Object Constraint Language Based Test Case Optimisation

Jin, Kunxiang

Awarding institution:
King's College London

Object Constraint Language Based Test Case Optimisation

by
Kunxiang Jin

Submitted in partial fulfilment of the requirements
for the Degree of Doctor of Philosophy in Computer Science

First Supervisor: Dr. Kevin Lano
Second Supervisor: Dr. Hana Chockler

Department of Informatics
King's College London
London, United Kingdom
March, 2024

Dedicated to

my mother Yongmei Zhao

my father Zhiqiang Jin

and

my fiancee Ning Guo

*who always support me unconditionally
during the whole PhD journey*

Acknowledgements

I would like to express my deepest gratitude to all those who have supported and guided me throughout my journey to completing this PhD thesis. The pursuit of this degree has been a challenging experience, and I am grateful for the opportunities and knowledge it has afforded me.

Foremost, I extend my heartfelt gratitude to my supervisor, Dr. Kevin Lano, for his invaluable guidance, mentorship, and support throughout my PhD journey. His wisdom, enthusiasm, and dedication have been instrumental in shaping both my research and personal growth.

My sincere appreciation goes to the faculty, staff, and colleagues in the Department of Informatics at King's College London. I am grateful for the vibrant and nurturing research environment that has allowed me to grow and learn. Special thanks to my second supervisor, Dr. Hana Chockler, who gave me significant support during my PhD experience.

To all my friends who sent a message of encouragement, shared a meal, celebrated a milestone, or drank a pint. This journey would have been immeasurably harder without you.

Lastly, I wish to acknowledge and honour the countless researchers before me whose work has paved the way for this research. Their contributions to the field have been a constant source of inspiration and guidance.

In conclusion, while this thesis carries my name, it is the culmination of collective efforts, sacrifices, and support from many. I am ever grateful and dedicate this work to all who have played a role in this challenging yet fulfilling journey.

Abstract

Software testing, a pivotal phase in the Software Development Life Cycle (SDLC), ensures the correctness and performance of the corresponding software system. Model-based testing (MBT) is a method to validate whether the software system or specification satisfies the pre-defined requirements through design models. However, with modern systems expanding in complexity, testing has become a labour-intensive and unpredictable process within the SDLC. Therefore, many test case optimisation (TCO) techniques have been proposed to make the testing process more manageable. But these approaches predominantly focus on code-based strategies, leaving systems expressed in Object Constraint Language (OCL) underserved. OCL is a part of the Unified Modeling Language (UML) standard and is a type of declarative language used to describe system specification by pre- and post- conditions. Initially, OCL has been proposed as a constraint language to add more details to the UML model, but alongside the development of OCL itself, there are more and more systems whose specifications are expressed in OCL.

This thesis aims to systematically investigate the feasibility of applying TCO techniques to the OCL-defined systems, with an emphasis on test case prioritisation (TCP) and test case minimisation (TCM) processes. A systematic literature review for the directly related topic, UML-based test case generation, is conducted in this thesis. Also, we adapted a set of test case optimisation algorithms and compared the performance between these algorithms under the context of OCL. Moreover, we modified one metric for the TCP evaluation process, which made the metric more suitable for the MBT and mutation testing environment. Furthermore, we introduce a full set of mutation operators and corresponding classifications to the OCL standard library, offering practical guidance for optimisation processes.

The proposed TCO processes are validated and evaluated through four real-world systems expressed in OCL with different complexities. The experiment results demonstrate that for the TCM process, the size of the minimised test suite is reduced from 33.33% to 81.8% without losing any fault detection ability. For the TCP process, leveraging the modified evaluation metric, the improvements are up to 50%, indicating that the prioritised test suite can detect system defects earlier when compared to the original one. Evaluating based on the considerations of effectiveness, efficiency and stability, we suggest the NSGA-II for the TCM process and the genetic algorithm for the TCP process. When combining TCP and TCM processes, the TCM process consistently increases the efficiency of the TCP process by reducing the search space for the prioritisation process.

Table of Contents

Chapter 1

Introduction

1.1 Overview

Software testing aims to validate the correctness and performance of the software system and determine whether the system specification conforms to the pre-set requirements [1]. As the size and complexity of the system increased, testing became one of the most time-consuming and unpredictable processes within the Software Development Life Cycle (SDLC). During the SDLC, more than 50% of time is typically spent on testing [2], so optimising this process is necessary and meaningful.

There are three most commonly used Test Case Optimisation (TCO) techniques to optimise the testing process, which are Test Case Minimisation (TCM), Test Case Selection (TCS) and Test Case Prioritisation (TCP). TCM seeks to speed up the testing process by removing redundant and unnecessary test cases. TCS categorises test cases into different subsets and decides which subsets are requisite for execution. Moreover, TCP aims to find the defects as early as possible by re-ordering the test cases. The underlying objective of these techniques is to help minimise efforts spent on testing [3].

However, the majority of current research is established for the code-based approaches [4]. We noticed a lack of research applicable to the specification of systems expressed by Object Constraint Language (OCL). OCL is a part of the Unified Modeling Language (UML) standard [5] and is a type of declarative language used to describe system specification by pre- and post- conditions. One of the critical benefits of modelling systems by OCL is the platform-independent character, which lets OCL has the same abstract level as system models. This feature helps us to perform the TCO processes once. Then the results can be applied to all implementations of OCL specifications, despite the differences between programming languages and platforms.

Model-Driven Engineering (MDE) is not a specific new technology but rather a natural aggregation or synthesis of various technologies around modelling and modelling software development [6]. The goal of MDE is not only to bring short-term efficiency to software developers but also to make software products less sensitive to change and increase software longevity, which in turn leads to long-term efficiency gains. Model-Based Testing (MBT) is a part of MDE that uses design models or specifications to perform the testing process. MBT allows the testing process to start before software implementation. We will provide more information related to MDE and MBT in Chapter 2.

Since MBT allows the testing process to start without actual system implementation, we need a technique to provide the necessary information to guide the TCO processes. One possible solution is artificial simulated system defects. Mutation testing is a fault-based technique that has been studied for over 50 years since it can be traced back to 1971 [7]. In mutation testing, from a program or specification p, a set of faulty versions p' called mutants are generated by making, for each p', a single simple change to the original program p.

Mutation operators are the transition rules defining how to perform these changes and derive the mutants. Mutant versions of expressions are syntactically and type-correct, but should have a distinct semantics from their source. The findings presented in [8][9] reveal a statistically significant correlation between detecting mutants and identifying actual faults, underscoring the potential of mutation testing as a robust predictor of real fault detection capabilities. This establishes a foundational basis for leveraging mutation scores to guide the TCO processes. Such an approach augments the effectiveness and efficiency of the testing process without compromising fault detection capabilities [10].

These analyses are applicable to both procedural code and OCL, where mutants simulate common programming errors like operator choice and logic mistakes. Moreover, regarding mutants produced from the procedural activities of UML operations, which are expressed in a procedural OCL extension similar to Pascal statements, mutation operators align with those used in traditional programming, indicating that the anticipated outcomes are consistent with those reported in related research.

Since there are still no systematic proposed mutation operators for OCL specification available, in this work, we propose full set mutation operators, which mainly follow Clause 11 *"OCL Standard Library"* within the OCL 2.4 standard [11]. We also present the classification of these operators based on different facets of OCL standards and expressions.

In this work, we adapted a set of algorithms to perform OCL-Based TCO processes, which mainly focus on TCP and TCM processes. After proposing the optimisation algorithms, we use one small-scale OCL specification as a running example to demonstrate the proposed approach. Then, four real-world OCL specifications are used as case studies within the evaluation process.

We neglect TCS in this work because the selection only retains essential test cases to re-run, and some of the test cases have been removed during the selection process. However, the discarded test case may still be helpful in the future testing period. Although TCM also removes some test cases, different from TCS, the minimisation process only discards redundant test cases [12]. The discarded test cases are unnecessary, which means that the remaining test case has the same testing effectiveness as the original ones. We will discuss more details in Chapter 3.

1.2 Motivations

As aforementioned, testing is one of the most time-consuming and labour-intensive activities within the SDLC. With the limited computing resource, the TCO is essential to save testing effort. When we conducted the systematic literature review on the topic of UML-based test case generation [13], we noticed that various approaches are proposed to constructing test cases. However, only a few of them performed the TCO after test case generation. Moreover, those works that applied the TCO after the generation phase indicate that the optimised test cases will benefit the testing process.

The motivation behind this research work is to explore the possibility of applying TCO processes to the systems whose specifications are expressed in OCL. Due to OCL being on the same abstract level as the system model, it is not dependent on any implementation language. Based on the platform-independent character, OCL is a promising practice for modelling the software system and supporting automatic MDE. It is challenging as there is seldom information to guide the OCL-based TCO. So, we propose a full set of mutation operators for the OCL standard library. Through the artificially simulated faulty specification to gather the corresponding information, which is necessary during the optimisation processes.

More specifically, the motivations of this research project are:

- Although TCO techniques have been widely studied over the last decades, most of these works are based on source code or system level. Compared to the code-based approaches, fewer works are performed under the MBT scenarios, especially for OCL.
- The MBT process has its natural advantage that can be conducted before the actual systems implementation phase. Moreover, expressing system specifications at the model level will be language or platform independent, enabling the optimisation result can be used for all implementations of the corresponding OCL specifications.
- Initially, OCL has been proposed as a constraint language to add more details to the UML model, but alongside the development of OCL itself, there are more and more systems that are expressed by OCL [14]. Although OCL is not as sophisticated as other 3GLs [1], many researchers and practitioners are contributing to the community. Therefore, exploring the different aspects of OCL-based techniques is worthwhile.
- Considerable approaches are proposed for various programming languages, like Java or C, but whether these approaches are feasible under the context of OCL. And how

[1]Third Generation Language

should we adapt these approaches and make them possible to the system specifications expressed in OCL.

• Mutation testing is one of the primary techniques used to guide (or as the objective) TCO processes. However, we noticed a lack of comprehensive mutation operators for OCL expression, and only a few works presented limited operators. There is a need to propose the corresponding mutation operators to the OCL library.

• When the TCO techniques are adapted to the OCL context, we need to be certain that comparing the performance and effectiveness of the optimisation algorithms is meaningful.

• Most MBT approaches only perform empirical studies on small-scale case studies. In our group, we have an archive of systems whose specifications are expressed in OCL, and these specifications have distinct sizes and complexities. These specifications can let us systematically validate the proposed works and verify our ideas that TCO techniques can be utilised for the systems whose specifications are expressed in OCL.

Although further testing automation may reduce the employment opportunities for software developers in testing, we expect that the overall high demand for software would mean that the testing work would be replaced by other development work. Overall, the *Object Constraint Language Based Test Case Optimisation* is a necessary research topic.

1.3 Research Objectives

Currently, most MBT works based on UML or OCL focus on the test case generation process. Still, there is a need to perform more research on the TCO processes, which will further improve the automaticity and save testing effort. This gap is addressed by doing a systematic literature review on the topic of UML-based test case generation. This research does not focus on the test case generation but moves further to TCP and TCM. These optimisation techniques will save numerous testing budgets and efforts in the testing process.

In short, the objectives of this thesis are:

- To explore the current MBT trends by conducting a systematic literature review.
- To propose mutation operators for OCL standard library.
- To classify the proposed mutation operators.
- To apply optimisation algorithms to the TCP process.
- To apply optimisation algorithms to the TCM process.
- To evaluate OCL-based TCO processes via four real-world case studies.

