]
11	Expand expressions in Stateflow models	Introducing negate operators, e.g., replacing '(x==y)' with '!(x==y)'. Adding arithmetic operations, e.g. , replacing 'x' with 'x+1', 'x-1' or 'x>1'. Adding condition statement, e.g. , replacing 'a==1' with 'a==1 || b==3'.	[31], [34]

Table 3.2: Simulink mutation operators that are not used in BERTiMuS, along with the reasons why these operators are not used

Mutation ID	**Mutation operators**	**Corresponding reasons**	**References**
1	Insert absolute value operator	BERTiMuS can not insert blocks.	[31]
2	Disable or create connections between blocks.	BERTiMuS only performs mutations inside blocks.	[7]
3	Statement Swap Operator	BERTiMuS only performs mutations inside blocks individually. Thus, it cannot operate on multiple blocks.	[39]
4	Scalar variable for array reference replacement	BERTiMuS can only replace a scalar with a scalar or an array with an array.	[31]
5	Block Removal Operator	BERTiMuS only performs mutations inside the blocks and thus cannot remove a whole block.	[39]

- Incorrect data types. The "incorrect data types" pattern refers to failures caused in math computations when variables with non-matching types are used together. For example, time cannot be multiplied by distance, and a scalar cannot be added to a vector. This fault type also includes changes in variables that lead to precision loss in mathematical formulas, such as changing a double variable to an integer variable.

- Incorrect ‘GoTo’ block or ‘From’ block. This mutation operator changes the label of a ‘GoTo‘ tag or a ‘From‘ tag, which affects the data and control flows within a Simulink model or the execution path of a model. In essence, it involves altering the source or destination of data flows. Additionally, a non-existent tag would disable the block, equivalent to deleting this block.

- Saturate integer overflow. In Matlab, mathematical blocks have an option called "saturate on integer overflow" that can be turned on or off. When this option is on,

operations exceeding the representable range of an integer type are capped at the maximum or minimum value, instead of wrapping around. Modifying this flag may affect mathematical precision and cause simulation failures or incorrect results, but in some cases, the overflow does not affect the final results.

- Incorrect constant. This may change value of 'Constant' block and result in false results.

- Incorrect Simulink blocks. This entails changing the property value of a block, which may affect the computation of the block and result in false output. For example, consider the 'Sum' and 'Inputs' blocks, where the property value of the blocks is represented as '++-'. This configuration computes the values from three blocks. If we change the property value to '++', then one block will not be included in the computation. Another example involves altering the logical operator value in the 'Operator' block, such as changing 'OR' to 'AND' or 'NOR', which can alter the state result and modify the computation pathway.

- False initial conditions and sample times. False initial conditions can lead to inaccurate starting states in Simulink models, impacting stability and results. Incorrect sample times can introduce errors and affect model accuracy, with lower sampling rates reducing precision.

- Incorrect transition conditions in Stateflows. This may lead to logical errors, incorrect states, or failures in state transitions, affecting the computational path and resulting in erroneous outcomes. The fault pattern for incorrect transition conditions may

involve the use of incorrect variable names in these conditions. In such cases, the state machine may be unable to access or update these variables properly, or it may improperly access other variables.

- Changing keywords in Stateflow. This can significantly impact the behavior and logic of the state machine. Below, we present possible types of keywords:

  1) Transition-related keywords, such as 'before', 'after', and 'at'. These keywords affect the transition conditions due to different timelines. 2) State-related keywords, such as 'entry', 'exit', 'during', 'on', and 'bind'. These keywords influence the model to perform or quit actions depending on the state. 3) Event-related keywords, such as 'event', 'send', and 'broadcast'. These keywords can declare, send, or broadcast events to trigger actions or transitions, and can activate one or multiple states. 4) Time-related keywords, such as 'timeout', 'after', and 'every'. These keywords affect the timing of executing or checking actions and transitions.

3.2 Simulink Pre-training on CodeBERT

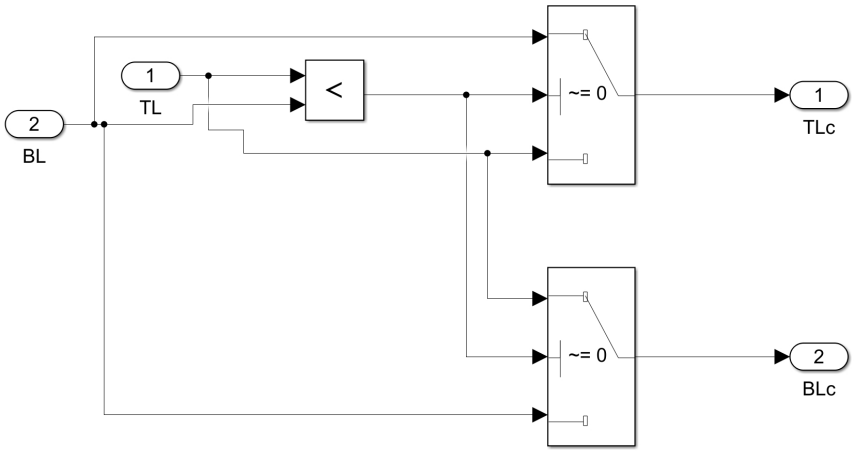
The pre-training stage mainly consists of three steps: Step 1: Convert Simulink models into text, which we refer to as a *Simulink sequence*. Step 2: Parse the information in Simulink sequences into sequence data, capturing details of the blocks such as block types and property values. Step 3: Perform MLM pre-training methodology [19] using CodeBERT with Simulink-sequence data. The pre-training stage provides Simulink-related knowledge to CodeBERT, which helps CodeBERT generate high-quality mutations. In the following

paragraph, we elaborate on the details of each of the three steps briefly outlined above.

- *Step 1:* Parse Simulink models into sequences. In order to apply CodeBERT to mutating Simulink models, we parse the structure of Simulink models and their blocks into sequence data. Given a Simulink model with the '.slx' extension, we extract the XML file from the slx model. The extracted XML file contains the structure of the model and the information about all the blocks in a Simulink model, such as SID, block types, and property values.

- *Step 2:* Parse the Simulink sequences into sequence data containing information about all blocks. It should be noted that a Simulink model may contain several hierarchical system levels, each containing certain blocks. We gather blocks that belong to the same system together because these blocks are strongly connected and can provide a code-semantic context for sequences. Here, we present an example of a Simulink block and its corresponding sequence data in Figure 3.1. The sequence data is parsed into JSON format using scripts. We view this JSON format sequence data as text for pre-training. The quantity of sequence data for a Simulink model equals the number of system levels in the model. We also apply filters to the sequence data by ensuring that only sequence data containing at least one block with a property value is considered valid. The essential information of a block includes block type, SID (identification of a block), and property value. This is because the input length of CodeBERT is 512, we only keep the essential information of blocks so that we can input as many blocks as possible.

  We note that we do not consider mutating the connections among blocks directly



a) Simulink blocks

{“system2": [{"BlockType": "Inport", "Name":
"TL",{"BlockType": "Inport", "Name": "BL"}, {"BlockType":
"RelationalOperator", "Name": "Relational", "property":
{"Operator": "lt;", "OutDataTypeStr": "boolean"}},
{"BlockType": "Switch", "Name": "Switch1", "property":
{"Threshold": "0.5", "SaturateOnIntegerOverflow": "off"}},
{"BlockType": "Switch", "Name": "Switch2", "property":
{"Threshold": "0.5", "SaturateOnIntegerOverflow": "off"}},
{"BlockType": "Outport", "Name": "TLc"}, {"BlockType":
"Outport", "Name": "BLc"}]}

b) Simulink sequence data

Figure 3.1: (a) shows several Simulink blocks at system level. We convert these blocks into sequence data, as shown in (b). The information in sequence data includes block types, names and property values of blocks.

because CodeBERT is designed only for sequence-format data and cannot be applied directly to graph-structured data. For a block, it may be connected to several blocks, and the information of connected blocks is simplified since we convert the graph connections into text. The blocks may be placed in different orders, but it is still better than not using the information of connected blocks at all. We ensure that connected blocks appear in close proximity in Simulink sequences for simple Simulink models. Thus, the information of nearby blocks can be used as context, allowing us to use the connections among blocks indirectly.

Since CodeBERT is based on BERT, a bi-directional transformer model, it can capture and understand the semantic relations among the current code tokens and their surrounding code tokens, referred to as context. The information of other related blocks serves as context for the current blocks. Additionally, within a block, the block type can be used as context to help CodeBERT predict the property value. For instance, the Sum block and the Gain block usually compute with blocks that have numeric values. If the block type is 'Gain' or 'Constant', then CodeBERT would likely predict the property values as numbers or arrays. If the block type is 'Operator', then CodeBERT would likely predict the property value as a logical expression, such as 'OR' or 'AND'.

- *Step 3:* pre-training CodeBERT on Simulink sequences. We collect 2,611 Simulink models from SLNET [1]. SLNET is a collection of third-party Simulink models gathered by mining open source repositories, including GitHub and Matlab Central. We

[1] https://zenodo.org/records/5259648

apply the information parsing method from Step 2 and parse 6,776 valid sequence data items, which are then used for pre-training. We adopt the MLM pre-training methodology [19] to further pre-train the CodeBERT model on Simulink sequences. The goal of our pre-training is to provide CodeBERT with Simulink-related knowledge and enhance its performance on Simulink-related tasks. Specifically, we randomly mask 15% of the tokens in a sequence data and perform pre-training based on CodeBERT for 15 epochs. Since the maximum input token length for CodeBERT is 512, if the length of a sequence data exceeds 512, we use a sliding window technique to extract sub-sequences of length 512 until the end of the sentence. To avoid introducing unnecessary names, we refer to the CodeBERT pre-trained on the Simulink corpus as BERTiMuS, using the same name as our approach, as this does not introduce any ambiguity.

3.3 BERTiMuS: CodeBERT-assisted Mutation of Simulink Models

We show the workflow of applying BERTiMuS for mutating Simulink models in Figure 3.2.

We use the Mask and Predict Mutation methodology [19] for generating mutations, it replaces a token with '<mask>' token and uses Pre-trained Language Models (PLMs) to predict new tokens. Given a Simulink model, we follow the steps 1 and 2 in Section 3.2 to obtain the Simulink sequences. To provide as much block information to BERTiMuS as possible, we set the input sequence length at the max input length 512 and use a

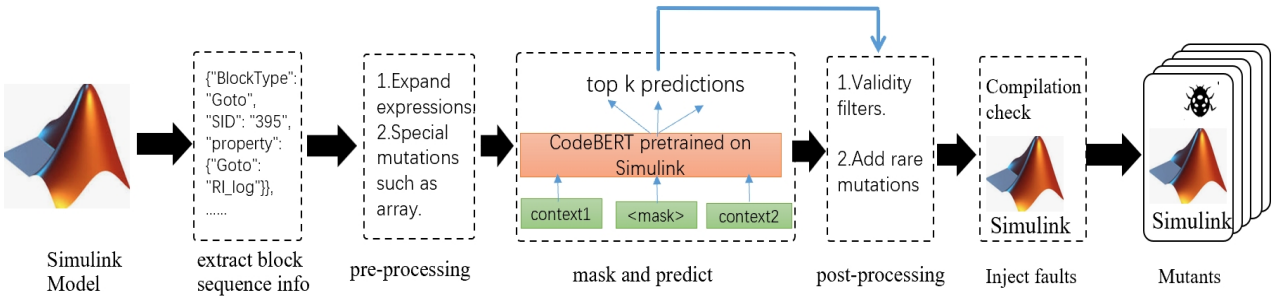


Figure 3.2: BERTiMuS workflow: (1) Given a Simulink model, extract the information of blocks using the method described in Step 2. (2) Use pre-processing strategies to enhance the diversity and quality of the mutations. (3) Mask the property values of the tokens in sequence data and use the CodeBERT pre-trained on Simulink to generate predictions for these tokens, obtaining the top-3 predictions. (4) Apply post-processing strategies to discard useless or low-quality mutations, and incorporate rare mutations. (5) Inject the mutations (faults) into the original Simulink model to create mutated Simulink models, referred to as mutants.

sliding window to truncate it if the sequence length exceeds 512. The sliding step is 128 to make sure we do not lose any information. Then, we perform masking on the Simulink sequences by masking only the property values of the blocks, while ignoring the mutation of block types and block names. This approach avoids changing the block types and block names, which would disable the block and thus yield low-quality mutations. We mask one property value at a time and obtain one prediction result, meaning we apply only one mutation operator to the Simulink model to produce a First Order Mutant (FOM). This approach helps us analyze the effects of mutations based on testing results, isolating the effects of other factors.