The proposed algorithms use heuristic algorithms to perform TCP and TCM processes, thus saving testing efforts. We presented these algorithms, suitable for the OCL-based approach, that can optimise test cases simultaneously during the system implementation phase to benefit the testing process. When the constructed or existing test cases have been optimised, the testing phase can directly use them without more budget. Until now, not much research has been conducted on OCL-based TCO processes. To the best of our knowledge, we are the first to apply these optimisations to the system specifications expressed in OCL. In our proposed algorithms, the optimisation problem has been suitably adapted to the corresponding heuristic algorithms, and the evaluation process has been performed through real-world OCL specifications.

In order to better evaluate this research work, the following research questions (RQs) are proposed, and we will discuss more details in Chapter 7:

- **RQ 1:** *Effectiveness.* How effective is the TCO process within the scope of OCL specifications?
- **RQ 2:** *Scalability.* To what extent do the proposed algorithms scale to big or real-world OCL specifications?
- **RQ 3:** *Comparison.* Which TCO algorithm can give the best performance?
- **RQ 4:** *Metric.* Since we modified one TCP evaluation metric, what are the differences from the original one? And what is the performance of the TCP process under this modified metric?
- **RQ 5:** *Efficiency.* What is the overhead (time consumption) when applying the TCO process to OCL specifications?

1.4 Aims and Contributions

This research aims to perform TCO processes on the systems whose specifications are expressed in OCL, then through the optimised test cases to benefit the software testing process. In this work, we apply different heuristic algorithms to optimisation problems and systematically evaluate the proposed algorithms through four real-world OCL specifications. We can conclude that performing TCO processes to systems expressed in OCL will benefit the testing process and save testing efforts.

In detail, the main contributions of this research work are as follows:

- Conducting a systematic literature review on a directly related topic, UML-based test case generation, to this research work. This review demonstrated the generic process of different approaches to the test case generation process.

(Chapter 3)

- Proposing the full set of mutation operators to the OCL standard library, which includes the operators to primitive types, collection-related types, pre-defined iterator expressions and structural operators. Moreover, the potential classification of these operators has been proposed. (Chapter 4)
- Classifying the proposed mutation operators into different groups according to the logic in common. (Chapter 4)
- Proposing and applying five optimisation algorithms for the TCP process and discussing the corresponding evaluation metrics. (Chapter 5)
- We analysed the limitations of the most commonly used metric, the Average Percentage of Fault Detection (APFD), for the TCP process. These weaknesses led to the APFD metric being unsuitable under the MBT context, and then we proposed a modified version of the APFD metric to overcome these defects. (Chapter 5)
- Proposing and applying five optimisation algorithms for the TCM process and discussing the corresponding evaluation metrics. Instead of the single-objective optimisation used in the TCP process, the TCM process employs a multi-objective optimisation process. (Chapter 6)
- Systematically evaluating the effectiveness and efficiency of OCL-based TCO processes on four case studies with different complexities. The performances of different optimisation algorithms for the TCP and TCM processes are compared and investigated. (Chapter 7)

After reviewing the UML-based test case generation, we noticed there is a need to conduct this research work. Defining the full set of mutation operators to OCL standard library allows us to simulate the system defects by modified

OCL specifications. Then, to guide the TCO processes by using the corresponding information. To better organise the proposed mutation operators, we also present the classification of these operators.

After the mutation operators to OCL specification have been proposed, we applied the heuristic algorithms to satisfy the OCL-based TCO problems. In the evaluation phase, we first use a small-scale specification to demonstrate the proposed approach, then validate the effectiveness and efficiency via four different scales of OCL specifications, and the comparisons between these algorithms are also conducted.

The following are the research stages involved in this thesis:

- Introduction and fundamental background.
- The systematic literature review.
- Design and classify mutation operators.
- Propose the TCO algorithms.
- Case studies for evaluation purposes.

1.5 Overall Thesis Structure

In order to achieve our aims and better organise the overall structure of this thesis, eight chapters have been categorised. After Chapter 1 introduces this research, Chapter 2 provides the essential background to the software development process, model-driven engineering, model-based testing and object constraint language. In Chapter 3, we report a systematic literature review on the directly related topic, UML-based test case generation. Also, some related works to TCO and mutation testing are discussed.

Chapter 4 proposes the mutation operators to OCL standard library and the corresponding classifications for these operators in detail. Chapter 5 and Chapter 6 demonstrate the proposed algorithms and evaluation metrics for the TCP and TCM processes, respectively. In Chapter 5, we also modify the APFD metric for the TCP evaluation process.

Chapter 7 provides the evaluation process for this research. One small OCL specification has been used as a running example. Then, four case studies are proposed to evaluate the OCL-based TCO processes. Finally, Chapter 8 summarises the outcomes of this research work, and some possible future works are also presented.

1.6 List of Publications

During the PhD journey, 6 publications have been published.

- **Jin, Kunxiang**, and Kevin Lano. "Generation of test cases from UML diagrams-a systematic literature review." *14th Innovations in Software Engineering Conference (formerly known as India Software Engineering Conference).* 2021.
- **Jin, Kunxiang**, and Kevin Lano. "Mutation Operators for Object Constraint Language Specification." *STAF Workshops.* 2021.
- Lano, Kevin, **Kunxiang Jin**, and Shefali Tyagi. "Model-based testing and monitoring using agileuml." *Procedia Computer Science* 184 (2021): 773-778.
- **Jin, Kunxiang**, and Kevin Lano. "OCL-based test case prioritisation using AgileUML." *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings.* 2022.
- **Jin, Kunxiang**, and Kevin Lano. "Design and classification of mutation operators for OCL specification." *Proceedings of the 25th International Conference on Model Driven*

Engineering Languages and Systems: Companion Proceedings. 2022.

- Lano, Kevin, S. Kolahdouz-Rahimi, and **Kunxiang Jin**. "OCL libraries for software specification and representation." *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. 2022.

Chapter 2

Background

In this chapter, the essential backgrounds to the software development process, model-driven engineering (MDE), model-based testing (MBT) and object constraint language (OCL) are demonstrated. Regarding the TCO problems and mutation testing, the detailed definition and the corresponding related works will be discussed in the next chapter.

2.1 Software Development Process

The software development process contains the necessary activities to develop and maintain software systems. Typically, during SDLC, various phases are included to ensure high-quality systems. A typical SDLC, like *Figure 2.1*, consists of the following stages: planning, requirement analysis, design, implementation, testing, deployment and maintenance.

In planning, the leaders of the project usually evaluate the terms of the project, such as labour cost, to create the timetable. Planning must include feedback and comments from the stakeholders or potential ones who will benefit from the project. Requirement analysis identifies the feasibility of the project, defines what the system is meant to do and which features should be included [15].

SDLC requires a design step to model the architecture of

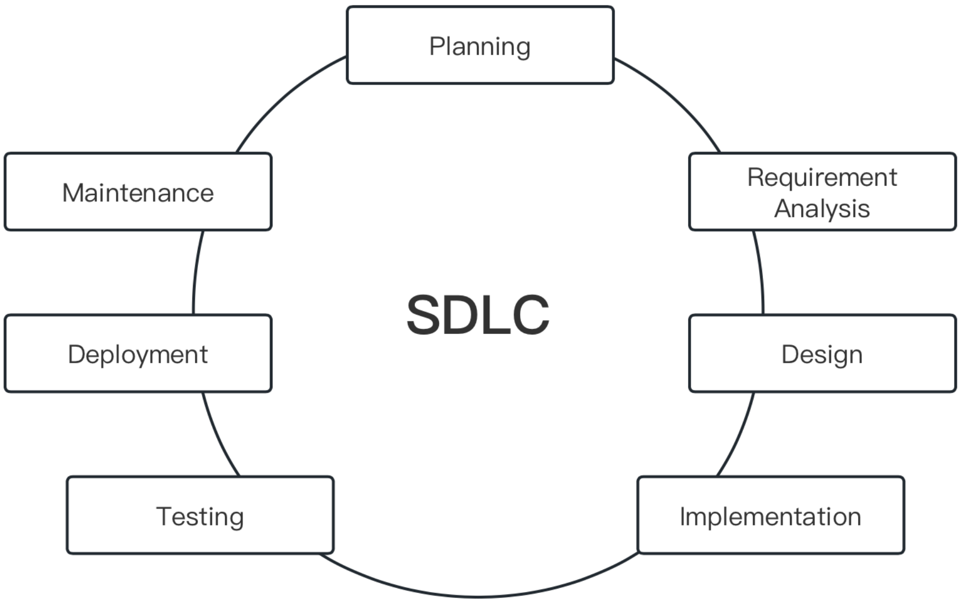


Figure 2.1: Software Development Life Cycle

the software system and how the system will work. Particularly, this step identifies the functionalities of the system, data representation, and internal or external communications. Since developers must follow the guidelines and design documentation of the project, the phases mentioned earlier need to be treated carefully. After the design stage, the actual system implementation will be started.

Testing activity is essential to ensure the system implementation conforms to the design and the functionalities run as expected. As the size and complexity of the system increased rapidly, testing became the most time-consuming and unpredictable activity. This research work mainly focuses on this phase and aims to optimise this process.

When the testing process is completed, the system should be deployed to the appropriate market. The system may first release limited segments to test in a real operating environment, then deploy the entire system. System maintenance is an ongoing activity which includes error fixing, functionality enhancements, keeping the system running correctly, etc.

Until now, several SDLC models have been introduced that can be applied to the software development process. In this chapter, we will mention two development methodologies, one traditional model and one agile model. The traditional one is called the Waterfall model, like *Figure 2.2*, which was introduced around 35 years ago [16].

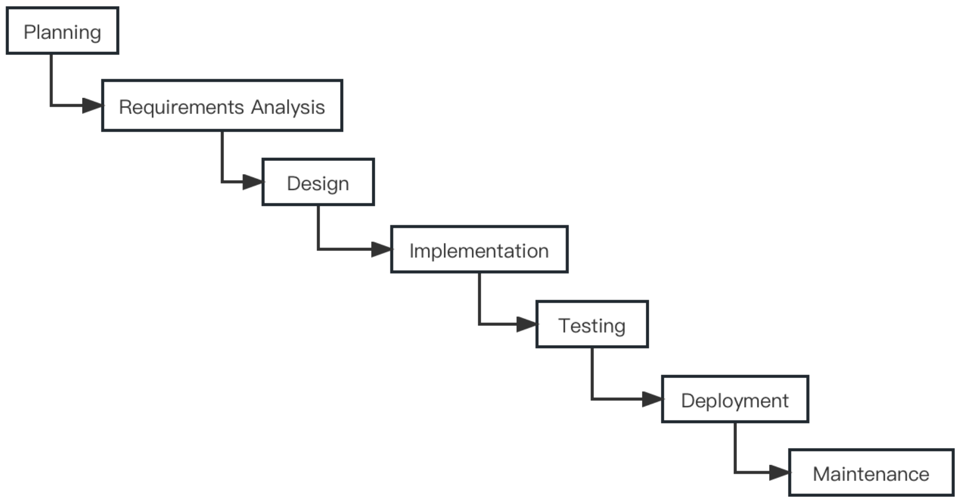


Figure 2.2: Waterfall Model

In the waterfall model, through all development stages, each process moves in a cascade mode [17]. In this model, the next step can only start after the previous one is completely finished. And the movements between stages are strictly documented and cannot be re-evaluated. For example, the design specifications cannot be revised when the implementation phase is started. The benefits of applying this model are that it is easy to use, understand and manage. Also, each step is strictly documented. However, the main disadvantage of this model is that the team can only operate the system once the last step is finished, which leads to high-risk and unpredictable project results.