Here, we describe the workflow of using BERTiMuS for generating mutations. Given an original Simulink model $\mathcal{M}$, we follow the steps described in Section 3.2 to extract the sequence data $S_1, S_2, \ldots, S_i$ from $\mathcal{M}$, where i denotes the number of system levels in the

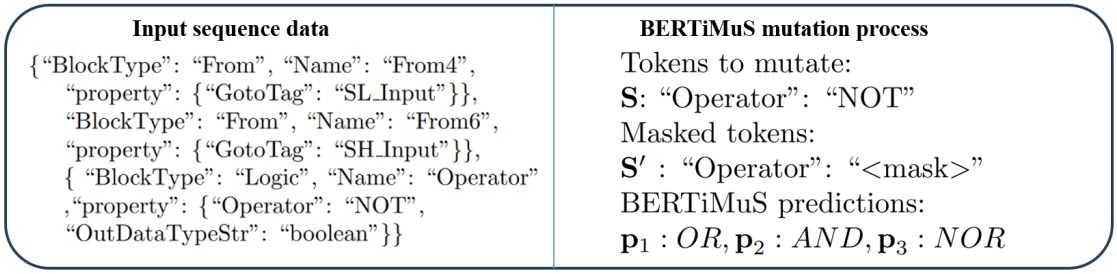
Simulink model. This set represents the sequences we obtain, corresponding to different levels within $\mathcal{M}$. For each sequence S_j (where $1 \leq j \leq i$), we identify parts that can be masked, leading to the generation of masked sequence data $S'_1, S'_2, \ldots, S'_N$, where N is the total number of masked sequences generated. Each S'_k (where $1 \leq k \leq N$) represents a variant of the original sequences with specific elements masked.

We input each masked sequence into BERTiMuS to generate probability distributions for the '<mask>' tokens. For each masked token, we analyze its probability distribution and select the top three tokens with the highest probabilities. These top predictions are denoted as p_1, p_2, and p_3, representing the most likely tokens to replace the masked positions.

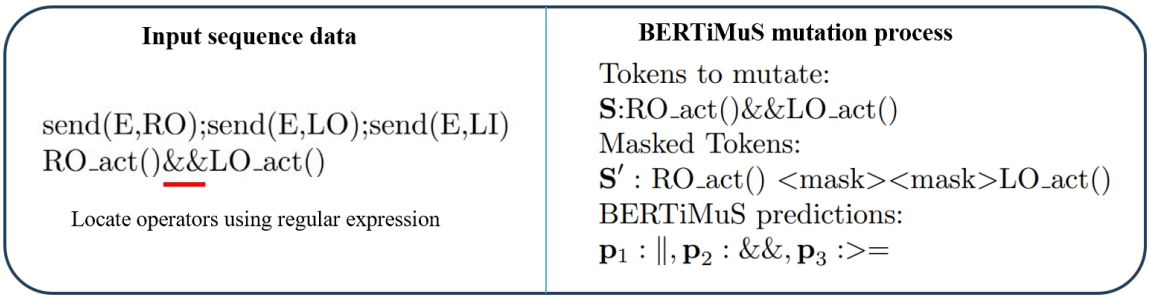
After obtaining predictions, we apply the predictions on the original Simulink model $\mathcal{M}$ to generate mutated Simulink models $\mathcal{M}_1, \mathcal{M}_2, \mathcal{M}_3$ (mutants). We discard the mutants that cannot be compiled, or mutants that are the same as the original Simulink model (i.e., the predicted value is the same as the original property value). Below, we present the different types of mutation operators that can be produced by BERTiMuS.

3.3.1 A. Arithmetic/Logical Operators Mutation

Arithmetic and logical operators are specified in two ways in Simulink: First, the operators directly appear as property values in the Simulink sequence data. For example, the input sequence data is ' "property": {"Operator": "NOT"} '. Second, arithmetic or logical operators appear in expressions; this usually occurs in Stateflow. We locate the operators using regular expressions and replace them with the '<mask>' token. We show two logical



a) Mutation on property values in blocks: The block contains a logical operator 'NOT', the BERTiMuS masks the 'NOT' token to make predictions.



b) Mutation on expressions in Stateflow model: For the operator appears in the expressions of Stateflow model, we first locate the operator using regular expression, and then mask the operator tokens and use BERTiMuS to make predictions.

Figure 3.3: BERTiMuS logical operator mutation examples

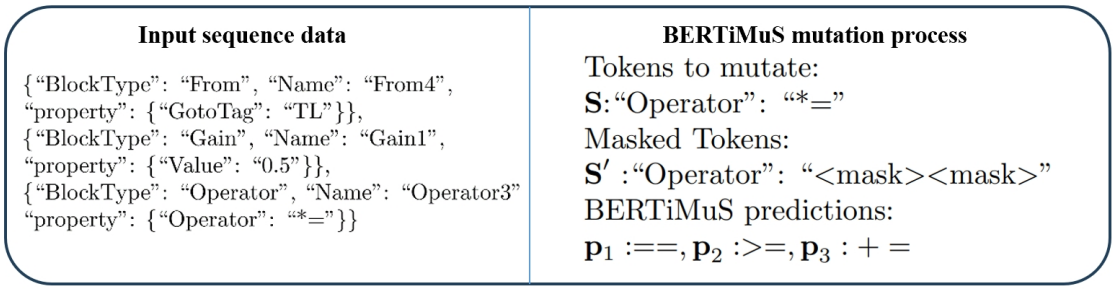


Figure 3.4: BERTiMuS arithmetic operator mutation example

operator mutation examples in Figure 3.3, representing the two different cases mentioned above, and an arithmetic operator mutation example in Figure 3.4.

3.3.2 B. Number Mutation

The numeric types in Simulink models include int, float, double, and bool. Similar to the operators, numbers are represented in two different ways in Simulink models: numbers appear as a property value in the Simulink sequence data, and numbers appear in expressions in Stateflow. For instance, after locating and masking the number tokens using regular expressions, the predictions for the number '20' may include numbers such as '50', strings such as 'is', and expressions like '+5'. We retain only the predictions for numbers and expressions and use functions to convert the expressions into numerical values.

3.3.3 C. Variable Name Mutation

Mutating the variable names may cause the blocks to lose certain functional abilities, lose connections with other blocks, or interact with incorrect blocks. Below, we present some examples for Simulink variable mutations and provide some possible mutations produced by BERTiMuS.

- For the variable name of Saturate on integer-overflow value, we replace the literal name with one '<mask>' token, and the possible predictions are 'off', 'on', 'ON' and 'OFF'.

- For the Gototag values in From or Goto block, which are literal names such as 'apfail', 'TL' and 'state', the predictions maybe random or other gototag values appeared in the context. For example, the prediction 'stby' is a gototag value of other blocks appeared in the sequence data, and 'scbe' is a meaningless string. Similarly, the predictions for mutations of the gototag 'BL', which means the bottom limit, may include another closely-related value 'TL', which denotes the top limit. BERTiMuS learns the knowledge of similar Gototag values through pre-training on Simulink corpus.

- For the variable name mutations, we use an example in a Stateflow model: the tokens to be mutated are 'send(E,LI);send(E,RO);send(E,LO)', by masking the 'LO', the meaningful predictions BERTiMuS provided are 'LI', 'RO', because these variable names have appeared in the proximity of 'LO', i.e., the same context for 'LO'.

3.4 Additional Processing Strategies

In order to generate more diverse and higher-quality mutations, and to discard low-quality mutations, we adopt some pre-processing and post-processing strategies outlined below:

3.4.1 Pre-processing

1. Expand expressions in Stateflow model. Expressions are common in Stateflow models, such as expressions 'x', 'a>=2' and 'condition1 && condition2'. To generate more complex mutations, we expand the expression in three ways. 1) Adding negate operators,

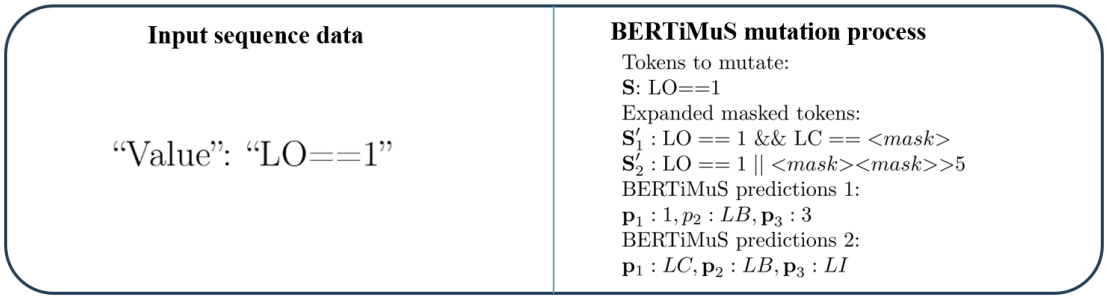


Figure 3.5: BERTiMuS expanding expressions mutation example: for the expression 'LO==1', we expand it with other expressions 'LC==1' or 'LC>5', then we connect them through some random operators, such as '&&' or '||'. After obtaining the expanded expressions, we mask the tokens and get prediction results from BERTiMuS.

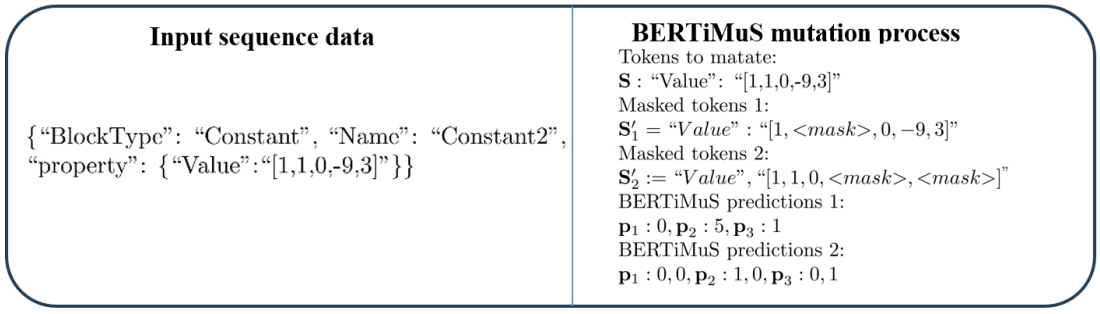


Figure 3.6: BERTiMuS special mutation example: If the value of a Constant block is an array, we mask the number inside the array instead of masking the whole array. This is because arrays appear very rarely as the value of a Constant block. If we mask the whole array, BERTiMuS typically would not predict an array as the result. However, if we only mask values inside the array, we can keep the results as an array, thus ensuring the computation remains error-free.

such as replacing 'LI_act()|RO_act()' with '!LI_act()|RO_act()', replacing '(x==y)' with '!(x==y)'. 2) Adding arithmetic operations, such as replacing 'x' with 'x+1', 'x-1' or 'x>1'. 3) Adding condition statement, such as in Stateflow model, replacing 'a==1' with 'a==1 && b==3'.