With the changing requirements and user needs, now 70%

of organisations employ the agile approach in their businesses [18]. There has yet to be a consensus on what actual "agile" means [19]. But in general, the core features of agile are iterative development, frequent communications, and early feedback [20]. One of the most popular agile models, Scrum, shown in *Figure 2.3*.

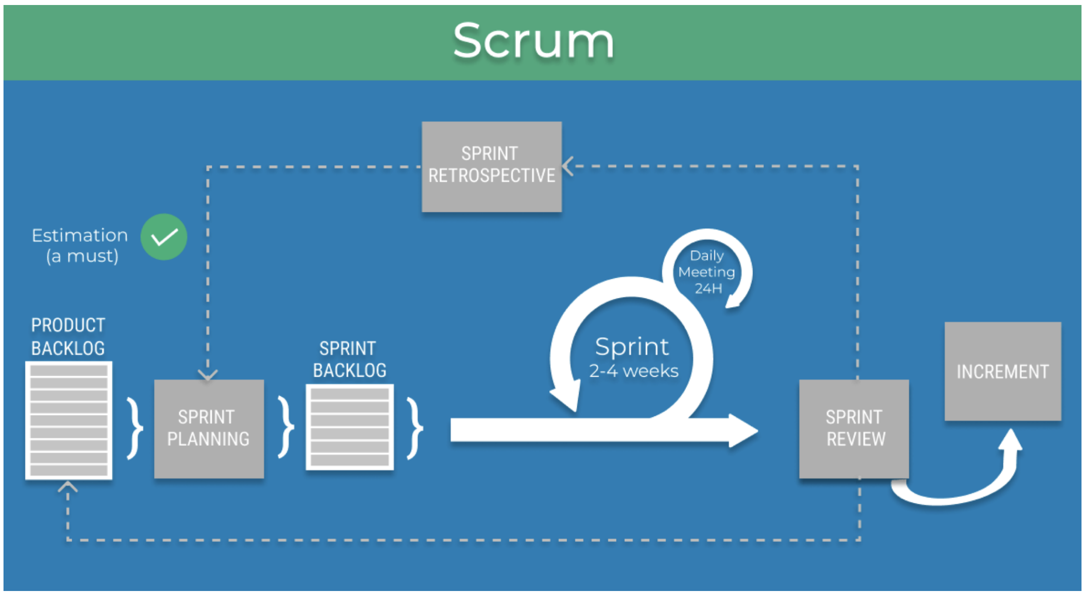


Figure 2.3: Scrum Model [21]

Scrum aims to divide the entire development process into multiple segments and repeat a cycle of planning, design, implementation, testing, etc. Each cycle is called a Sprint in this model. Scrum is getting more attractive as this model focuses on smaller goals and iteratively integrates these goals into the main objectives of the project. The primary advantage of applying this agile model is flexibility. Scrum is adaptable to the environments and requirements, especially when the requirements are not clearly identifiable initially. This model is suitable for project requirements that need frequent adjustments and may not be ideal for projects that require a well-defined plan. Moreover, Scrum needs experienced team members to participate in the development process because everyone involved needs to perform their duties successfully and quickly.

Due to this research not focusing on the SDLC models, the detailed review and analysis of the traditional and agile models can refer to [22, 23, 24, 25, 19].

Regardless of traditional or agile development models, testing is always one of the most critical phases within the SDLC. Software testing determines whether the implemented system meets the expected requirements and ensures the system does not contain apparent defects. Testing is not a "silver bullet" to promise the system is bug-free, but trying to find as many bugs as possible [26]. Why is testing essential, and many efforts spent on this activity? Testing is necessary because if there are any software system defects, testing helps identify and fix them earlier before the system is delivered. And during the development process, the cost of fixing a system defect increases exponentially as the development process moves forward in SDLC [27].

Test case and test suite are two essential components of the testing activity. In this thesis, we informally define the test case as the combination of a set of inputs to the software system and the corresponding expected output. We consider the test case *passed* during the testing phase if the actual output equals the expected one. Otherwise, the test case *failed*. Also, for the test suite, we define this concept as a set or collection of test cases.

2.2 Model-Driven Engineering

In the field of computer science, the concept of MDE has attracted broad attention worldwide, mainly after the concept of Model-Driven Architecture (MDA) put forward by the Object Management Group (OMG) in 2000 [28]. The broader context dates can be traced back to the 1980s and 1990s when various modelling techniques flourished.

MDE is not a specific new technology but rather a natural aggregation or synthesis of various technologies around modelling and modelling software development. The goal of MDE is not only to bring short-term efficiency to software developers but also to make software products less sensitive to change and increase software longevity, which in turn leads to long-term efficiency gains. MDE offers a promising approach to solving the problem that third-generation languages are unable to alleviate the complexity of platforms and effectively express domain concepts [29]. In contrast to the object-oriented world, MDE researchers have developed the basic principle of *everything is a model.*

Compared with other software development methods, MDE pays more attention to the abstract description for constructing different domain knowledge based on the idea of depicting the software system through the models. In MDE, the developers use models and layers of automatic or semi-automatic transformation to perform SDLC activities partially or entirely.

MDE is based on high-level system models, using highly abstract domain models as components and various model-driven transformations to complete the system development process. Like *Figure 2.4*, the system is the implementation of the domain models, while the domain knowledge is presented by the corresponding models. By using these models to reduce development costs and respond to complex requirements changes. The basic idea of MDE is to move the development centre from programming to higher-level abstraction. The developers utilise models to derive actual implementation code or other development artefacts, then let some or all SDLC activities can be performed automatically. This automaticity is used to address two fundamental crises in software, which are complexity and the flexibility to change.

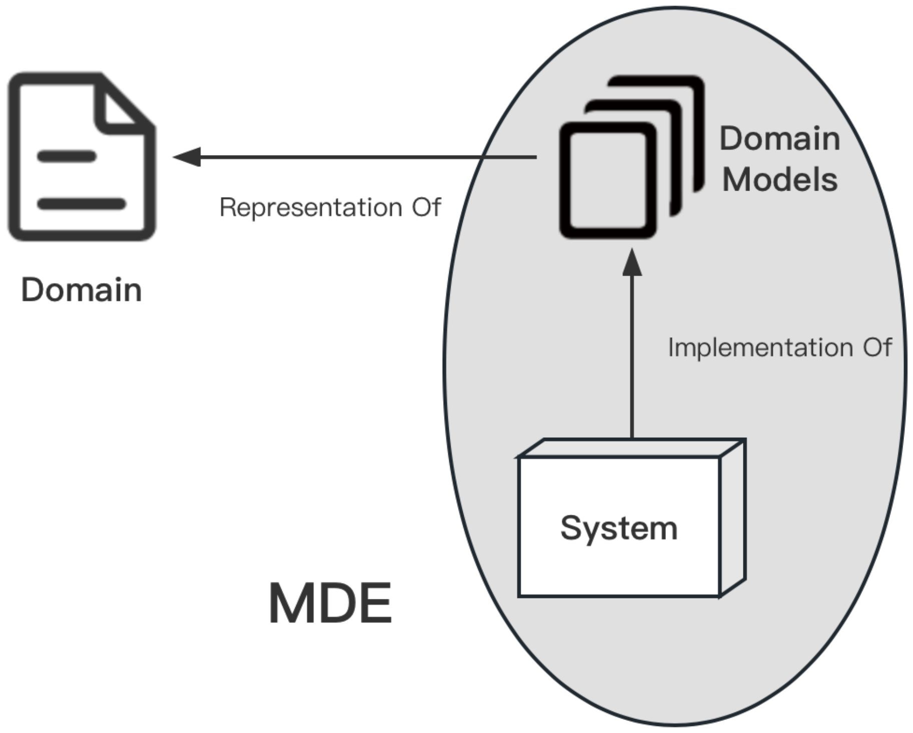


Figure 2.4: Model-Driven Engineering

The advantage of MDE is using models which are closer to human understanding and knowledge, especially visualised models. This benefit allows designers to focus on information relevant to the business logic rather than prematurely thinking about platform-specific implementation details. Especially in the face of different application domains, the model-driven approach emphasises using convenient and flexible models to describe the system. Based on domain knowledge, achieve good communication among domain experts, designers, system developers and others.

In general, MDE is a software development method which takes models as the centre. Moreover, MDE has the following benefits [30]:

- *Portability:* Increasing system reusability and decreasing the complexity of system development and management from the present into the future.
- *Platform Interoperability:* Ensuring systems all achieve the same business functions through rigorous approaches, regardless of the detailed implementation methodologies.
- *Platform Independence:* The models can be used across platforms, which significantly alleviates the time, cost, and complexity of repositioning systems for different platforms, including those platforms that may not yet be introduced.
- *Domain Specificity:* Through domain-specific models to enable rapid implementation of systems over various platforms.
- *Productivity:* Allowing developers, designers and system administrators to use their own familiar language or business concepts during the development process, which also allows seamless communication and integration between different teams.

2.3 Model-Based Testing

Given the importance of testing activity, MBT is one of the most significant concepts within the scope of MDE. MBT is a testing method in the field of software testing to validate whether the system or specification under test (SUT) [1] satisfies the pre-defined requirements by using models. One key observation is that even well-constructed manual testing activity still consumes numerous efforts and resources [31]. MBT alleviate this deficiency by improving the automatic level of the testing process.

[1]In this thesis, SUT stands for the system or specification under test. We do not clearly distinguish them here because the system is expressed using OCL specifications in this work.

Modern software engineering emphasises incremental and iterative development processes, adopts object-oriented development technology, improves software development quality and accelerates software development speed. Most design specifications of object-oriented software systems are expressed by models, which contain a large amount of information that can be used for software testing. Since MBT does not require actual implementation details, the testing activity can be advanced to an earlier stage within SDLC. In addition to the early start feature, the MBT also have a high level of automation, which allows the testing efforts and resources to be saved substantially.

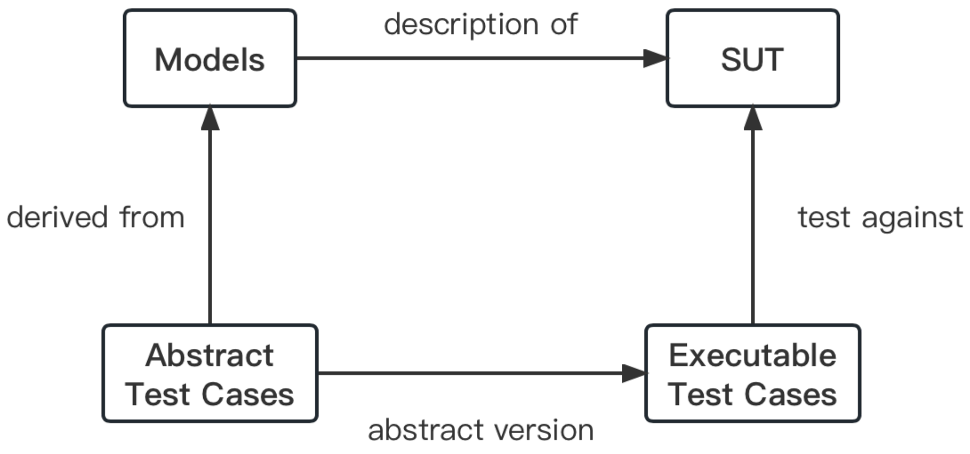


Figure 2.5: Model-Based Testing

Figure 2.5 demonstrates the relationships between models, SUT, abstract and executable test cases. In MDE, models are used to represent the software system, and executable test cases are used to test against the corresponding SUT. The abstract version of executable test cases is called abstract test cases, which are derived from models directly.