However, using the mask and predict method cannot add new tokens, which means we cannot add new content. For instance, the expression 'a==1' correspond to 4 mask tokens in BERTiMuS, so it can only give a prediction length of 4 tokens, thus cannot expand the expression by adding a condition statement. Therefore, for expressions in Stateflow model, we randomly add elements into the expression, including: 1) Randomly add negate operators '!' in front of the expression. 2) Randomly add arithmetic operations if it is a single variable. For instance, we expand 'x' into 'x+<mask>' or 'x==<mask>' and then ask BERTiMuS to predict the mask value. 3) Randomly add a new condition statement, such as expanding 'a==1' to 'a==1 && b==<mask>' or 'a==1 || <mask> > 5' and then ask the BERTiMuS to predict the masked value. We provide an example of expanding expressions mutation in Figure 3.5.

2. Special mutations. The values of the Constant block could be arrays. Since replacing an array with a scalar, or replacing a scalar with an array, usually cannot be compiled, we only consider how to use special mutation strategies to replace an array with a different array. That is, if a value is an array, we only mask each value inside the array. For example, for the array '[5,7,9,-1]', the masked sequence would be '[<mask>, <mask>, <mask>, <mask>]'. BERTiMuS will then offer predictions for the masks and generate new arrays. We provide an example of special mutations in Figure 3.6.

3.4.2 Post-processing

1. Validity filter. We apply a filter to the prediction results based on the block type to discard invalid mutations. For instance, for the block type 'Saturate on integer-overflow', we only keep valid values 'on' and 'off'. For the block type 'Operator', we only retain arithmetic or logical operators as results, ignoring strings or numbers. For the block type 'OutDataTypeStr', we only keep valid values such as 'uint8', 'double', 'int16', and so on. For the block type 'Gain', we only keep predictions that are numbers.

2. Add rare mutations. Some blocks may have special values that rarely appear, making it difficult for BERTiMuS to learn enough information about them, and as a result, BERTiMuS seldom predicts these special values. For instance, for the 'LogicalOperator' block, since most of the values are 'NOT' or 'AND', it is challenging for the model to predict rare values such as 'isNaN' and 'isInfinite'. For the 'RelationalOperator' block, while the common values are '<', '>', '<=', and '>=', there are also rare values such as '==' and ' ='. Since BERTiMuS is unlikely to predict these uncommon values, we randomly select some uncommon values and include them in the predictions to create more diverse mutants.

3.5 Requirements-driven Mutation Testing

To measure the extent to which the model's requirements are affected, we design three measurements for determining when a mutant is killed by a test case. We recall the notation defined in Section 2.2.1: p denotes a mutant, $\mathcal{M}$ denotes the original Simulink model, t represents a test case, $\mathcal{T}$ denotes the test suite, $O(t, \mathcal{M})$ denotes the output of t

when exercising model $\mathcal{M}$, and $O(t, p)$ denotes the output of t when exercising the mutant model p.

In Section 2.2.1, we discussed the three conditions under which a test case can kill a mutant. However, this approach is not practical when the software under test must be validated against specific requirements, since output changes can not directly be related to requirements. Recently, property-based mutation testing (PBMT) [8] has been proposed to assess the capabilities of a test suite to evaluate the software under test with respect to a given property. It aims to find a test case t such that the output of the original model $\mathcal{M}$ satisfies the property ϕ, while the output of the mutant p violates ϕ.

For Simulink models, we typically capture requirements satisfaction using quantitative fitness measures [45]. Compared to output values (classical mutation testing [49,50]), these fitness measures provide a more nuanced way to evaluate the requirements of a Simulink model. Briefly, for a Simulink model $\mathcal{M}$ and a test case t, when the fitness measure related to a requirement R is positive, it indicates that t does not show any violation of R for $\mathcal{M}$, i.e., R holds for $\mathcal{M}$. Conversely, when the fitness measure is negative, it suggests that t reveals that $\mathcal{M}$ violates R. Provided with such a quantitative fitness measure, we can assess mutants in two ways: (1) the mutant, when compared with the original model, leads to a different value of the fitness measure; and (2) the mutant, when compared with the original model, leads to a violation of the requirements. In the former case, the values of the fitness measure on the mutant and the original model differ but do not necessarily change polarity, indicating that the mutant impacts the fitness measure for a requirement but does not necessarily cause its violation. In the latter case, the value of the fitness measure changes from positive on the original model to negative on the mutant, clearly

indicating that the mutant causes a violation of the requirement.

Below, we respectively describe the different notions of killing mutants by test cases based on outputs, changes in fitness measures, and violations of requirements.

Definition III.1 *o-killed mutant*, which means output difference, if any one of the output values of the mutant model is different from the original model, we assume this mutant is killed by the test case. Formally, a mutant p is said to be $o - killed\ mutant$ by a test case t if there exists an output value $o \in O(t, p)$ such that it differs from the corresponding output value $o' \in O(t, \mathcal{M})$, that is, $\exists o \in O(t, p), o \neq o'$.

Definition III.2 *f-killed mutant*, which means changes in fitness measures, if any one of the fitness values of the mutant model is different from the original model, we assume it is killed. We denote the fitness values of a test case t exercised on original Simulink model as $F(t, \mathcal{M})$, and exercised on mutant p as $F(t, p)$. Formally, a mutant p is said to be $f - killed\ mutant$ by a test case t if there exists a fitness value $f \in F(t, p)$ such that it differs from the corresponding fitness value $f' \in F(t, \mathcal{M})$, that is, $\exists f \in F(t, p), f \neq f'$.

Definition III.3 *r-killed mutant*, which means violations of requirements. We assume the value of fitness function above or equal to zero as meeting the requirement, and below zero as violating the requirement. Therefore, if any one of the fitness values' polarity changes from non-negative to negative, we assume it is killed. Formally, a mutant p is said to be $r - killed\ mutant$ by a test case t such that, if there exists a non-negative fitness value in the original model $f' \in F(t, \mathcal{M}), f' >= 0$, and the corresponding fitness value in the mutant is negative $f \in F(t, p), f < 0$, that is, $\exists f' \in F(t, \mathcal{M}), f' >= 0, f < 0$.

Chapter 4

Evaluation Strategy

This chapter presents our empirical strategy for evaluating BERTiMuS. Section 4.1 introduces the research questions addressed by our evaluation. In Section 4.2, we provide a detailed description of the study subjects (models) used for experiments. In Section 4.3, we introduce the experimental setup, including training experiment settings, test suites, and the baseline approaches we used. Finally, in Section 4.4, we elaborate on the experimental procedure for conducting our evaluation.

4.1 Research Questions

Our experiments seek to address the following research questions:

RQ1. *Can the mutants generated by BERTiMuS generate test cases that state-of-the-art mutant generation tools for Simulink cannot?* To answer this question, we compare

the test cases selected to kill the mutants produced by BERTiMuS and a state-of-the-art mutation testing technique with respect to both classical and property-based mutation testing, as described in Section 3.5.

For the baseline mutation tool, we consider the Fault Injection and Mutation for Simulink (FIM) tool [7], an open-source toolkit designed for automated fault injection and mutant generation in Simulink models. It supports the injection of faults between blocks and mutating individual blocks. In our experiments, we have utilized eight out of the ten different FIM fault types, as discussed later in Section 4.3.3. Specifically, these eight fault operators are those that yield the highest number of mutants.

In addition to comparing the test cases derived using mutants generated by BERTiMuS and FIM, we qualitatively analyse the mutants generated by BERTiMuS and compare them with the list of fault patterns for Simulink models in the literature [45] through the following question:

RQ2 *How do the mutants generated by BERTiMuS compare with those generated by existing fault patterns for Simulink models? And what are the advantages of BERTiMuS when compared with a human using the same fault patterns for Simulink models?* We analyze the mutants generated by BERTiMuS and qualitatively compare them with those that are derived from the Simulink fault patterns and mutation operators proposed in the literature [45]. We showcase the example mutants that BERTiMuS can generate but may not be derived from the existing Simulink fault patterns and mutation operators. Similarly, we outline the existing mutation operators and fault patterns that cannot be captured by BERTiMuS. We also analyse the advantages of BERTiMuS compared to a naive baseline,

which uses humans to generate the mutants using the same fault patterns.

4.2 Study Subjects

We use five Simulink models in our experiments: Tusin, Twotanks, FSM , ATCS and AECS. Table 4.1 shows the characteristics of our study subjects including their number of blocks, number of requirements, number of outputs, and whether the model include Stateflows or not. The first three models are taken from the Lockheed Martin benchmark [1] for cyber physical systems, and the last two models are from the FIM repository, our baseline tool for generating Simulink model mutants. We note that the number of requirements and the number of outputs are included because this information is important for computing our classical and property-based mutation testing scores. Specifically, the classical mutation score concerns outputs and the property-based mutation testing concerns requirements.

Below, we describe the Simulink models listed in Table 4.1 and used as experimental subjects:

Tustin: Tustin Integration model is designed for the precise computation of signal integration in flight control systems using the Tustin method. It computes the integration of an input signal xin, producing an output yout based on a formula involving the time step T and the previous values of xin and yout. The model is distinct for its integration range limits, defined by Top Limit (TL) and Bottom Limit (BL), ensuring that the output *yout* remains within specified boundaries.

Crucial to Tustin's functionality is a reset control (Boolean type) and an initial condi-

Table 4.1: Key features of the Simulink models used for experiments.

Name	**Description**	**#blocks**	**#Reqs**	**#Outputs**	**Stateflow(Y/N)**
Tustin	A common flight control utility for computing the Tustin Integration.	23	9	1	N
Twotanks	Two tank system controls liquid level, sensors and pump.	267	34	11	N
FSM	Finite State Machine containing Autopilot System and Sensor.	120	13	6	N
ATCS	A typical automotive drivetrain.	16	1	3	Y
AECS	An aircraft system controlled by the elevator position.	86	2	2	Y

tion, ic (double type). The reset control allows the output to revert to the initial condition value, providing flexibility and control in dynamic operational scenarios. An initial condition (ic) is a predefined starting value that the system's output reverts to when the reset control is activated, ensuring precise and controlled computations from a known state.

Twotanks: Twotanks model is a system designed to simulate and control a dual-tank setup, each governed by individual controllers with integrated sensors. This model manages the liquid levels in both tanks, ensuring they remain within safe operational ranges while facilitating the controlled flow of liquid between the two. The core functionality of the model lies in its ability to balance the inflow and outflow in each tank, guided by the feedback from various sensors that monitor liquid levels at different heights.

Inputs to Twotanks model consist of the initial liquid heights in both Tank 1 and Tank

2, which set the starting conditions for the simulation. The model's outputs are a total of eleven parameters that include the current liquid height in each tank, the binary state of sensors indicating specific liquid levels, and the operational states of pumps and valves in both tanks. The output data provides fine-grained visibility into the dynamic interplay between the tanks, offering essential information for understanding and optimizing the control of liquid levels and flows in such coupled tank systems.

FSM: The Finite State Machine (FSM) in this context represents a cyber-physical system designed for an Advanced Autopilot System interacting with an independent sensor platform. This FSM is structured to ensure safe automatic operation, particularly when navigating hazardous obstacles. It operates by integrating two main components: the autopilot system and the sensor platform. The autopilot, closely linked with the vehicle's flight control computer, is responsible for executing safety maneuvers in response to detected hazards. The sensor platform acts as the reporting mechanism, alerting the autopilot system of imminent dangers. The FSM effectively manages the states and transitions of these components based on various inputs, ensuring responsive and safe operation.

In terms of inputs and outputs, the FSM is governed by a range of Boolean and double-type variables. Inputs include 'standby', indicating pilot control; 'apfail', a failure indication; 'supported', reflecting the system's health; and 'limits', a sensor-related safety indicator. The primary output is 'pullup', a Boolean variable indicating whether the autopilot is in a maneuver state. Additionally, the autopilot has its own inputs like 'good', a sensor output, and outputs such as 'MODE' and 'REQUEST' for internal moding, and 'PULL' and 'STATE' indicating the autopilot's current state. The sensor, on the other hand, receives inputs like 'mode' and 'request' from the autopilot, and its outputs include

'good', indicating hazard detection, and 'SENSTATE', a state indicator. These inputs and outputs collectively ensure that the FSM dynamically adapts to changing conditions, prioritizing safety in the autopilot system's operation.