Usually, the test cases derived from the model have the same abstraction level as the model, and these test cases are so-called abstract test cases. Abstract test cases typically cannot validate the SUT directly because those test cases are on a higher abstraction level than the SUT [32] [33]. The executable test cases can be derived from the corresponding abstract test cases, and these test cases are used to validate the SUT directly. In most scenarios, the transformation between abstract and executable test cases can be performed automatically. However, in some circumstances, the executable test cases can be created directly if the model contains explicit information.

Compared to source code-based testing, one advantage for MBT is that even if source code-based testing passed all test cases and satisfied all testing criteria, it is still hard to say that the SUT meets the design requirements. However, MBT solves this problem by directly generating test cases from the design specifications. MBT is a black-box testing technique because this process does not require any source code.

There are various MDE tools that support the MBT process, and three tools are briefly described as follows:

• *Papyrus* [34]. This tool is a graphical editing application for UML 2 standard and has been provided as an Eclipse plugin. Papyrus supports code generation, such as Java, C, and C++, etc., and facilitates the links between external tools, which allows the model as a driving artefact for the development process. The generated code can help the software system implementation and create the expected test output from the corresponding test data.

• *Umbrello* [35]. Umbrello is an open-source UML diagram modelling tool developed by the Umbrello Team. The tool supports the model-to-text transformation, which can convert UML diagrams to Java, C++, C#, PHP, Python, and SQL specifications and supports reverse engineering. Umbrello supports XMI [2] 1.2 and 2.0 as import and export files, and this tool also supports third-party file import formats. The XMI format specification for the UML diagram makes the analysis and manipulation of UML models more convenient and manageable. However, Umbrello does not supports test case generation.

- *AgileUML* [36]. Unlike Umbrello and Papyrus, the in-house MDE tool AgileUML supports the features like modelling systems by UML or OCL, generating test cases and constructing mutants from OCL specifications [3]. AgileUML provides the code generator for the target languages, including Java, Swift, C++, etc. For the test case generation feature, this tool can generate abstract test cases for all supported target languages. But for JAVA, both abstract and executable test cases can be constructed.

2.4 Object Constraint Language

When introducing OCL, UML cannot be bypassed. First, the fundamental knowledge of UML will be discussed.

UML is a general-purpose modelling language first adopted as a standard by OMG in 1997. It is already a de-facto standard both in industry and academia. UML has well-defined semantics in order to avoid ambiguities. The visualisation ability of UML models makes the system structure more intuitive and easy to understand.

Using UML to demonstrate a software system will benefit the development process by promoting communication between developers and simplifying the remaining development activities. A model is an abstract level description of the

[2]XMI - XML Metadata Interchange; XML - Extensible Markup Language
[3]Umbrello and Papyrus do not support mutation testing.

system, which contains a set of views used to represent the design specifications of the corresponding system, from functional to non-functional aspects. The information contained within the models will help developers better plan the development process and guide other SDLC activities. With suitable models, the system design can be appropriately realised to ensure the requirements are met. Moreover, these models can enhance flexibility when facing requirement changes.

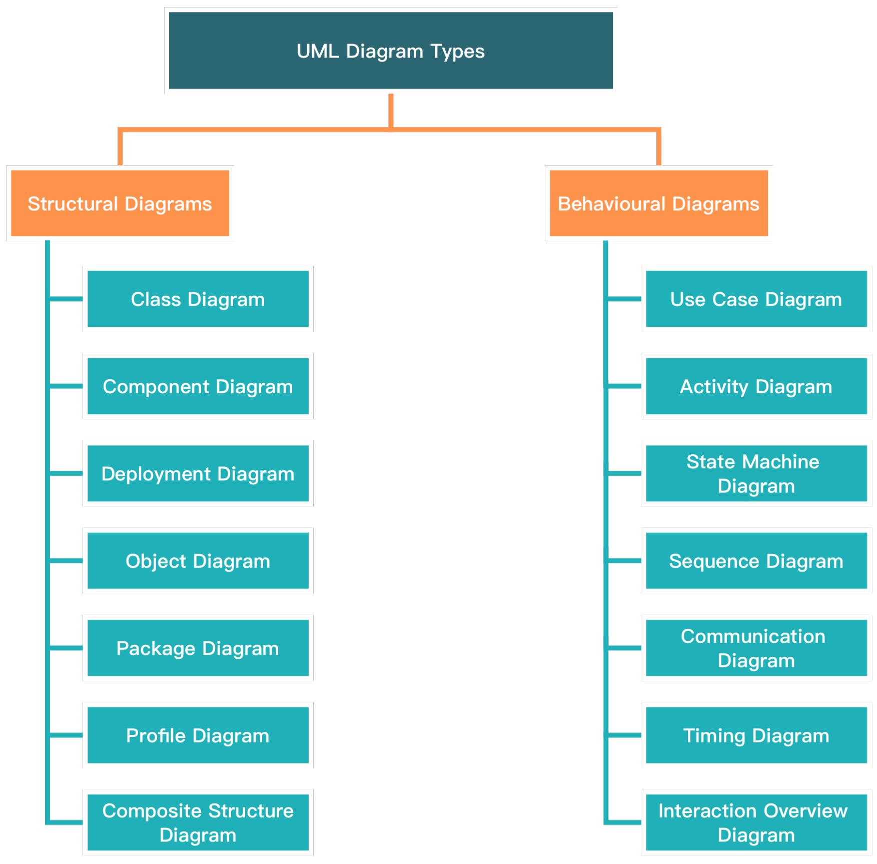


Figure 2.6: UML Diagram Types

With the progression of the UML standard, the current version of the standard is UML 2.5. There are 14 different types of models used to describe the system. These models can represent different views of the corresponding system, from structural to behavioural aspects [37]. As shown in *Figure 2.6*, the structural models contain the *class dia-*

gram, component diagram, deployment diagram, object diagram, package diagram, profile diagram and *composite structure diagram.* At the same time, the behavioural models include the *use case diagram, activity diagram, state machine diagram, sequence diagram, communication, timing diagram* and *interaction overview diagram.*

In MBT, the types of UML models used in most approaches, especially in test case generation, concentrate on the class diagram, state machine diagram, sequence diagram and activity diagram. In these four types of diagrams, only the class diagram is the structural diagram, which describes the system from a high-level perspective. The remaining three types are all behavioural diagrams. Due to this research mainly focusing on OCL rather than UML, we will briefly introduce these four types of UML models here instead of comprehensively demonstrating all UML diagrams.

Class diagram is a structural diagram that defines the entities of the system with their internal data and interrelationships.

State machine diagram is a state machine composed of states, transitions, events and activities. A state machine diagram is used to describe all possible states within the system, and the transitions between these states, which are caused by events [38].

Sequence diagram describes the order of messages sent between objects, emphasising the chronological order. The primary purpose of sequence diagrams is to translate requirements into further, more formal refinement levels. System requirements can be refined into one or more sequence diagrams commonly. At the same time, sequence diagrams more effectively describe how to assign responsibilities to each class and why each class has corresponding responsibilities.

Activity diagram is a flowchart that describes the flow of control from activity to activity. The primary purpose of activity diagrams is to capture the dynamic behaviours of the system, and each activity is a particular operation within the system. The Activity diagram also can be used to construct the executable system via forward and reverse engineering. Different to the sequence diagram which emphasises the flow of control from object to object, the activity diagram emphasises the flow of control from activity to activity [39].

Graphical modelling languages, like UML, are preferred by developers since these languages are convenient for defining the structural and behavioural aspects of the software system [40]. Nevertheless, the advantages always come with disadvantages. To better keep the notational elements more manageable, these languages must limit the expressiveness. This leads to only a limited subset of all domain information can be expressed [41]. OCL is designed as a complement of the UML in order to be able to identify all domain details precisely and is already a part of UML standard [5].

OCL was only designed as a constraint language for UML originally but rapidly expanded the application scope. Since OCL can be applied to many MDE activities, such as model transformation and specification requirements, OCL already became a key component of MDE. Several versions of the OCL standard have been released to adopt this language into various MDE application domains, and the current version of OCL is 2.4 [11].

OCL is a declarative language without any side effects, which means OCL expressions can query or constrain the corresponding systems but not modify them. OCL has many benefits, and the most critical point is that OCL can add the pre- and post- conditions to methods, operations, and models, which can be used to express system specifications [42]. Pre-conditions define the necessary conditions that must be satisfied before executing a specific operation. Conversely, post-conditions outline the expected outcomes or behaviours resulting from the execution of that operation. Detailed insights into using OCL specifications within the context of TCO processes, including illustrative examples, will be discussed in the running example section within Chapter 7.

However, OCL is not a programming language and does not provide direct execution [4]. Each OCL expression can indicate a value or object in the system. Due to OCL expressions can evaluate any value or set of values in a system, the OCL expressions have at least the same power as SQL.

OCL is based on the set theory and predicate logic and has formal mathematical semantics [43]. However, OCL does not use any mathematical notation because although mathematical symbols can express things clearly and unambiguously, only a few experts can understand them. So mathematical symbols are not appropriate for a widely used standard language. Natural language is the most understandable, but it is ambiguous. OCL takes a compromise between natural language and mathematical notation.

In this research, we perform TCO processes to the systems whose specifications are expressed in OCL. We fully supported OCL standard library within our in-house AgileUML tool. The standard library is Clause 11 *"OCL Standard Library"* within the OCL 2.4 standard.

As shown in *Figure 2.7*, this clause describes the predefined types, their operations, and pre-defined expression templates in the OCL. Within the OCL standard Library, the special types, the primitive types, the collection-related

[4]But some research attempted to make OCL executable.

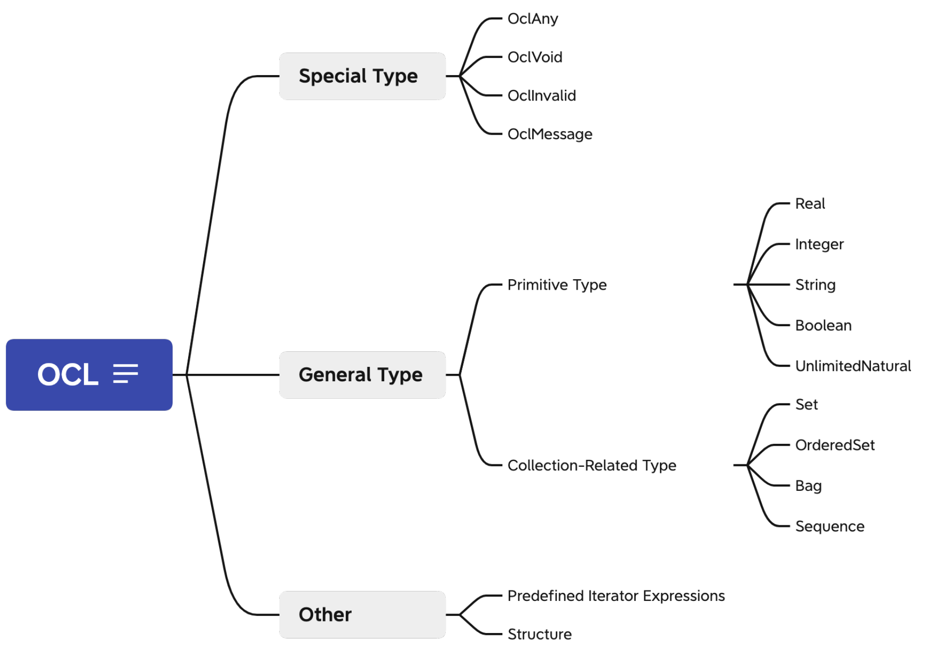


Figure 2.7: OCL Standard Library

types and pre-defined iterator expressions are demonstrated.

Based on the standard library of OCL standard version 2.4, there are two main groups of types, primitive types and collection-related types and pre-defined iterator expressions. The primitive types contain five individual types, *Real*, *Integer*, *String*, *Boolean* and *UnlimitedNatural*. Meanwhile, the collection-related types include one supertype *Collection*, and four sub-types, *Set*, *OrderedSet*, *Bag* and *Sequence*. The four sub-types depend on whether duplicated elements are allowed and whether the elements are ordered. Also, some pre-defined iterator operations are included in the standard library. This clause contains almost all necessary operators and expressions to describe the system specifications.

We fully support the OCL standard library in AgileUML, which lets most of the existing OCL specifications can be benefited from our work. But the standard library has no facilities for some common software aspects, such as files and processes. Various research has been proposed to make OCL more comprehensive like [44] and [45]. We cannot guarantee all the new proposed works can be supported by AgileUML. But in addition to the standard library, AgileUML supports some additional operators, making OCL more expressive. The extra operators are described in [46] and [14].

To better understand UML and OCL, the architecture that OMG chose for its standards is necessary to be mentioned. The four-layered architecture shows like *Figure 2.8*.

Layer M0 is the actual running system, and M1 is a system model which defines the types of entities and relationships that make up a system. M2 is the level where the meta-model is defined. Model elements on M1 instantiate meta-classes on M2. M3 is the most top-level to define meta-model, and the language used is MOF [5]. The element on a certain layer is always an instance of an element from one level higher [6] [47].

Compared to most of the TCO research work on the M0 layer, this research increases the abstract level to M1. With the description of the system by OCL instead of the actual system, we can ignore the differences across different programming languages and platforms. It enables the TCP process to be performed once, and the results can be used for all implementations of specifications in various programming languages or platforms.

Since OCL can easily express system behaviours, an increasing number of systems use OCL to describe system specifications [48][49]. OCL specifications are used to represent the expected behaviours of these systems through pre- and post- conditions.

Figure 2.9 is the running example that will be used in

[5]Meta Object Facility
[6]except M3

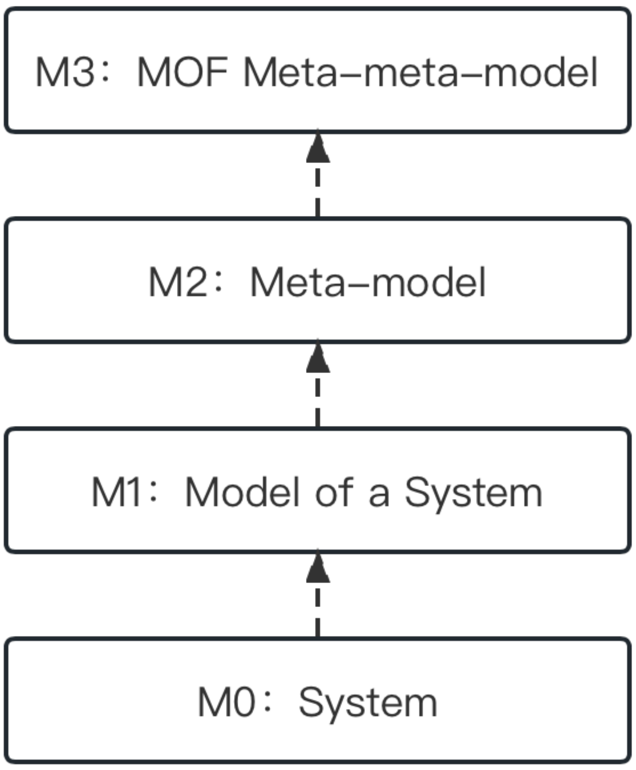
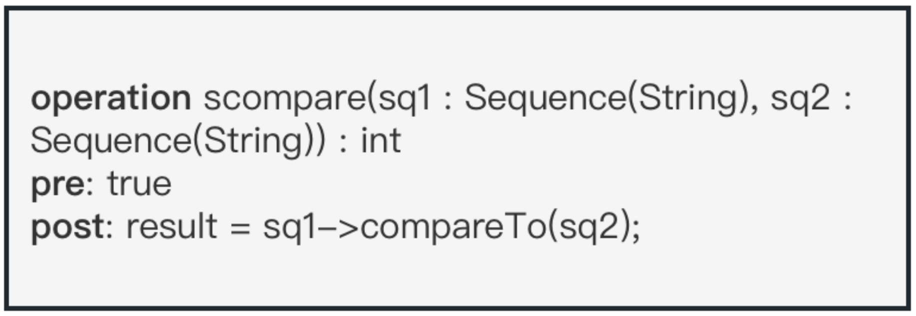


Figure 2.8: 4 Layered Architecture



```
operation scompare(sq1 : Sequence(String), sq2 :
Sequence(String)) : int
pre: true
post: result = sq1->compareTo(sq2);
```

Figure 2.9: OCL Example

Chapter 7, and we will recall this example during the evaluation process. This example models the function that compares whether two sequences of strings are equal. The desired system behaviour is described through pre- and post- conditions. Through this example, we can notice OCL is a very expressive language to express system specifications, and one function only requires three lines to describe.

Chapter 3

Related Works

In this chapter, the related works for this research are discussed. First, a systematic literature review of the directly related topic, UML-based test case generation, is demonstrated.

One of the key findings of this review is that most existing approaches neglect the optimisation of test cases. There is a need to conduct more research on TCO processes, which will further benefit the testing process. Also, with the flourishing of OCL in recent years, more and more system specifications are expressed in OCL. These findings inspired us to perform this research work, *OCL-based test case optimisation.*

After the systematic literature review, the related works to the three main TCO processes (TCP, TCM and TCS) are presented. Finally, we will also discuss mutation testing.

3.1 Systematic Literature Review to Model-Based Testing

This section presented a systematic literature review on the topic of UML-based test case generation. Sixty-two primary studies, which range from 1999 to 2019, are chosen from 443 papers according to the selection criteria. This review demonstrated the generic process of different approaches to the test case generation process. The comparison standards mainly focus on the model type, intermediate format and coverage criteria. The research trends, deficiencies and future works are proposed based on the analysis results of these primary studies.

3.1.1 Introduction

The research topic of test case generation from UML diagrams has existed for decades, and there are numerous publications in this area. Test case generation is mainly from the source code specification or design documentation [50]. Testing based on source code specification always seems unattainable in the software development process because the source code is not always fully accessible to the developer in most projects [51]. And more important, testing on the source code level needs to postpone the testing phase behind the implementation process, which is time-consuming. The source code level testing is cumbersome and hard to automate. On the other hand, the design level testing can start the testing phase before the system implementation [52].

The software testing process should run through the whole SDLC [53]. To better benefit the development process, the test cases are reasonable to be generated before the implementation. The generated test case can be executed immediately when the system is implemented. In MBT, design-level documentation is used for test case generation, which leading the testing phase can begin earlier in the SDLC. MBT is a technique that performs testing activities based on the models, which allows test cases to be generated from one or more system models. A model is an abstract-level description of the SUT, and each model represents a different aspect of the corresponding software system.

There are numerous kinds of models, for example, UML, Labelled Transition System, Petri-Net and Markov Chains. MBT is still a progressing field, especially for complex software systems. One of the common approaches in the MBT community is to perform the test case generation process automatically by using UML [54]. Therefore, using UML-based test case generation becomes a practical and promising topic [55].

In this systematic literature review, there are 443 publications selected through the search strings, and 62 are identified as primary studies. The main comparison points focus on the model type, intermediate format, number of case studies, coverage criteria, test case execution manually or automatically, and whether the test cases are optimised. Through the systematic literature review, the common practice, strengths and weaknesses of generating test cases from UML models are demonstrated, and future works are also proposed for the MBT community.

3.1.2 Research Method

This part reviews the topic through a systematic literature review methodology. And this part is divided into the research questions, search string, exclusion criteria, quality assessment, and data extraction.

A. Research Questions

The systematic literature review protocol is shown as *Figure 3.1.* The review protocol contains the necessary phases of the review and guides the whole review process [33]. The review protocol has three phases, plan review, conduct review and document review. The plan review phase includes defining research questions, developing and validating the corresponding protocol. The conduct review phase contains primary studies selection, quality assessment, data extraction and synthesis. The Document review phase consists of writing and validating the review report.

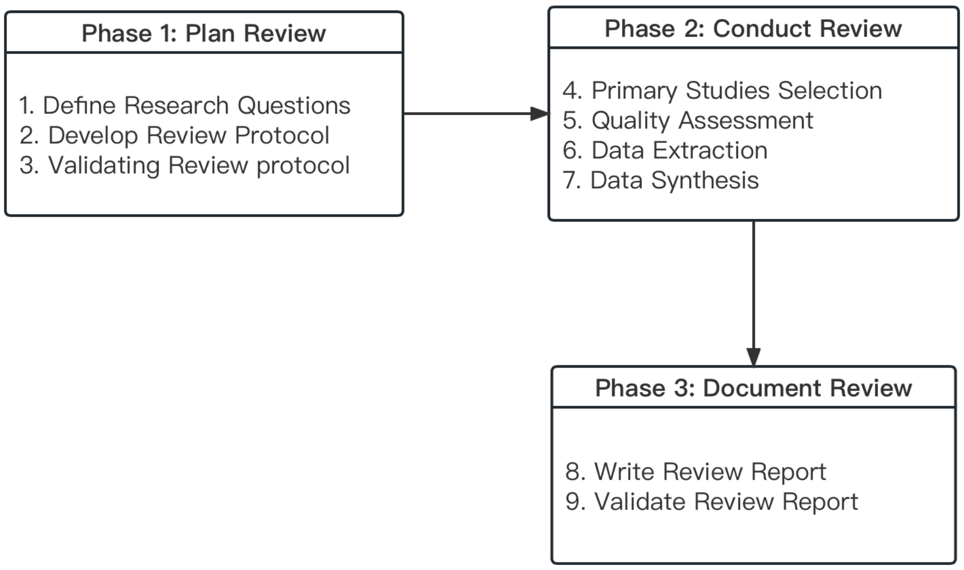


Figure 3.1: Review Protocol

The fundamental step of a systematic literature review is to identify the research questions. The research questions are the discipline and guidance of the review. The research question determines how to select the primary studies, extract the data, and synthesise the data.

Based on the objectives of the review, the research defined the following:

Question 1: Which UML model(s) is used by the primary study?

Question 2: Whether the primary study required the intermediate format?