ATCS: The Automatic Transmission Controller System (ATCS) is an integral component in modern automotive drivetrain systems, primarily focused on regulating vehicle dynamics. It operates through user inputs via throttle and brake, which are instrumental in controlling the vehicle's speed and the engine's RPM. Central to the ATCS's design is its adherence to safety norms, which mandate that both the vehicle's speed and engine speed must not surpass certain threshold limits. These thresholds are a crucial part of the system's safety features, ensuring the vehicle operates within safe and efficient parameters.

AECS: Aircraft Elevator Control System (AECS) plays a pivotal role in controlling the positions of an aircraft's left and right elevators based on pilot commands. The AECS's primary objective is to maintain a stable elevator position, especially crucial during level flight, to ensure the aircraft's balance and stability. Key to its operation is a safety requirement that demands stabilization of the elevator position within a specific range when the pilot's command exceeds a predetermined threshold.

4.3 Experimental Setup

4.3.1 Experimental Setting

We conducted our experiments on a Windows 11 system with an Intel i9-14900k CPU and 64 GB RAM. The MATLAB versions we use are R2020b and R2022b. We use version

R2022b for all models expect AECS, which was incompatible with this version. For this model only, we did the analysis in version R2020b.

For pre-training CodeBERT on our Simulink corpus, we conduct experiments on Python 3.10 and CUDA 11.3, and we use PyTorch 1.10 and Hugging Face Transformers library [62]. The hyper-parameter settings for further pre-training include: a learning rate of 5e-6, a batch size of 64, 15 epochs, a warm-up ratio of 0.1 for the learning rate scheduler, and a maximum sequence length of 512.

4.3.2 Test Suites

We generate test suites using Adaptive Random Testing (ART) [40]. ART is a basic approach that produces test cases that are uniformly distributed across valid input ranges, thereby helping increase diversity in the test inputs. To be noted, the RQ1 results depend on the quality of the test suite created, and we create 20 test cases for each model to eliminate the effects of randomness.

4.3.3 Baselines

We compare our approach with two baselines: (1) The FIM tool [7], which is used as a baseline approach in RQ1. (2) A naive manual baseline that compares the mutants generated by BERTiMuS with those that could be obtained from the list of Simulink fault patterns and mutation operators in the literature. This latter baseline is used in RQ2.

FIM prototype tool approach. Fault Injection and Mutation for Simulink (FIM) [7]

is an open-source toolkit designed for automated fault injection and mutant generation in Simulink models. It supports the injection of common types of faults into specific parts of the model, and is able to customize the parameters of mutation operators. FIM employs automated tools to inject faults and observe the model's response, facilitating a comprehensive evaluation of the model's behavior under adverse conditions.

FIM generates mutants in two ways: (1) by injecting faults between blocks and (2) by mutating individual blocks. In our experiments, we have used only the former way of generating mutants by FIM, because BERTiMuS already supports the latter way. We utilize the following faults in our experiments:

1) "Negate": u to -u (for all signals of type 'double');

2) "Invert": inverts a non-zero signal 'u' to 0, otherwise if u is 0, then inverts u to not u i.e., 0 to 1 (for all signals);

3) "Stuck-at 0": (Zero fault): makes the signal value 0 for the specified time (for all signals);

4) "Absolute": u to |u| (for all signals of type 'double');

5) "Noise": Adds a band limited white noise to the input signal based on the specified fault value;

6) "Bias/Offset": Adds a predefined +ve or -ve offset (bias) value to the input signal (for all signals of type 'double');

7) "Stuck-at": the signal value stucks at the last correct value before fault occurrence (for all signals);

8) “Time Delay”: introduces a delay of specified duration in the input signal;

Naive approach uses the same mutation operators mentioned in Section 3.1, but creates mutants manually.

4.4 Experimental Procedure

In this Section, we describe the experimental procedure for conducting a quantitative evaluation of the effectiveness of BERTiMuS and the FIM-based mutation generation approach.

4.4.1 Test Suites Generation Procedure

To answer RQ1, we start by generating mutants using the BERTiMuS and FIM methods. Since FIM generates more mutants overall than BERTiMuS, we adopt a prior experimental procedure to compare classical mutation testing with CodeBERT-based mutation testing techniques for code [16]. Knowing that the two techniques generate different numbers of mutants, this experimental procedure aims to objectively compare the two mutant generation techniques in terms of the number of mutants generated and the effectiveness of the test cases yielded by each mutation strategy.

The experiment procedure is outlined in Algorithm 4.1. We apply this procedure to the set of mutants generated by each of the FIM and BERTiMuS methods applied to each of our Simulink subject models. The input to Algorithm 4.1 is a set of mutants. These mutants are generated by BERTiMuS and FIM, respectively.

Algorithm 4.1 Minimal mutant-killing test cases selection algorithm for comparing different mutant generation approaches [7]

Input: A test suite $\mathcal{T}_{main}$, and a set $I = \{i_1, \ldots, i_k\}$ of mutants
Output: A minimal set $\mathcal{T} \subset \mathcal{T}_{main}$ that kills some mutant in I, and a subset $I' \subset I$ of unkilled mutants

```
while I ≠ ∅ do
    Randomly select a mutant i_j ∈ I and remove it from I
    if some tc ∈ T_main can kill i_j then
        Add tc to T and remove from I, any other mutant i_j' ∈ I that tc can kill
    else if no tc ∈ T_main kills i_j then
        Add i_j to the set I' of unkilled mutants
```

The strategy in Algorithm 4.1 simulates a scenario where a tester selects mutants based on which he/she designs tests to kill them. Provided with the same test suite $\mathcal{T}_{main}$ for a set I of mutants obtained by method A, Algorithm 4.1 generates a minimal set $\mathcal{T} \subset \mathcal{T}_{main}$ of test cases that can kill as many mutants as possible in I as well as a subset I' of I of mutants that could not be killed by any test case in $\mathcal{T}_{main}$. In our experiment, we consider both classical and property-based mutation testing, described in Section 3.5.

4.4.2 Test Case and Mutant Comparison Procedure

We provide an experimental procedure for comparing the effectiveness of the test cases generated by BERTiMuS and FIM. Based on Algorithm 4.1, let $\mathcal{T}_{Main}$ be the reference test suite for Simulink model i (BERTiMuS and FIM use the same reference test suite $\mathcal{T}_{Main}$ for mutation testing). We denote the minimal mutant-killing test cases for each approach as $\mathcal{T}_{Bert}$ and $\mathcal{T}_{FIM}$, respectively. Note that $\mathcal{T}_{Bert} \subseteq \mathcal{T}_{Main}$, $\mathcal{T}_{FIM} \subseteq \mathcal{T}_{Main}$. We aim to investigate the difference and intersection of $\mathcal{T}_{Bert}$ and $\mathcal{T}_{FIM}$, then we try to investigate the

number of mutants of each approach that can only be killed by $\mathcal{T}_{Bert}$, $\mathcal{T}_{FIM}$ or by both.

We describe the procedure for test case comparison in Algorithm 4.2. We obtain the test cases $\mathcal{T}^i_{Bert}$ and $\mathcal{T}^i_{FIM}$ of each Simulink model i, and sum the number of test cases by 5 models. More precisely, to make sure each model contributes equally to the total results, we normalize the result of each model by dividing the total number of test cases in $\mathcal{T}^i_{Main}$. Then, we sum the number of the normalized test cases of each model, and get the difference and intersection between $\mathcal{T}_{Bert}$ and $\mathcal{T}_{FIM}$, where: $|\mathcal{T}_{Bert\text{-}only}|$ denotes the normalized number of test cases that appear only in $\mathcal{T}_{Bert}$; $|\mathcal{T}_{FIM\text{-}only}|$ denotes the normalized number of test cases that appear only in $\mathcal{T}_{FIM}$; and, $|\mathcal{T}_{overlap}|$ denotes the normalized number of test cases that appear in both $\mathcal{T}_{Bert}$ and $\mathcal{T}_{FIM}$. Note that $|\quad|$ represents cardinality operator, which is the number of elements in a set and $\setminus$ represents the set difference operator.

Based on $|\mathcal{T}_{Bert\text{-}only}|, |\mathcal{T}_{FIM\text{-}only}|$ and $|\mathcal{T}_{overlap}|$, in Figure 5.1, we plot the test case intersection and difference by output-based mutant killing, fitness-based mutant killing and requirement-based mutant killing.

Algorithm 4.2 Experimental procedure for test cases comparison

Input: Five Simulink models, mutants of these models generated by BERTiMuS and FIM.
Output: $|\mathcal{T}_{Bert\text{-}only}|, |\mathcal{T}_{FIM\text{-}only}|$ and $|\mathcal{T}_{overlap}|$
for *each model* i **do**
  let $\mathcal{T}^i_{Main}$ be the reference test suite for model i,
  use Algorithm 4.1 to obtain $\mathcal{T}^i_{Bert}$ and $\mathcal{T}^i_{FIM}$ based on $\mathcal{T}^i_{Main}$.
Normalize the number of test cases of each model, and sum the results of five models:
$|\mathcal{T}_{Bert\text{-}only}| = \sum_{i=1}^{5} |\mathcal{T}^i_{Bert} \setminus \mathcal{T}^i_{FIM}|/|\mathcal{T}^i_{Main}|$
$|\mathcal{T}_{FIM\text{-}only}| = \sum_{i=1}^{5} |\mathcal{T}^i_{FIM} \setminus \mathcal{T}^i_{Bert}|/|\mathcal{T}^i_{Main}|$
$|\mathcal{T}_{overlap}| = \sum_{i=1}^{5} |\mathcal{T}^i_{FIM} \cap \mathcal{T}^i_{Bert}|/|\mathcal{T}^i_{Main}|$

Then, we describe the procedure for mutants comparison in Algorithm 4.3. Based on

Algorithm 4.1, we know that for each model i, $\mathcal{T}^i_{Bert}$ can kill all BERTiMuS mutants M^i_{bert} of Simulink model i, $\mathcal{T}^i_{FIM}$ can kill all FIM mutants M^i_{FIM} of Simulink model i. Firstly, we obtain the mutants of each model that can only be killed by $\mathcal{T}_{Bert}$ and $\mathcal{T}_{FIM}$, and can be killed by both. Then we union these set of mutants of five models. Lastly, we get $|M_{k\text{-}Bert}|$, which denotes the number of BERTiMuS mutants that can only be killed by $\mathcal{T}_{Bert}$, and $|M_{k\text{-}FIM}|$, which denotes the number of FIM mutants that can only be killed by $\mathcal{T}_{FIM}$, and $|M_{k\text{-}both}|$, which denotes the number of BERTiMuS and FIM mutants that can be killed by both $\mathcal{T}_{Bert}$ and $\mathcal{T}_{FIM}$. We plot them in Figure 5.2 by output-based mutant killing, fitness-based mutant killing and requirement-based mutant killing methods.

Algorithm 4.3 Experimental procedure for mutants comparison

Input: Five Simulink models
Output: $|M_{k\text{-}Bert}|$, $|M_{k\text{-}FIM}|$ and $|M_{k\text{-}both}|$
for *each model* i **do**
  let $\mathcal{T}^i_{Main}$ be the reference test suite for model i,
  let M^i_{Bert} be the mutants generated for model i by BERTiMuS approach,
  let M^i_{FIM} be the mutants generated for model i by FIM approach,
  use Algorithm 4.1 to obtain $\mathcal{T}^i_{Bert}$ and $\mathcal{T}^i_{FIM}$ based on $\mathcal{T}^i_{Main}$.
  let $M'^{,i}_{Bert}$ and $M'^{,i}_{FIM}$ be the mutants that can be killed by $\mathcal{T}^i_{Bert} \cap \mathcal{T}^i_{FIM}$.
  let $M''^{,i}_{Bert}$ be the bert mutants that can be killed by $\mathcal{T}^i_{Bert} \setminus \mathcal{T}^i_{FIM}$.
  let $M''^{,i}_{FIM}$ be the FIM mutants that can be killed by $\mathcal{T}^i_{FIM} \setminus \mathcal{T}^i_{Bert}$.
Union the set of mutants of five models:
$M'_{Bert} = \cup_{i=1}^{5} M'^{,i}_{Bert}$
$M''_{Bert} = \cup_{i=1}^{5} M''^{,i}_{Bert}$
$M'_{FIM} = \cup_{i=1}^{5} M'^{,i}_{FIM}$
$M''_{FIM} = \cup_{i=1}^{5} M''^{,i}_{FIM}$
Get the total number of mutants:
$|M_{k\text{-}Bert}| = |M''_{Bert}|$
$|M_{k\text{-}FIM}| = |M''_{FIM}|$
$|M_{k\text{-}both}| = |M'_{FIM} \cup M'_{Bert}|$

Chapter 5

Evaluation Results and Analysis

In this section, we present the evaluation results and analysis of the experiments conducted to address the research questions outlined in Section 4.1.

5.1 Evaluation Metrics for RQ1

We begin by outlining some definitions needed to establish our metrics for RQ1:

(1) Number of compiled mutants, which represents the total count of mutants that can be compiled, generated through mutation approaches BERTiMuS and FIM [7].

(2) Number of killed mutants, which denotes how many mutants can be killed among all mutants generated by BERTiMuS and FIM. We recall that mutants can be killed in three ways as discussed in Section 3.5.

(3) Generation time, which refers to the time required to generate a mutant using each approach.

(4) Number of test cases, which denotes the minimal number of test cases that can kill the mutants generated by different approaches using the mutant killing Algorithm 4.1.

(5) Observation effort, which refers to the proportion of non-killed mutants among all compiled mutants. It is important to evaluate each non-killed mutant for equivalence. A lower observation effort is preferable.

(6) Mutation coverage of the other approach, which denotes how many test cases in TSBert/TSFIM can kill mutants generated by the other approach. A higher coverage value indicates that the test suite is more effective in detecting changes in mutants. For example, if test cases can kill mutants generated by BERTiMuS but fail to kill those created by FIM, this indicates that BERTiMuS can produce more effective mutants. Hard-to-kill mutants are a better representation of real faults when the mutations closely mirror typical programming errors and are more likely to uncover problems in Simulink models under realistic conditions.

5.2 RQ1 results

5.2.1 Mutants Generation Information

We provide the generation time taken for each model and each approach in Table **??**. The clock starts at when the program of generating mutants begin, and ends at all mutants are

generated. Note that the generation time for BERTiMuS mutants varies greatly based on the machine's computing power, such as different GPUs or CPU only. From the results, we can tell that by running on a GPU, the generation time per mutant for BERTiMuS is less than that of the FIM tool. This is because BERTiMuS operates directly on text, while FIM works with Simulink and requires identifying specific locations to inject faults. Additionally, the generation time of the FIM tool varies for different models, and models with Stateflow would take more time to generate mutants.

We observe that FIM generates more mutants compared to BERTiMuS. This is because FIM injects all types of faults at every possible position, which does not guarantee high-quality mutations. In contrast, BERTiMuS uses a top-k strategy to control the number of mutations generated. By setting a smaller k, we increase the likelihood of producing higher-quality mutations, as CodeBERT assigns higher probabilities to top predictions. In our experiments, we set k to 3, with the exploration of other k values left for future work.

Furthermore, we provide the number of mutants generated using processing strategies. For the Twotanks model, 9 mutants are created by adding rare mutations. For the ATCS and AECS models, a total of 16 mutants are created by expanding expressions in Stateflow. For the Tustin and ATCS models, 0 mutants are created using processing strategies.

5.2.2 Experimental Results and Analysis

Recall that our experimental procedure in Algorithm 4.1 uses a reference test suite $\mathcal{T}_{main}$ for each Simulink model, and we get test cases $\mathcal{T}_{Bert}$ that are selected by the BERTiMuS mutants only, and test cases $\mathcal{T}_{FIM}$ that are selected by the FIM mutants only. We show

the results using pie charts when applying our experimental procedure described in Algorithm 4.2. Figure 5.1 presents pie charts representing the percentage of $\mathcal{T}_{Bert}$, the percentage of $\mathcal{T}_{FIM}$, and the percentage of test cases that are selected by both FIM and BERTiMuS mutants. These charts illustrate three different ways of mutation testing: classical mutation testing in Figure 5.1(a), fitness-based mutation testing in Figure 5.1(b), and requirements-based mutation testing in Figure 5.1(c). The pie charts in Figure 5.1(a), (b) and (c) show, on average, how many test cases from these reference test suites for all subject models are selected exclusively by BERTiMuS, how many are selected only by FIM, and how many are selected by both approaches.

As Figure 5.1 shows, BERTiMuS can select test cases that cannot be selected by FIM and vice versa, indicating that the mutants generated by the two approaches are complementary. The figure also shows that the test cases that are solely identified by FIM are more than test cases that are solely identified by BERTiMuS, although the difference between the two is smaller when we move from classical mutation testing to requirements-based mutation testing, since the proportion of BERTiMuS is increasing.

To better understand this difference, we need to analyze the total number of mutants generated by both approaches, the number of killed mutants and the number of test cases that Algorithm 4.1 selects for each approach. As shown in Table **??**, FIM generates considerably more mutants compared to BERTiMuS. We further present results in Tables 5.1, 5.2, 5.3 to show the number of mutants killed by BERTiMuS and FIM, the number of test cases required to kill mutants generated by BERTiMuS and FIM, and the observation effort for both techniques, corresponding to classical mutation testing, fitness-based mutation testing and requirements-based mutation testing, respectively. As these tables show,

the number of test cases selected by BERTiMuS is lower than those selected by FIM. This is consistent with the results in Figure 5.1, where the number of FIM-only test cases is more than the number of BERTiMuS-only test cases. As for the observation effort and mutation coverage score, BERTiMuS and FIM achieve equivalent results. Furthermore, as we move from classical mutation testing to requirement-based mutation testing, the mutation coverage score decreases, consistent with findings from previous Simulink mutation studies [7, 8].

For both FIM and BERTiMuS, several mutants could be killed by a single test case leading to the selection of a small number of test cases by both approaches, i.e., 19/120 for BERTiMuS and 38/120 by FIM. This finding is consistent with those from a recent study on mutation testing for Simulink [8]. In addition, we expect the number of killed mutants and the number of selected test cases to decrease as we move from classical to requirements-based mutation testing. This observation is also consistent with the recent Simulink mutation study [8]. This is because classical mutation testing primarily focuses on exercising a fault and easily reveals the impact of that fault through changes in the output. However, revealing a fault based on requirements violation is more challenging, as it depends on changes in the polarity of fitness values.

What is interesting and new in our study is that the gap between the number of BERTiMuS-only test cases ($\mathcal{T}_{Bert}$) and the number of FIM-only test cases ($\mathcal{T}_{FIM}$) reduces as we move from classical to requirements-based mutation testing. As shown in Figure 5.1, the normalized proportion of the test cases generated by BERTiMuS increases from 16.1% to 20.0% and 28.0%. This reinforces the idea that a larger proportion of the faults generated by BERTiMuS are relevant to the requirements, and hence, are more meaningful

to the modelers and engineers.

We also show the results using pie charts when applying our experimental procedure described in Algorithm 4.3 in Figure 5.2. This figure shows the proportion of mutants that can be killed by test cases $\mathcal{T}_{Bert}$ only and test cases $\mathcal{T}_{FIM}$ only, and the proportion of mutants that can be killed by both. As the figure shows, the test cases selected by BERTiMuS ($\mathcal{T}_{Bert}$) can kill more BERTiMuS mutants compared to the test cases selected by FIM ($\mathcal{T}_{FIM}$), and the proportion of mutants killed by $\mathcal{T}_{Bert}$ grows from classical mutation testing to requirements-based mutation testing. This further enhances the idea that test cases selected by BERTiMuS are more effective in killing mutants, and the faults generated by BERTiMuS are more relevant to the requirements compared to FIM approach.

In summary, our study suggests that BERTiMuS is complementary to the existing mutation generation baseline for Simulink models [7], as it leads to identifying test cases that could not be selected by FIM-generated mutants. Additionally, when considering the more meaningful notion of mutation testing for Simulink models, i.e., requirements-based mutation testing, the proportion of test cases that BERTiMuS identifies compared to the baseline increases. This suggests that the faults identified by BERTiMuS are overall more relevant to system requirements compared to those generated by the state-of-the-art baseline. We qualitatively investigate the meaning and generation process of the faults generated by BERTiMuS in RQ2. We show the experimental results of each Simulink model in Appendix 7.1.

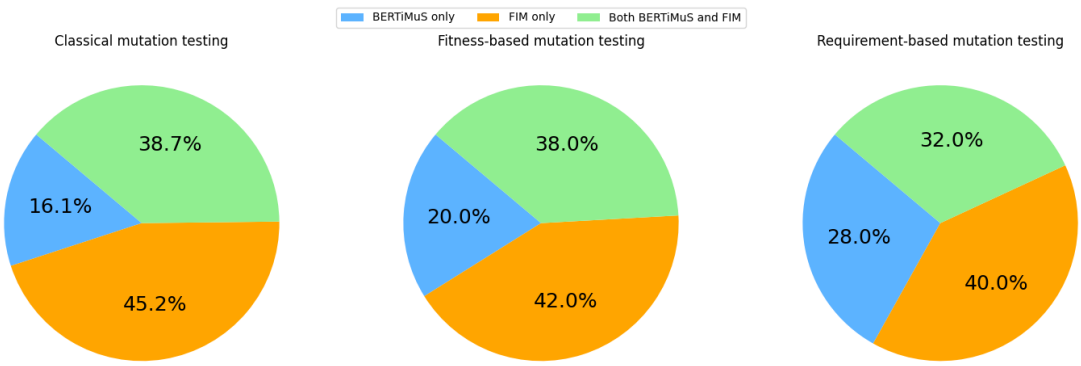


Figure 5.1: Overlaps between test cases selected by BERTiMuS and FIM for the different notions of mutation testing presented in Section 3.5.

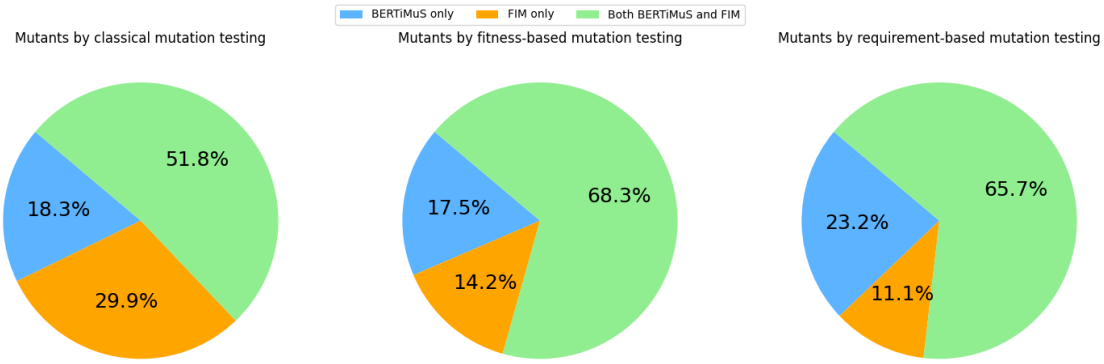


Figure 5.2: Overlaps between mutants generated by BERTiMuS and FIM for the different notions of mutation testing presented in Section 3.5.