Question 3: Which coverage criteria does the primary study intend to satisfy?

Question 4: Whether the generated test cases can be executed automatically?

Question 5: Whether the generated test cases have been optimised?

B. Search String

The primary studies selection process starts with defining a selection strategy. Through the selection strategy, the primary studies should be selected. The primary studies need to cover as many approaches as possible based on the review area without any redundant and irrelevant studies. The selection strategy contains two steps. The first step is to choose the databases to select as many relevant works as possible, and the second is to construct the exclusion criteria to eliminate unrelated and repeat works [56].

The related works selected from six primary databases range from 1999 to 2019. The six central databases are *ACM, IEEE, ISI Web of Knowledge, Science Direct, Springer* and *Wiley Interscience.* The end year is 2019 because we conducted this review that year, and the range of studies covers two decades.

The search strings are created manually to perform the selection process. Based on the different databases, the corresponding search strings are constructed. The overall strategy for creating strings is: • The Title should include *model based testing* OR *model based testing and test case generation* OR *model based testing and UML* OR *UML and test case generation.* • The Abstract should contain *UML* OR *UML and model based testing.* These two strategies are used conjunctively during the selection process.

After the search strings have been constructed, the relevant works are extracted from the database through the corresponding search string. There are 443 papers selected by search strings, the details shown as *Table 3.1.*

Table 3.1: Search Result

Database	Number
ACM	2
IEEE	80
ISI Web of Knowledge	41
Science Direct	71
Springer	249
Wiley Interscience	0
Total	**443**

C. Exclusion Criteria

After the first step of the selection strategy, we designed the search strings to minimise the risk of overlooking relevant studies. Then the irrelevant and redundant works should be eliminated by using exclusion criteria. Applying exclusion criteria is the second step of the search strategy. The systematic literature review used the following rules to choose works as primary studies.

Rule 1: The full text of the paper is not available.

Rule 2: The content is not written in English.

Rule 3: Papers selected by search strings are similar or duplicated

Rule 4: Papers are irrelevant to UML-based test case generation.

Rule 5: Papers are survey paper.

Rule 6: Papers only focus on a specific type of case study.

The results paper selection process is shown in *Table 3.1*. Then after adopting the defined exclusion criteria, 62 papers are identified as primary studies. The details of each rule and the number of chosen works are shown as *Table 3.2*.

Table 3.2: Exclusion Result

	ALL	R1	R2	R3	R4	R5	R6	**Valid**
ACM	2	0	0	0	0	1	0	**1**
IEEE	80	0	0	3	44	1	7	**25**
ISI	41	5	0	14	9	2	1	**10**
Science Direct	71	0	0	1	47	2	10	**11**
Springer	249	1	0	2	210	13	8	**15**
Wiley	0	0	0	0	0	0	0	**0**
Total	443	6	0	20	310	19	26	**62**

D. Quality Assessment

When the primary studies are selected, quality assessment is a necessary process. This process ensures the quality of the systematic literature review by applying assessment questions to evaluate the primary studies [57].

Each question is answered when every primary study is reviewed, and the corresponding result is recorded. Each question has three answers with different marks. The answers have Fully Satisfied (1 mark), Partially Satisfied (0.5 marks), Not Satisfied or Not Mentioned (0 mark). The proposed assessment questions are as follows:

Question 1: Is the aim of the study explained clearly?

Question 2: Are all research questions answered?

Question 3: Is the case study described explicitly?

Question 4: Is there any deficiency or future work presented?

Question 5: Is the conclusion consistent with the goal?

E. Data Extraction

The data extraction process aims to gain all relevant information from the primary studies by reading all these 62 studies to extract data and answer the research questions. The obtained data includes authors, publish year, publication platform, model type, intermediate format, coverage criteria, number of case studies, execution type and test cases optimisation option.

Data synthesis is also an essential process in the systematic literature review, through the extracted data, categorise the data and answer the research questions. Using the quantitative method to read and organise the primary studies and using related information to perform data synthesis.

3.1.3 Review Results

The review results for the primary studies will be presented in this part. The primary studies are selected from six leading publication platforms, *ACM, IEEE, ISI Web of Knowledge, Science Direct, Springer* and *Wiley Interscience*. By combining with search strings and exclusion criteria, there are 62 primary studies chosen out of 443 works. *Figure 3.2* presents the publication year distribution of 62 studies. *Table 3.3* shows the name and publication platform for each primary study, and *Figure 3.3* demonstrates the citation relationship between these studies.

The UML-based test case generation approaches are derived by analysing and summarising each primary study. The detailed reviews for each primary study are demonstrated in *Appendix A* to have a better layout.

Table 3.4 demonstrates the review results. The first column is the label of each primary study. The second column is the UML diagram used in primary studies, and the third col-

Table 3.3: Publications Details

NO.	Name	Platform
1 [58]	Test Cases Generation from UML State Diagrams	IEEE
2 [59]	Automated Test Case Generation from Dynamic Models	Springer
3 [60]	Automated Generation of Statistical Test Cases from UML State Diagrams	IEEE
4 [61]	Test Case Generation for UML Statecharts	ISI
5 [62]	UML-Based Statistical Test Case Generation	ISI
6 [63]	A Method for the Automatic Generation of Test Suites from Object Models	Science Direct
7 [64]	Formal Test-Case Generation for UML Statecharts	IEEE
8 [65]	Boundary Value Testing based on UML Models	IEEE
9 [66]	Formal Test Generation from UML Models	Springer
10 [67]	Test Cases Generation from UML Activity Diagrams	IEEE
11 [68]	Automatic Test Case Generation from UML Communication Diagrams	ISI
12 [69]	A State-Based Approach to Integration Testing Based on UML Models	Science Direct
13 [70]	Automatic Test Case Generation from UML Sequence Diagrams	IEEE
14 [71]	Improving Test Coverage for UML State Machines using Transition Instrumentation	Springer
15 [72]	Automatic Test Case Generation from UML Models	IEEE
16 [73]	Test Case Automate Generation from UML Sequence Diagram and OCL Expression	IEEE
17 [74]	Test Case Generation by means of UML Sequence Diagrams and Labeled Transition Systems	IEEE
18 [75]	Automatic Test Case Generation using Unified Modeling Language UML State Diagrams	IEEE
19 [76]	Deriving Input Partitions from UML Models for Automatic Test Generation	Springer
20 [77]	Automatic Generation of Test Specifications for Coverage of System State Transitions	Science Direct
21 [78]	Model Based Functional Testing using Pattern Directed Filmstrips	IEEE
22 [79]	Research on Method of Object-Oriented Test Cases Generation Based on UML and LTS	IEEE
23 [80]	UML Activity Diagram-Based Automatic Test Case Generation For Java Programs	ISI
24 [81]	Test Case Generation from UML Subactivity and Activity Diagram	IEEE
25 [82]	Model-Based Software Regression Testing for Software Components.	Springer
26 [83]	A Novel Approach to Generate Test Cases using Class and Sequence Diagrams	Springer
27 [84]	A Hybrid Genetic Algorithm Based Test Case Generation using Sequence Diagrams	Springer
28 [85]	Generation of Executable Test Cases Based on Behavioral UML System Models	ACM
29 [86]	A Novel Approach of Test Case Generation for Concurrent Systems using UML Sequence Diagram	IEEE
30 [87]	A Search-Based OCL Constraint Solver for Model-Based Test Data Generation	IEEE
31 [88]	A Specification-Based Test Case Generation Method for UML OCL	Springer
32 [89]	Automated Test Case Generation for Object Oriented Systems Using UML Object Diagrams	ISI
33 [90]	Test Case Generation and Prioritization from UML Models	IEEE
34 [91]	Automatic Test Case Generation for UML Collaboration Diagrams	ISI
35 [92]	Synthesis of Test Scenarios using UML Activity Diagrams	Springer
36 [93]	Feedback-Directed Test Case Generation Based on UML Activity Diagrams	IEEE
37 [94]	Construction of Test Cases from UML Models	Springer
38 [95]	Combining UML Sequence and State Machine Diagrams for Data-Flow Based Integration Testing	Springer
39 [96]	Test Case Design using Slicing of UML Interaction Diagram	Science Direct
40 [97]	Generation Test Case from UML Activity Diagram Based on AC Grammar	IEEE
41 [98]	Extenics-Based Test Case Generation for UML Activity Diagram	Science Direct
42 [99]	The Web Services Composition Testing Based on Extended Finite State Machine and UML Model	IEEE
43 [100]	Improving Test Case Generation from UML Statecharts by using Control, Data and Communication Dependencies	ISI
44 [101]	Dataflow Test Case Generation from UML Class Diagrams	IEEE
45 [102]	Generating Test Data from A UML Activity using the AMPL Interface for Constraint Solvers	Springer
46 [103]	The Research on Test Case Generation Technology of UML Sequence Diagram	IEEE
47 [104]	SeTGaM Generalized Technique for Regression Testing Based on UML-OCL Models	IEEE
48 [105]	A Novel Approach for Test Case Generation from UML Activity Diagram	IEEE
49 [50]	Test Case Generation and Optimization using UML Models and Genetic Algorithm	Science Direct
50 [106]	A Hybrid Test Case Model for Medium Scale Web Based Applications	IEEE
51 [107]	Automated Model Driven Testing using AndroMDA and UML2 Testing Profile in Scrum Process	Science Direct
52 [108]	Test Case Generation for Concurrent Systems using UML Activity Diagram	IEEE
53 [109]	Automated Test Case Generation from UML Activity Diagram and Sequence Diagram using Depth First Search Algorithm	Science Direct
54 [110]	Synthesizing Test Scenarios in UML Activity Diagram using A Bio-Inspired Approach	Science Direct
55 [111]	Coverage Criteria for Test Case Generation using UML State Machine Diagram	ISI
56 [112]	Test Case Generation for Embedded System Software using UML Interaction Diagram	ISI
57 [113]	Generating and Evaluating Effectiveness of Test Sequences using State Machine	Springer
58 [114]	Agent-Based Regression Test Case Generation using Class Diagram Use Cases and Activity Diagram	Science Direct
59 [115]	A Memorization Approach for Test Case Generation in Concurrent UML Activity Diagram	ISI
60 [116]	A Search-Based Approach to Generate MC DC Test Data for OCL Constraints	Springer
61 [117]	Validating Object-Oriented Software at Design Phase by Achieving MC DC	Springer
62 [118]	Transition Coverage Based Test Case Generation from State Machine Diagram	Science Direct

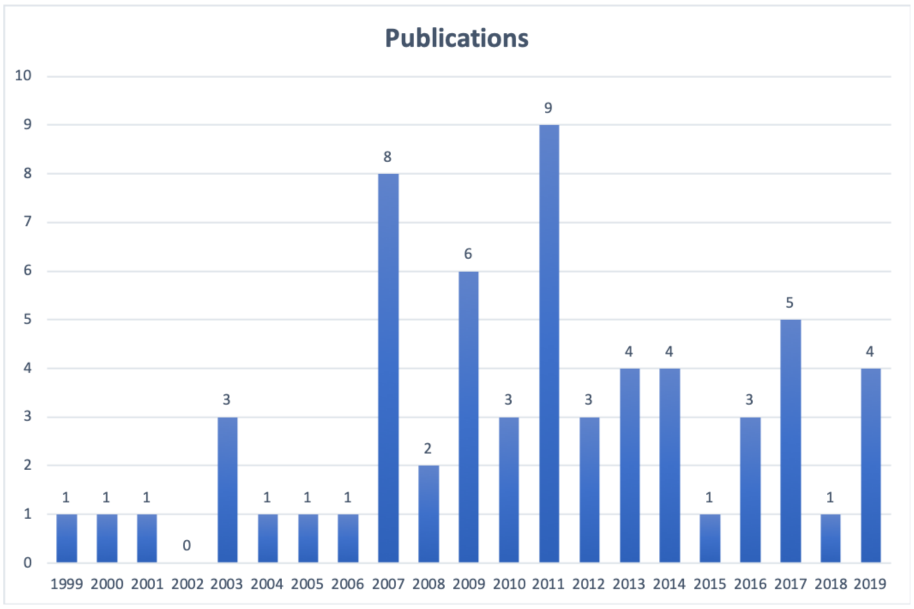


Figure 3.2: Distribution of Publication Year

umn is the intermediate format used in each approach. The fourth column is the number of case studies in each work, and the fifth column is the coverage criteria that need to be satisfied. The execution column indicates the execution methods for the generated test cases, where M represents manually and A stands for automatically. The final column is whether the generated test cases have been optimised.