Table 5.1: Results of total five Simulink models on output-based mutation testing based on metrics in Section 5.1

	BERTiMuS	FIM
Number of compiled mutants	387	594
Number of killed mutants	282	361
Observation effort	27.1%	33.2%
Number of test cases in $\mathcal{T}_{Bert}/\mathcal{T}_{FIM}$	19/120	38/120
Mutation score of the other approach	71.7%	85.0%

Table 5.2: Results of total five Simulink models on fitness-based mutation testing based on metrics in Section 5.1

	BERTiMuS	FIM
Number of compiled mutants	315	496
Number of killed mutants	239	315
Observation effort	22.7%	27.8%
Number of test cases in $\mathcal{T}_{Bert}/\mathcal{T}_{FIM}$	17/120	35/120
Mutation score of the other approach	69.4%	60.8%

Table 5.3: Results of total five Simulink models on requirement-based mutation testing based on metrics in Section 5.1

	BERTiMuS	FIM
Number of compiled mutants	315	496
Number of killed mutants	146	191
Observation effort	21.2%	20.2%
Number of test cases in $\mathcal{T}_{Bert}/\mathcal{T}_{FIM}$	9/120	14/120
Mutation score of the other approach	50.0%	50.8%

Table 5.4: Existing Simulink fault patterns and their sources: Green cells show patterns supported by BERTiMuS; yellow cells show unsupported patterns

Pattern	Pattern	Pattern
Mutate signal data types [45]	Mutate GoTo/From blocks [45]	Mutate "Saturate on integer overflow" [45]
Mutate constant and gain values [11, 26, 31, 39, 45]	Mutate math, relational and logical operator blocks [11, 26, 31, 39, 45]	Mutate initial conditions and sample time [11, 45]
Mutate Stateflow transition conditions [26, 31, 39, 45]	Mutate Stateflow variable names [11, 39, 45]	Mutate Stateflow actions [11, 31, 39, 45]
Mutate Stateflow keywords [11, 45]	Replace a scalar value with a scalar, or replace an array with an array [31]	Insert absolute value operator [31]
Remove or add connections between blocks [7]	Swap blocks [31]	Remove blocks [39]

5.3 RQ2 results

Table 5.4 summarizes 15 major mutation patterns for Simulink models that we have collected by extensively surveying the literature [11, 26, 39, 45, 53, 55]. To address RQ2, we compare the mutants generated by BERTiMuS with the mutation patterns listed in Table 5.4. This comparison aims to determine which BERTiMuS's mutants could be generated based on these patterns, which could not, and whether there are any mutation patterns that BERTiMuS cannot cover. Our analysis showed that all the 387 mutants generated by BERTiMuS could be generated by the 11 patterns in Table 5.4 highlighted in green, and yet BERTiMuS cannot generate mutants related to the four patterns highlighted in yellow in this table. The four patterns that cannot be covered by BERTiMuS require the mutation operator to change links between blocks, modify an entire block, operate over multiple blocks. These are beyond the capability of BERTiMuS , which is limited to changing property values inside the blocks.

While in terms of the mutation patterns, BERTiMuS is limited to the already known

mutation patterns, our investigation shows that BERTiMuS, due to its pretrained knowledge of Simulink structure and its access to the context at the time of masking, BERTiMuS generates mutant instances that are unlikely to be generated based on syntactic and manual mutation generation rules. In particular, we note that while in code such as Java and Python, mutation operators are derived based on the grammar of the targeted programming language [53], however, for Simulink, mutation operators are often manually coded following the pattern descriptions due to lack of an explicit grammar for the Simulink syntax, thus making it hard to generate complex and high-quality mutations.

We highlight three main points to illustrate the advantages of BERTiMuS:

1. Ability to change a variable name to another variable name that appeared in the blocks of Simulink model, based on the block information in the context. Since BERTiMuS takes all block information as input, it can easily change the current variable name to another variable that exists in the Simulink model, thus performing actions on another block, but if the mutation result is just a meaningless string, then it would disable the blocks or actions related to this variable name, which is considered as a low-quality mutation. In contrast, if we require humans to produce variable names that appeared in the Simulink model, they would need to examine the Simulink model, which would require effort.

   For example, in a car engine model, BERTiMuS may change variable name 'engine_speed' with 'gear_ratio' that appeared in other blocks of the same model. This would change the behaviour of the system: Control systems designed to respond to engine speed would now receive data about gear ratios. Now, a system that

adjusts fuel injection based on engine speed could erroneously adjust based on gear ratio, leading to inefficient combustion and potentially damaging the engine. Also, performance metrics related to 'engine_speed' would give incorrect results for such a mutation.

Another example would be changing the action 'send(LO)' to 'send(LI)' in Stateflow. The send keyword is used to dispatch an event. Changing the event can trigger transitions or actions within the Stateflow chart, thus altering the flow or behavior of the Stateflow model according to the newly specified event's associated logic or transitions.

2. Ability to predict property values according to the block type, such as predicting '==' or '>=' if the block type is 'Operator', predicting 'int8' or 'float' if the block type is 'OutDataTypeStr', and predicting numerals or array such as '0.5', '100' and '[0,1,5]' if the block type is 'Constant' and 'Gain'. This ability can also be attributed to BERTiMuS's capacity to learn from the pre-training Simulink corpus. In contrast to the naive baseline, it would require human effort and sufficient expert knowledge to generate similiar mutations.

3. Ability to expand expressions in Stateflow. We use the pre-processing strategies in Section 3.4.1 to expand expressions for Stateflow model, which improves the quality and diversity of the mutations, making them more closely resemble real-world scenarios. For example, we can expand on a simple expression like 'a==1' to a more complex form such as 'a==1 || b==2'. This expansion not only maintains the original intent of the expression but also enriches it, providing a deeper, more realistic

context for simulation and testing. The result is a more robust and versatile model, better equipped to handle the complexities of real-world applications.

In summary, we show that BERTiMuS can support 11 out of 15 common Simulink faults patterns. Moreover, we list three advantages to prove that BERTiMuS can generate high-quality mutants using these fault patterns. We further compare BERTiMuS with the naive baseline qualitatively. For humans, to generate similar mutations need programming effort and expert knowledge of computer science and Simulink.

5.4 Threats to Validity

The most pertinent aspects of validity for our evaluation are construct validity, external validity and conclusion validity.

With regard to *construct validity*, we note that the metrics we used for evaluation, such as the total number of mutants analyzed and the number of test cases selected, may not fully reflect the actual costs or effectiveness of the testing process. Nonetheless, these metrics are commonly used in the literature [3, 16, 37, 53] due to their intuitive nature. The number of mutants analyzed approximates the manual labour required by testers, and the effectiveness of the testing can be gauged by the ability of the test suites to identify and kill these mutants.

With regard to *external validity*, we note that our experimental results are based on five Simulink models only. To reduce this threat, we tried to be as representative as possible in the selection of these models [48]: As discussed in Section 4.2, Tustin is a value-related

model, Twotanks is a Gototag-related model, FSM is a state-related model, ATCS is a Stateflow model, and AECS is a Stateflow model associated with Gototag. These different types of Simulink models cover the popular types used in real applications. That being said, we recognize that more comprehensive case studies are required and that the outcomes may vary for models from different domains.

Finally, with regard to *conclusion validity*, we note that the selection of our baseline for comparison could pose threats to the validity of our results. While many mutation testing approaches have been proposed, FIM [7] represents the most recent work specifically targeting Simulink. As such, we believe that FIM is the most appropriate comparison baseline to evaluate our work against. Furthermore, we adopt a naive baseline that utilizes the same mutation operators used by humans to demonstrate that BERTiMuS achieves significantly higher efficiency. We recognize that to draw more definitive conclusions, we need to consider a wider variety of baselines, but doing so is beyond the scope of this thesis and is left for future work.

Chapter 6

Related Work

In this chapter, we review previous research in the areas of mutation testing, mutation testing for Simulink models, and mutation testing using large language models.

Mutation testing is a method for assessing the effectiveness of test cases by introducing faults and has been a major area of research in software engineering. Initially introduced by Richard Lipton in the early 1970s, the concept was further developed by DeMillo et al. [17], who examined the efficiency of mutation testing as a tool for evaluating the effectiveness of test cases, particularly under the common industry constraints of limited time and budget for testing activities. Mutation testing emphasizes the economic and practical challenges faced by programmers in conducting thorough testing, presenting a solution that balances cost with the need for comprehensive bug detection. By providing insights into the diminishing returns of bug discovery in the later stages of the testing cycle, DeMillo et al. advocate for a focused application of mutation testing to optimize the identification

of critical errors early on. Jia et al. [31] present a comprehensive review and examination of mutation testing's evolution. The authors thoroughly explore the theoretical foundations, practical implementations, and advancements in mutation testing, tracing its progression from a conceptual framework to an essential tool in the software testing arsenal. They discuss the challenges, methodologies, and the significant impact of mutation testing on improving software reliability and testing effectiveness. This survey serves as a cornerstone for understanding the evolution and impact of mutation testing in software testing.

More recently, the advancements and new directions in mutation testing have been discussed by Papadakis et al. [53]. The authors provide a thorough examination of the development of new mutation operators, frameworks for automated mutation test generation, and empirical studies assessing the impact of mutation testing on software quality. They study the intricacies of various mutation testing tools and methodologies, discussing their applicability, efficiency, and effectiveness in different programming languages and environments. This work also addresses the challenges faced in mutation testing, such as the computational cost associated with generating and testing a large number of mutants, and proposes strategies to mitigate these challenges, including selective mutation testing and the use of sophisticated analysis techniques to reduce the number of mutants.

Many research papers, e.g., [35, 38, 42, 45, 51], have studied the design of fault patterns or mutation operators in order to generate mutants for specific purposes and to reduce the need for mutants. Some work strands [42], [45], [38] develop patterns for mutant generation based on the grammar of the underlying programming language, and then select from the generated mutations through follow-up analysis [35], [51] or through metrics [8] to eliminate redundant or invalid mutations.

Despite the limitations of pattern-based mutation generation [38, 42, 45], which include the need for expert design, a limited range of mutation types, and potential lack of alignment with real-world scenarios, recent studies have focused on generating high-quality mutations that reveal faults. Defects4J [33] provides a large database of real, reproducible bugs from various open-source Java projects, enabling researchers to conduct controlled testing studies with actual faults rather than hypothetical ones. Brown et al. [12] propose generating mutations based on real bug fixes rather than merely on the programming language grammar. Tufano et al. [59] propose using a neural machine translation method to train a neural network to inject faults by learning from real bug fixes. Although the above-mentioned papers generate mutation operators based on the known fault space and produce mutants of more fault patterns and higher quality, they heavily rely on large-scale and untangled bug fix commits [27], which may be difficult to obtain for industrial Simulink models due to the absence of large-scale public repositories.

Several research papers use LLMs for mutation testing. This is done in two main ways: (1) MLM-based, using infilling LLMs such as BERT. The strategy here is to mask certain tokens and ask the model to predict new results to replace the masked tokens. (2) Prompt-based, using generative LLMs such as GPT-4 [2] and LLaMA [58]. Here, the strategy is to instruct the mutation of input code through prompting.

With regard to MLM-based LLMs for mutation testing, we have identified three studies which we describe below: μBERT [16] employs a pre-trained language model, CodeBERT [21], to mutate Java programs. μBERT provides the following benefits: First, using μBERT to generate mutants eliminates the manual effort of designing fault patterns. Second, μBERT generates more realistic and higher quality mutations, based on its learning

ability from a large code corpus and code semantic context. The code generated by μBERT is more 'natural', which means that the modified code or statement adheres to the implicit norms, coding standards, and is typically indicative of the code written by proficient programmers. Khanfir et al. [34] adopt Abstract Syntax Tree (AST) to parse the code and locate important code to mutate. The authors further apply a method named condition-seeding additive mutation to μBERT to mutate more than one token at a time, so as to generate more diverse and complex mutations, such as expanding expressions. This approach has been shown to find more bugs than μBERT alone. Deng et al. [18] employ both generative and infilling LLMs to create diverse input DL programs for fuzz testing DL libraries. Specifically, they initially utilize a generative LLM to produce a collection of seed programs, such as code snippets that use the targeted DL APIs. Subsequently, they substitute part of these seed programs with masked tokens, apply various mutation operators, and capitalize on the code-prediction capabilities of an infilling LLM to generate new code snippets.

As for prompt-based LLMs for mutation testing, we have identified three recent papers [13, 15, 41]. Dakhel et al. [15] introduce MuTAP, which has been developed to enhance the capability of test cases generated by LLMs in uncovering bugs. This approach employs mutation testing to strengthen test suites. Specifically, prompts are augmented with surviving mutants, thereby emphasizing areas where test cases fail to identify defects. Brownlee et al. [13] propose using LLMs as mutation operators for program repair based on search-based techniques. LLMs allow for the generation of more diverse and complex code that can better simulate real-world scenarios. Liu et al. [41] use LLMs to generate test inputs together with mutation rules for text-oriented fuzzing in an attempt to reduce

the human effort necessary for designing mutation rules.

Many research papers have focused on mutation testing for Simulink models, aimed at seeding faults into Simulink models using mutation operators. SIMULATE [57] uses XML-defined fault models and minimal cut set generation to assess system robustness against single and multiple faults in Simulink. Matinnejad et al. [45] introduce a black-box test generation technique using meta-heuristic search to increase the diversity of test outputs, coupled with a test prioritization algorithm that ranks tests based on their fault-revealing potential. This approach aims to mitigate common challenges such as incompatibility, oracle, and scalability issues in Simulink testing. The technique is shown to outperform traditional testing through evaluation with industrial Simulink models, thus offering a more efficient and cost-effective solution for engineers working within a limited test budget. FIM [7] presents a toolkit for enhancing Simulink model testing by allowing automated injection of faults and generation of mutants. FIM supports various fault types and mutation operators with customizable timing and duration settings. This approach enables targeted fault activation for impact analysis on system reliability. The toolkit's design, architecture, and utility are studied through an avionics case study, demonstrating FIM's effectiveness in improving model robustness. Property-based mutation testing (PBMT) [8] is another approach to Simulink mutation testing. It posits that a mutant which does not affect a property should not contribute to measuring the adequacy of a test suite against that property. However, regular mutation testing does not distinguish between these mutants. PBMT works by selecting only those mutants that affect the software's adherence to defined properties, ensuring tests are more meaningful by requiring a mutant's impact to cause a property violation for it to be considered successfully killed. This approach

improves mutation testing by aligning test effectiveness closely with property satisfaction, offering a more targeted and efficient assessment of test suite quality.

To our knowledge, none of the methods in the existing literature use LLMs for Simulink mutation testing. In our work, we use a MLM-based mutation testing method, similar to Khanfir et al. [34] but applied to Simulink. We use both classical mutation testing [53] and property-based mutation testing [8] to benchmark our approach against FIM [7].

Chapter 7

Conclusions

In this thesis, we introduced a novel approach, BERTiMuS, for mutation testing of Simulink models by leveraging the capabilities of pre-trained language models, specifically through the development and application of CodeBERT. By converting Simulink models into a textual format and pre-training CodeBERT with a large Simulink corpus, BERTiMuS enhances the efficiency and quality of mutation testing of Simulink models. The mutations generated by BERTiMuS tend to be less prone to basic errors and more reflective of natural modifications that may occur in the development process. We also developed several post-processing steps to further improve the quality and diversity of the generated mutants. Through rigorous experimentation and comparison with both traditional Simulink mutation methods and human-driven mutation techniques, we show that our method has superior performance in detecting requirements violations, offering a more effective means of evaluating and improving the robustness of Simulink models. Furthermore, our approach has advantages over human mutation methods, including consistency, scalability,

and the ability to leverage deep, contextualized knowledge embedded within pre-trained language models. This research not only provides a practical tool for software engineering testing but also opens new avenues for the application of pre-trained language models in the analysis and testing of block-based languages.

7.1 Future Work

In the future, we would like to enhance and further explore this work in the following directions:

Firstly, we plan to experiment with more Simulink models of different types such as Stateflow, mathematical computations, and others to validate the generalizability of our approach. There are also more experiments needed to validate, such as the effects of pre-processing and post-processing strategies, the settings of top-k, and comparison to more baselines.

Secondly, since LLMs can take increasingly long inputs, such as 64k tokens, and also have powerful understanding and instruction-following abilities, a promising direction for future work is to input the textual representation of Simulink models into LLMs to generate more complex mutations.

Finally, we would like to expand our mutation operators. Currently, our work focuses only on mutations inside a block, while ignoring the connections among blocks and block-level mutations. By addressing this limitation, we believe we will obtain a more comprehensive and useful approach.

References

[1] Lockheed Martin, 2024. https://www.lockheedmartin.com/en-us/index.html.

[2] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[3] James H Andrews, Lionel C Briand, Yvan Labiche, and Akbar Siami Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, 32(8):608–624, 2006.

[4] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[5] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[6] Luciano Baresi and Mauro Pezze. An introduction to software testing. *Electronic Notes in Theoretical Computer Science*, 148(1):89–111, 2006.

[7] Ezio Bartocci, Leonardo Mariani, Dejan Ničković, and Drishti Yadav. Fim: fault injection and mutation for simulink. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1716–1720, 2022.

[8] Ezio Bartocci, Leonardo Mariani, Dejan Ničković, and Drishti Yadav. Property-based mutation testing. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 222–233. IEEE, 2023.

[9] Moritz Beller, Chu-Pan Wong, Johannes Bader, Andrew Scott, Mateusz Machalica, Satish Chandra, and Erik Meijer. What it would take to use mutation testing in industry—a study at facebook. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 268–277. IEEE, 2021.

[10] Yoshua Bengio, Réjean Ducharme, and Pascal Vincent. A neural probabilistic language model. *Advances in neural information processing systems*, 13, 2000.

[11] Nguyen Thanh Binh et al. Mutation operators for simulink models. In *2012 Fourth International Conference on Knowledge and Systems Engineering*, pages 54–59. IEEE, 2012.

[12] David Bingham Brown, Michael Vaughn, Ben Liblit, and Thomas Reps. The care and feeding of wild-caught mutants. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 511–522, 2017.

[13] Alexander EI Brownlee, James Callan, Karine Even-Mendoza, Alina Geiger, Carol Hanna, Justyna Petke, Federica Sarro, and Dominik Sobania. Enhancing genetic improvement mutations using large language models. In *International Symposium on Search Based Software Engineering*, pages 153–159. Springer, 2023.

[14] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony 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.

[15] Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C Desmarais. Effective test generation using pre-trained large language models and mutation testing. *Information and Software Technology*, page 107468, 2024.

[16] Renzo Degiovanni and Mike Papadakis. μbert: Mutation testing using pre-trained language models. In *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 160–169. IEEE, 2022.

[17] Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[18] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*, pages 423–435, 2023.

[19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[20] Hamza Ed-Douibi, Javier Luis Cánovas Izquierdo, and Jordi Cabot. Automatic generation of test cases for rest apis: A specification-based approach. In *2018 IEEE 22nd international enterprise distributed object computing conference (EDOC)*, pages 181–190. IEEE, 2018.

[21] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[22] Gordon Fraser and Andrea Arcuri. Evolutionary generation of whole test suites. In *2011 11th International Conference on Quality Software*, pages 31–40. IEEE, 2011.

[23] Rahul Gopinath, Carlos Jensen, and Alex Groce. Mutations: How close are they to real faults? In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 189–200. IEEE, 2014.

[24] Farah Hariri, August Shi, Hayes Converse, Sarfraz Khurshid, and Darko Marinov. Evaluating the effects of compiler optimizations on mutation testing at the compiler ir level. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 105–115. IEEE, 2016.

[25] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[26] Nannan He, Philipp Rümmer, and Daniel Kroening. Test-case generation for embedded simulink via formal concept analysis. In *Proceedings of the 48th Design Automation Conference*, pages 224–229, 2011.

[27] Kim Herzig and Andreas Zeller. Untangling changes. *Unpublished manuscript, September*, 37:38–40, 2011.

[28] John J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8):2554–2558, 1982.

[29] Nadjim Horri and Mikolaj Pietraszko. A tutorial and review on flight control co-simulation using matlab/simulink and flight simulators. *Automation*, 3(3):486–510, 2022.

[30] Krisztián Horváth, Marton Kuslits, and Szilard Lovas. Model-based control algorithm development of induction machines by using a well-defined model architecture and rapid control prototyping. *Electrical Engineering*, 102(3):1103–1116, 2020.

[31] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2010.

[32] René Just. The major mutation framework: Efficient and scalable mutation analysis for java. In *Proceedings of the 2014 international symposium on software testing and analysis*, pages 433–436, 2014.

[33] René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*, pages 437–440, 2014.

[34] Ahmed Khanfir, Renzo Degiovanni, Mike Papadakis, and Yves Le Traon. Efficient mutation testing via pre-trained language models. *arXiv preprint arXiv:2301.03543*, 2023.

[35] Marinos Kintis, Mike Papadakis, Andreas Papadopoulos, Evangelos Valvis, Nicos Malevris, and Yves Le Traon. How effective are mutation testing tools? an empirical analysis of java mutation testing tools with manual analysis and real faults. *Empirical Software Engineering*, 23:2426–2463, 2018.

[36] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[37] Bob Kurtz, Paul Ammann, Jeff Offutt, Márcio E Delamaro, Mariet Kurtz, and Nida Gökçe. Analyzing the validity of selective mutation with dominator mutants. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 571–582, 2016.

[38] Thomas Laurent, Mike Papadakis, Marinos Kintis, Christopher Henard, Yves Le Traon, and Anthony Ventresque. Assessing and improving the mutation testing

practice of pit. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 430–435. IEEE, 2017.

[39] Khuat Thanh Le Thi My Hanh and Nguyen Thanh Binh Tung. Mutation-based test data generation for simulink models using genetic algorithm and simulated annealing. *International Journal of Computer and Information Technology*, 3(04):763–771, 2014.

[40] Bing Liu, Shiva Nejati, Lucia, and Lionel C Briand. Effective fault localization of automotive simulink models: achieving the trade-off between test oracle effort and fault localization accuracy. *Empirical Software Engineering*, 24:444–490, 2019.

[41] Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Zhilin Tian, Yuekai Huang, Jun Hu, and Qing Wang. Testing the limits: Unusual text inputs generation for mobile app crash detection with large language model. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–12, 2024.

[42] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. Mujava: an automated class mutation system. *Software Testing, Verification and Reliability*, 15(2):97–133, 2005.

[43] MathWorks. Simulink. http://www.mathworks.nl/products/simulink/.

[44] MathWorks. Stateflow. https://www.mathworks.com/products/stateflow.html.

[45] Reza Matinnejad, Shiva Nejati, Lionel C Briand, and Thomas Bruckmann. Test generation and test prioritization for simulink models with dynamic behavior. *IEEE Transactions on Software Engineering*, 45(9):919–944, 2018.

[46] Phil McMinn. Search-based software test data generation: a survey. *Software testing, Verification and reliability*, 14(2):105–156, 2004.

[47] Shiva Nejati, Khouloud Gaaloul, Claudio Menghi, Lionel C Briand, Stephen Foster, and David Wolfe. Evaluating model testing and model checking for finding requirements violations in simulink models. In *Proceedings of the 2019 27th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 1015–1025, 2019.

[48] Shiva Nejati, Khouloud Gaaloul, Claudio Menghi, Lionel C. Briand, Stephen Foster, and David Wolfe. Evaluating model testing and model checking for finding requirements violations in simulink models. In Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo, editors, *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, pages 1015–1025. ACM, 2019.