Through reviewing all primary studies, the common process of UML-based test generation can be described as *Figure 3.4*. The UML model and test criteria are typically required in order to generate test cases. The majority of the primary studies need to transfer the UML model to an intermediate format, but in some approaches, the step of constructing the intermediate structure is omitted.

Based on the coverage criteria and combined with different test case generation algorithms, the abstract test cases can be extracted from the intermediate form or directly from the UML diagram. The generated abstract test cases cannot be executed directly, and most works need to create executable test cases manually. Once the executable test cases are gener-

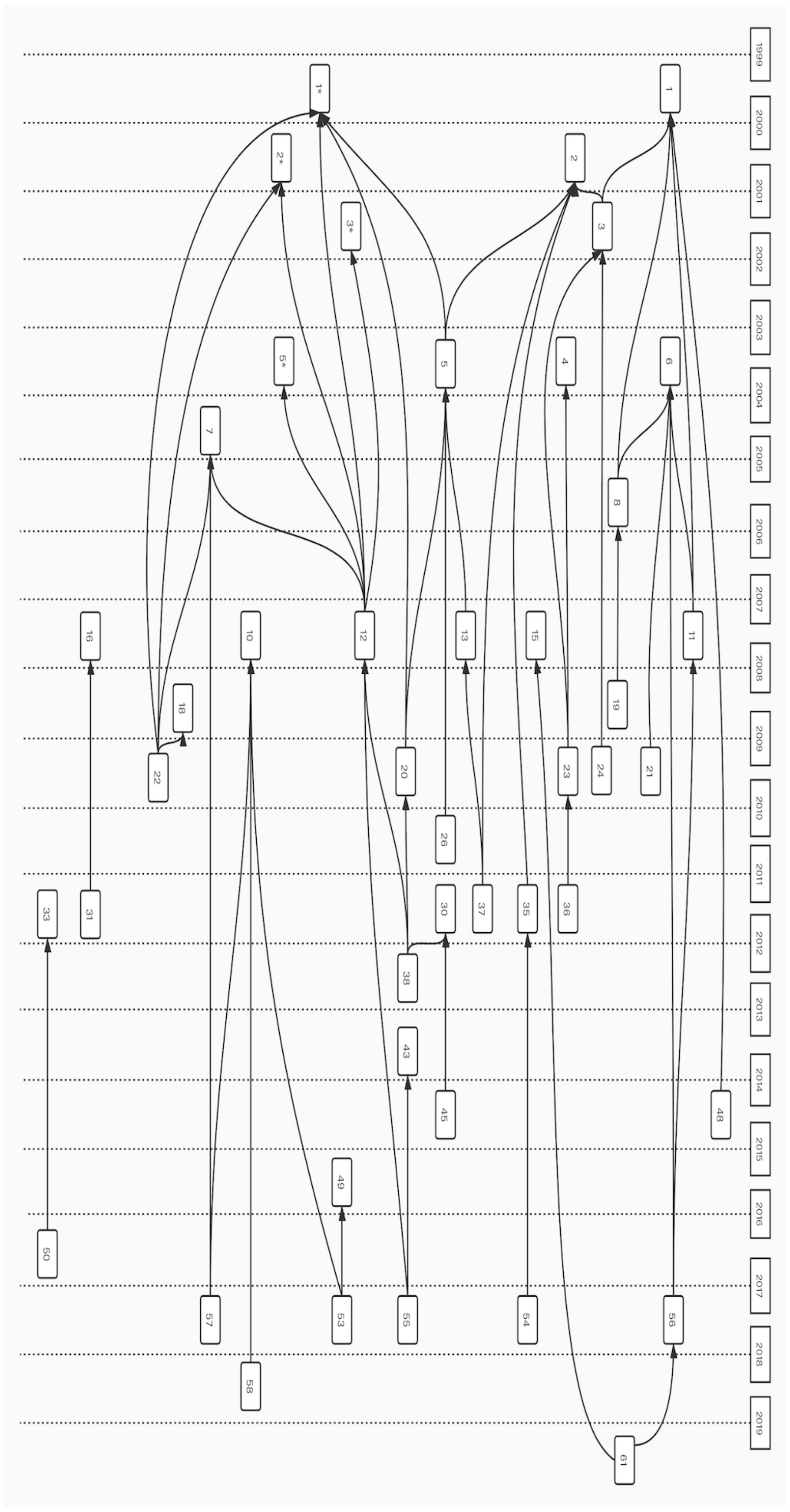


Figure 3.3: Citation Map

Table 3.4: Review Results

NO.	Model Type	Intermediate Format	Number	Coverage Criteria	Execution	Optimization
1	State Machine Diagram	Extended Finite State Machine	1	Data Flow Criteria	M	N
2	State Machine Diagram	STRIPS Planning Problems	1	Branch Coverage	M	N
3	State Machine Diagram	NULL	1	Transition Coverage	M	N
4	State Machine Diagram	Labelled Transition System	1	NULL	M	N
5	Use Case	State Machine Diagrams & Usage Model	1	Transition Coverage	M/A	Y
6	Class, State Machine Diagram, Object Diagram	IF Language	0	Branch Coverage	M	N
7	State Machine Diagram	IOLTS	1	NULL	M/A	N
8	State Machine Diagram	NULL	1	Predicate, Transition Path Coverage	M	Y
9	Fondue UML	CO-OPN	1	NULL	M	Y
10	Activity Diagram	I/O Explicit Activity Diagrams	1	All Path Coverage	M	Y
11	Communication Diagram	Communication Tree	1	Message Path, Full Predicate Coverage	M	Y
12	Collaboration, State Machine Diagram	SCOTEM	1	All Path Coverage	A	N
13	Sequence Diagram	Sequence Diagram Graph	1	All Path Coverage	M	N
14	State Machine Diagram	NULL	2	Improved MC/DC Coverage	M	N
15	Use Case, Sequence Diagram	System Testing Diagram	1	Use Case Dependency, Message Paths	M	N
16	Sequence Diagram, OCL	Scenario Tree	1	Scenario Paths Coverage	M	N
17	Sequence Diagram	Labelled Transition System	2	All Path Coverage	M	N
18	State Machine Diagram	NULL	1	Predicate Coverage	M	Y
19	Class, State Machine Diagram, OCL	Test Case Tree	1	All One Loop Path Coverage	M	N
20	Use Case, Sequence, State Machine Diagram	System State Graph	4	Transition Path Coverage	M	N
21	OCL	NULL	1	NULL	M	N
22	State Machine Diagram	Labelled Transition System	1	Complete Transition Path Coverage	M	N
23	Activity Diagram	NULL	1	Simple Path, Trail Coverage	M	Y
24	Activity Diagram	Composition Tree	1	Branch Coverage	M	N
25	Sequence Diagram	Control Flow Graph	1	NULL	M	N
26	Class, Sequence Diagram	NULL	1	Transition Coverage	M	N
27	Sequence Diagram	NULL	1	Message Sequence Coverage	M	Y
28	State Machine Diagram	Symbolic Transition System	1	NULL	A	N
29	Sequence Diagram	Concurrent Composite Graph	1	Message Path Coverage	M	N
30	OCL	NULL	1	NULL	M	Y
31	OCL	High Order Logic Expression	1	NULL	A	N
32	Object Diagram	Weighted Graph	1	Message Coverage	M	N
33	Activity Diagram	Extend Activity Diagram	1	Transition Coverage	M	Y
34	Communication Diagram	Weighted Graph	1	NULL	M	N
35	Activity Diagram	Intermediate Testable Model	4	Selection, Loop Adequacy, Concurrent Coverage	M	N
36	Activity Diagram	NULL	1	Simple Path Coverage	A	N
37	Use Case, Sequence, Class Diagram, OCL	Sequence Diagram Graph	1	All Path Coverage	M	N
38	Sequence, State Machine Diagram	Control Flow Graph	1	Coupling-Based Data Flow	M	N
39	Sequence Diagram	Message Dependency Graph	1	Message Path, Slice Coverage, Boundary Testing	M	N
40	Activity Diagram	Activity Convert Grammar	1	All Path Coverage	M	N
41	Activity Diagram	Euler Circuit	1	Transition Coverage	M	Y
42	Sequence, Extended State Machine Diagram	EFSM-SeTM	1	All Path Coverage	M	N
43	State Machine Diagram	LOTOS	8	Transition Coverage	M	Y
44	Class, State Machine Diagram	Control Flow Graph	1	Path, DU Pairs Coverage	M	N
45	Activity Diagram	A Mathematical Programming Language	2	Control Flow Coverage	A	N
46	Sequence Diagram	Scene Test Tree	1	State Coverage	M	N
47	Class, State Machine Diagram	SMT Language	2	NULL	M	N
48	Activity Diagram	Activity Flow Graph	1	Path Coverage	M	Y
49	State Machine Diagram, Sequence Diagram	System Testing Graph	1	Path Coverage	M	Y
50	Activity Diagram	Weighted Base Graph	1	Path Coverage	M	N
51	Sequence Diagram	UML 2.0 Testing Profile	1	NULL	M	N
52	Activity Diagram	Input/Output Activity Diagram	1	Path Coverage	M	Y
53	Sequence, Activity Diagram	System Testing Graph	1	Path Coverage	M	N
54	Activity Diagram	Intermediate Testable Table	8	Path Coverage	M	Y
55	State Machine Diagram	State Graph	0	NULL	M	N
56	Interaction Diagram	Stimulus Linking Table	2	Object, Message Path, Condition Coverage	M	N
57	State Machine Diagram	Composition Control Flow Graph	1	Path Coverage	M	N
58	Use Cases, Activity, Class Diagram	NULL	1	NULL	M	N
59	Activity Diagram	Concurrent Activity Graph	1	Causal Ordering Coverage	M	Y
60	OCL	NULL	4	MC/DC Coverage	M	Y
61	Activity Diagram	JAVA Code	4	MC/DC Coverage	M	N
62	State Machine Diagram	State Machine Intermediate Graph	2	All Transition, Round Trip Path, All Transition Pair Coverage	M	N