[49] A Jefferson Offutt and Jie Pan. Automatically detecting equivalent mutants and infeasible paths. *Software testing, verification and reliability*, 7(3):165–192, 1997.

[50] A Jefferson Offutt and Roland H Untch. Mutation 2000: Uniting the orthogonal. *Mutation testing for the new century*, pages 34–44, 2001.

[51] Milos Ojdanic, Aayush Garg, Ahmed Khanfir, Renzo Degiovanni, Mike Papadakis, and Yves Le Traon. Syntactic vs. semantic similarity of artificial and real faults in mutation testing studies. *IEEE Transactions on Software Engineering*, 2023.

[52] Papadakis. Mutation testing publications. https://mutationtesting.uni.lu/tools.php.

[53] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Mutation testing advances: an analysis and survey. In *Advances in Computers*, volume 112, pages 275–378. Elsevier, 2019.

[54] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

[55] Ajitha Rajan, Michael Whalen, Matt Staats, and Mats PE Heimdahl. Requirements coverage as an adequacy measure for conformance testing. In *International Conference on Formal Engineering Methods*, pages 86–104. Springer, 2008.

[56] Chayanika Sharma, Sangeeta Sabharwal, and Ritu Sibal. A survey on software testing techniques using genetic algorithm. *arXiv preprint arXiv:1411.1154*, 2014.

[57] Rickard Svenningsson, Jonny Vinter, Henrik Eriksson, and Martin Törngren. Modifi: a model-implemented fault injection tool. In *Computer Safety, Reliability, and Security: 29th International Conference, SAFECOMP 2010, Vienna, Austria, September 14-17, 2010. Proceedings 29*, pages 210–222. Springer, 2010.

[58] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[59] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. Learning how to mutate source code from bug-fixes. In

2019 IEEE International conference on software maintenance and evolution (ICSME), pages 301–312. IEEE, 2019.

[60] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[61] Bo Wang, Yingfei Xiong, Yangqingwei Shi, Lu Zhang, and Dan Hao. Faster mutation analysis via equivalence modulo states. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 295–306, 2017.

[62] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*, pages 38–45, 2020.

APPENDICES

Effects of Different Mutants on Tustin Model's Properties

To demonstrate the effects of different mutations on the Tustin model's properties, we selected a small and simple model named Tustin and displayed the results in Table 1. In the table, the "Block Mutated" column indicates the original and mutated values in the Simulink model. "Compile" shows whether the mutant can compile, with 1 denoting successful compilation. "R1a-R4b" represent the fitness functions of the Tustin model, where 1 indicates a change in the fitness value compared to the original, and 0 means the value remains unchanged. We list the descriptions of the four requirements of the Tustin model below; for details of the Tustin model, readers are referred to section 4.2.

Table 1: Effects of different mutants on Tustin model's properties

IDX	SID	Block Mutated	compile	R1a	R1b	R1c	R1d	R1e	R2a	R2b	R3	R4a	R4b
1	155	SampleTime:0 → 1	1	0	0	0	0	0	0	1	1	1	1
2	155	SampleTime:0 → D	0	0	0	0	0	0	0	0	0	0	0
3	141	GotoTag:TL → T	1	1	0	1	0	1	0	1	1	1	1
4	141	GotoTag:TL → 1	1	1	0	1	0	1	0	0	1	1	1
5	142	GotoTag:BL → 1	1	0	1	0	1	1	0	1	1	1	1
6	142	GotoTag:BL → 1	1	0	1	1	1	0	0	0	1	1	1
7	91	Gain:.5 → -1	1	1	1	1	1	1	1	1	1	1	1
8	91	Gain:.5 → K01	0	0	0	0	0	0	0	0	0	0	0
9	91	Gain:.5 → K02	0	0	0	0	0	0	0	0	0	0	0
10	91	SaturateOnIntegerOverflow:off → on	1	1	1	1	1	1	1	1	1	1	1
11	132	GotoTag:TL → T	1	1	0	1	1	1	1	0	1	1	1
12	132	GotoTag:TL → TS	1	1	0	1	0	1	0	0	1	1	1
13	140	GotoTag:BL → BR	1	0	1	0	1	1	0	1	1	1	1
14	140	GotoTag:BL → TL	0	0	0	0	0	0	0	0	0	0	0
15	92	SaturateOnIntegerOverflow:off → on	1	1	1	1	1	1	1	1	1	1	1
16	94	Inputs:|++ → +-	1	1	1	1	1	1	1	1	1	1	1
17	94	Inputs:|++ → ++-+	1	1	1	1	1	1	1	1	1	1	1
18	94	SaturateOnIntegerOverflow:off → on	1	1	1	1	1	1	1	1	1	1	1
19	95	SaturateOnIntegerOverflow:off → on	1	1	1	1	1	1	1	1	1	1	1
20	96	Threshold:0.5 → 0.5	1	1	1	1	1	1	1	1	1	1	1
21	96	Threshold:0.5 → 5	1	1	1	1	1	1	1	1	1	1	1
22	96	SaturateOnIntegerOverflow:off	1	1	1	1	1	1	1	1	1	1	1
23	126	SampleTime:-1 → +01	0	0	0	0	0	0	0	0	0	0	0
24	126	SampleTime:-1 → 12	0	0	0	0	0	0	0	0	0	0	0
25	127	SampleTime:0 → +01	0	0	0	0	0	0	0	0	0	0	0
26	127	SampleTime:0 → 02	0	0	0	0	0	0	0	0	0	0	0
27	129	Operator:< → ==	1	1	1	1	0	1	1	1	1	1	1
28	129	Operator:< → <=	1	1	1	1	1	1	1	1	1	1	1
29	129	Operator:< → >	1	0	0	0	0	0	1	1	1	1	1
30	128	Threshold:0.5 → 0.5	1	1	1	1	1	1	1	1	1	1	1
31	128	Threshold:0.5 → 1.0	1	1	1	1	1	1	1	1	1	1	1
32	128	Threshold:0.5 → 225	1	1	1	1	1	1	1	1	1	1	1
33	128	SaturateOnIntegerOverflow:off → on	1	1	1	1	1	1	1	1	1	1	1
34	137	Threshold:0.5 → 0.5	1	1	1	1	1	1	1	1	1	1	1
35	137	Threshold:0.5 → 1.0	1	1	1	1	1	1	1	1	1	1	1
36	137	Threshold:0.5 → 225	1	1	1	1	1	1	1	1	1	1	1
37	137	SaturateOnIntegerOverflow:off → on	1	1	1	1	1	1	1	1	1	1	1

Experimental result for each Simulink model

In this part, we show the results of five Simulink models: Tustin, Twotanks, FSM, ATCS and AECS, and we run each model based on three mutant-killing ways: output-based, fitness-based and requirement-based. To be noted, all mutants of FSM model cannot be killed by requirement measurement, all mutants of AECS model cannot be killed by fitness and requirement measurement.

Table 2: Result of Tustin model on output-based mutation testing based on metrics in Section 5.1

	BERTiMuS	FIM
number of compiled mutants	22	55
number of killed mutants	15	47
observation effort	7/22=31.82%	8/55=14.55%
number of test cases in $\mathcal{T}_{Bert}/\mathcal{T}_{FIM}$	3	12
Mutation score of the other approach	3/3=100.0%	12/12=100.0%

Table 3: Result of Tustin model on fitness-based mutation testing based on metrics in Section 5.1

	BERTiMuS	FIM
number of compiled mutants	22	55
number of killed mutants	15	47
observation effort	7/22=31.82%	8/55=14.55%
number of test cases in $\mathcal{T}_{Bert}/\mathcal{T}_{FIM}$	3	12
Mutation score of the other approach	3/3=100.0%	12/12=100.0%

Table 4: Result of Tustin model on requirement-based mutation testing based on metrics in Section 5.1

	BERTiMuS	FIM
number of compiled mutants	22	55
number of killed mutants	13	45
observation effort	9/22=40.91%	10/55=18.18%
number of test cases in $\mathcal{T}_{Bert}/\mathcal{T}_{FIM}$	3	10
Mutation score of the other approach	3/3=100.0%	7/10=70.0%

Table 5: Result of Twotanks model on output-based mutation testing based on metrics in Section 5.1

	BERTiMuS	FIM
number of compiled mutants	154	176
number of killed mutants	115	81
observation effort	39/154=25.32%	95/176=53.98%
number of test cases in $\mathcal{T}_{Bert}/\mathcal{T}_{FIM}$	1	3
Mutation score of the other approach	1/1=100.0%	1/3=33.33%

Table 6: Result of Twotanks model on fitness-based mutation testing based on metrics in Section 5.1

	BERTiMuS	FIM
number of compiled mutants	154	176
number of killed mutants	115	81
observation effort	39/154=25.32%	95/176=53.98%
number of test cases in $\mathcal{T}_{Bert}/\mathcal{T}_{FIM}$	1	3
Mutation score of the other approach	1/1=100.0%	1/3=33.33%

Table 7: Result of Twotanks model on requirement-based mutation testing based on metrics in Section 5.1

	BERTiMuS	FIM
number of compiled mutants	154	176
number of killed mutants	94	68
observation effort	60/154=38.96%	108/176=61.36%
number of test cases in $\mathcal{T}_{Bert}/\mathcal{T}_{FIM}$	3	3
Mutation score of the other approach	1/3=33.33%	1/3=33.33%

Table 8: Result of ATCS model on output-based mutation testing based on metrics in Section 5.1

	BERTiMuS	FIM
number of compiled mutants	41	79
number of killed mutants	39	78
observation effort	2/41=4.88%	1/79=1.27%
number of test cases in $\mathcal{T}_{Bert}/\mathcal{T}_{FIM}$	3	1
Mutation score of the other approach	2/3=66.67%	1/1=100.0%

Table 9: Result of ATCS model on fitness-based mutation testing based on metrics in Section 5.1

	BERTiMuS	FIM
number of compiled mutants	41	79
number of killed mutants	39	78
observation effort	2/41=4.88%	1/79=1.27%
number of test cases in $\mathcal{T}_{Bert}/\mathcal{T}_{FIM}$	3	1
Mutation score of the other approach	2/3=66.67%	1/1=100.0%

Table 10: Result of ATCS model on requirement-based mutation testing based on metrics in Section 5.1

	BERTiMuS	FIM
number of compiled mutants	41	79
number of killed mutants	39	78
observation effort	2/41=4.88%	1/79=1.27%
number of test cases in $\mathcal{T}_{Bert}/\mathcal{T}_{FIM}$	3	1
Mutation score of the other approach	2/3=66.67%	1/1=100.0%

Table 11: Result of FSM model on output-based mutation testing based on metrics in Section 5.1

	BERTiMuS	FIM
number of compiled mutants	98	186
number of killed mutants	70	109
observation effort	28/98=28.57%	77/186=41.4%
number of test cases in $\mathcal{T}_{Bert}/\mathcal{T}_{FIM}$	9	10
Mutation score of the other approach	1/9=11.11%	1/10=10.0%

Table 12: Result of FSM model on fitness-based mutation testing based on metrics in Section 5.1

	BERTiMuS	FIM
number of compiled mutants	98	186
number of killed mutants	70	109
observation effort	28/98=28.57%	77/186=41.4%
number of test cases in $\mathcal{T}_{Bert}/\mathcal{T}_{FIM}$	9	10
Mutation score of the other approach	1/9=11.11%	1/10=10.0%

Table 13: Result of AECS model on output-based mutation testing based on metrics in Section 5.1

	BERTiMuS	FIM
number of compiled mutants	79	84
number of killed mutants	34	27
observation effort	45/79=57.0%	53/94=67.9%
number of test cases in $\mathcal{T}_{Bert}/\mathcal{T}_{FIM}$	1	1
Mutation score of the other approach	1/1=100%	1/1=100%