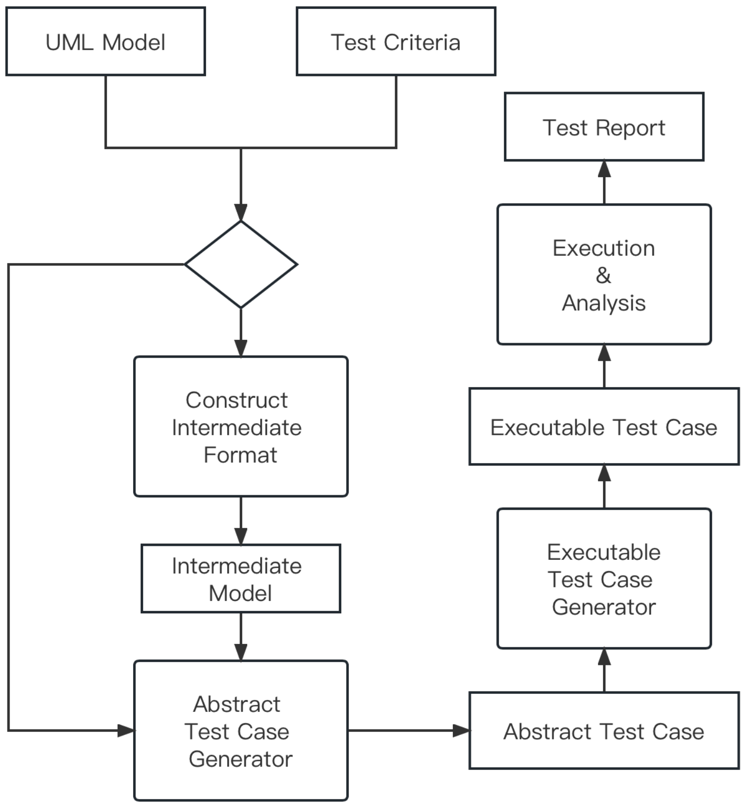


Figure 3.4: Common Process in MBT

ated, execution and analysis of these test cases can be carried out in various ways. In some primary studies, the proposed approach only describes the process until the abstract test cases are constructed.

After reviewing the 62 primary studies and categorising the related information, the answers to each research question are as followings.

• *Question 1:* Which UML model(s) is used by the primary study?

The UML models are categorised as structural models and behavioural models. Most primary studies used behavioural models to perform the proposed approach since structural models typically concern the structure of SUT on a higher abstract level. The structural models contain general information without detailed implementation information, and this situation leads researchers to prefer using behavioural models [119]. The UML models used in primary studies focus on the state machine, use case, class, object, activity, sequence diagrams, OCL expression and others. *Table 3.5* shows the number of different UML models used in primary studies.

Table 3.5: Model Usage Statistic

UML Model	Number of Studies
State Machine Diagram	22
Use Case	5
Class Diagram	7
Object Diagram	2
Activity Diagram	17
Sequence Diagram	17
OCL Expression	7
Others	6
Total	*83*

The statistical results show that the primary studies prefer using state, activity and sequence diagrams, which are all behavioural models, to perform the test case generation process. Meanwhile, the other most commonly used standards are the OCL expression and the structural model class diagram.

The behavioural model provides a more detailed description of the SUT, this information provides more detail to guide the test cases generation process, but this is only restricted to a single functionality or simple system behaviour. The structural model or OCL specification provides a high-level overview of the SUT, and the created test cases can comprehensively test the system. Still, the lack of system information makes test cases more challenging to generate. There are 83 models used in 62 primary studies, which means some proposed approaches used more than one type of UML model. These approaches combined the benefits from behavioural and structural models, which can better guide the test case generation process.

- *Question 2:* Whether the primary study required the intermediate format?

We found that the majority of proposed approaches perform the test cases generation process with an intermediate format, and only 12 out of 62 works without any intermediate form.

Different types of intermediate formats can be used to generate test cases. The original UML diagrams may not contain enough details or are not suited for extracting the test cases directly, especially for structural models. The majority of intermediate formats are in one of the tree formats, and the proposed approaches then apply a tree traversal algorithm to construct test cases [120], like the breadth-first search algorithm or depth-first search algorithm, to traverse the tree to create test cases.

- *Question 3:* Which coverage criteria does the primary study intend to satisfy?

The primary studies address various coverage criteria, but 13 out of 62 primary studies do not mention which coverage strategy is used by the proposed approach.

Coverage criteria is the measurement used to describe the extent of the source code or model covered by the test suite. Coverage criteria intend to guarantee that the generated test cases meet the pre-defined goals in the testing process. The majority of intermediate formats in primary studies are in tree formats, and the test cases are generated by traversing the intermediate form, so most of the coverage criteria are different forms of path coverage. The most substantial coverage is all path coverage, which requires the traversal of all possible paths in the graph. But without restrictions, the created test cases are vulnerable to the paths explosion problem [121], especially if loops exist in the intermediate format. More than twenty proposed approaches in the primary studies used path-related coverage criteria.

Another valuable and robust coverage criteria is MC/DC coverage [1], which focuses on the combination of conditions within decision expressions [122]. MC/DC coverage is always used in safety-critical software, especially in aircraft and automotive systems.

- *Question 4:* Whether the generated test cases can be executed automatically?

Only 7 out of 62 primary studies partially or fully support the automatic execution of generated test cases. Most of the proposed approaches require executing the constructed test case manually. Moreover, they can only create abstract test cases instead of executable ones, which misses the step that transfers abstract test cases to executable ones.

The UML models hide the implementation details of the corresponding software system, such as the programming languages or platforms. Due to the platform-independent character that the proposed approaches cannot generate executable test cases directly is reasonable. The transformation between abstract and executable test cases is relatively easy to perform. The users can create executable test cases when there is a need to test the software system on a specific platform and sufficient implementation details are available. Once the executable test cases are generated, the test

[1]Each test needs to independently affect the outcome of the corresponding decision expression.

execution process can be performed in various ways, either manually or automatically.

• *Question 5:* Whether the generated test cases have been optimised?

The majority of faults can be detected by a small subset of test cases, while the generated test cases always contain redundant test cases. How to prioritise these test cases, eliminate unnecessary test cases and find the most effective test execution sequence are essential issues.

18 out of the 62 studies propose the optimisation of the generated test cases. Most test optimisation strategies concern reducing the number of test cases whilst guaranteeing the corresponding coverage criteria [123]. The most common coverage criteria used in test cases optimisation is MC/DC coverage, which is that each condition in a decision predicate can independently affect the decision result. In the DO-178B and DO-178C standard [116], MC/DC coverage is the criteria that must be satisfied by the most critical software.

Another common optimisation strategy is by using a stochastic algorithm, most often a genetic algorithm. But all the primary studies which included a stochastic optimisation algorithm only proposed the approach in theory and only provided an elementary case study. A stochastic algorithm optimises the test cases by re-ordering the execution sequence (TCP process) of the test cases, which lets the test cases have the strong ability to find the faults that will be executed early.

The optimisation of the constructed test case is seldom mentioned in primary studies, although this process will benefit the testing activity further. Within the primary studies, the most common optimisation is reducing the number of test cases while maintaining the corresponding coverage criteria (TCM process). The exploration of other optimisation methods, like TCP and TCS processes, still requires further investigation.

3.1.4 Quality Assessment

The process that applies quality criteria to primary studies is a critical step in the systematic literature review. Five research questions are proposed above to assess the quality of selected primary studies.

Table 3.6 demonstrates the quality assessment result for each primary study by quality score. The first column is the label of each study, and the following six columns are the score for each assessment question and the total score. The score of each evaluation can be 0(Not Mentioned), 0.5(Partially Satisfy) and 1(Fully Satisfy).

Following are the scoring criteria for each question:

Question 1: 1. *Fully Satisfy*: The abstract and introduction clearly represent the aim and are consistent with the research topic. 2. *Partially Satisfy*: The abstract and introduction are related to the research topic but not well represented.

Question 2: 1. *Fully Satisfy*: The answers to research questions are easy to find and conclude. 2. *Partially Satisfy*: The answers to research questions are vague and hard to find or miss some research questions.

Question 3: 1. *Fully Satisfy*: The case study is described clearly, and the experiment results are also clearly demonstrated. 2. *Partially Satisfy*: The case study is presented but not well demonstrated.

Question 4: 1. *Fully Satisfy*: The study demonstrates the shortage and future work or describes either aspect explicitly.

Table 3.6: Assessment Result

NO.	R1	R2	R3	R4	R5	*Score*
1	1	1	0.5	1	1	*4.5*
2	1	1	1	0.5	1	*4.5*
3	1	0.5	1	0.5	1	*4*
4	1	0.5	0.5	0.5	1	*3.5*
5	1	1	0.5	1	1	*4.5*
6	1	1	0	0.5	1	*3.5*
7	1	1	1	1	1	*5*
8	1	0.5	0.5	0.5	1	*3.5*
9	1	0.5	1	0.5	1	*4*
10	1	1	1	1	1	*5*
11	1	1	1	0.5	1	*4.5*
12	1	1	1	1	1	*5*
13	1	1	0.5	0	1	*3.5*
14	0.5	0.5	0.5	1	1	*3.5*
15	1	0.5	0.5	0	1	*3*
16	1	1	1	0	1	*4*
17	0.5	0.5	1	0.5	0.5	*3*
18	1	0.5	0.5	0.5	1	*3.5*
19	1	1	0.5	0.5	1	*4*
20	1	1	1	0.5	1	*4.5*
21	0.5	0.5	0.5	0.5	1	*3*
22	1	0.5	1	0.5	0.5	*3.5*
23	1	1	1	0.5	1	*4.5*
24	1	1	1	0.5	0.5	*4*
25	1	0.5	1	0.5	1	*3.5*
26	1	0.5	0.5	0.5	1	*3.5*
27	0.5	0.5	0.5	0	1	*2.5*
28	1	0.5	0.5	0.5	1	*3.5*
29	1	1	1	0.5	0.5	*4*
30	0.5	0.5	0.5	1	1	*3.5*
31	1	0.5	1	1	1	*4.5*
32	1	0.5	1	0.5	0.5	*3.5*
33	1	0.5	0.5	1	1	*4*
34	1	0.5	1	0	1	*3.5*
35	1	1	1	1	1	*5*
36	1	0.5	0.5	0	1	*3*
37	1	0.5	1	0	1	*3.5*
38	1	1	0.5	0.5	1	*4*
39	1	1	1	0.5	1	*4.5*
40	1	0.5	1	0.5	1	*4*
41	1	0.5	0.5	0.5	1	*3.5*
42	1	1	1	0.5	1	*4.5*
43	0.5	1	1	0.5	1	*4*
44	1	0.5	1	0.5	1	*4*
45	1	1	1	1	1	*5*
46	0.5	1	0.5	0	0.5	*2.5*
47	0.5	0.5	1	1	0.5	*3.5*
48	0.5	0.5	1	1	1	*4*
49	1	1	0.5	0.5	1	*4*
50	1	0.5	1	0	1	*3.5*
51	1	0.5	0.5	0.5	1	*3.5*
52	1	0.5	1	0	1	*3.5*
53	1	1	1	0.5	1	*4.5*
54	1	1	1	1	1	*5*
55	0.5	0.5	0.5	0	0.5	*2*
56	1	1	1	0.5	1	*4.5*
57	1	1	1	1	0.5	*4.5*
58	1	0.5	0.5	0.5	0.5	*3*
59	1	0.5	1	0.5	1	*4*
60	1	1	1	1	1	*5*
61	1	1	0.5	1	1	*4.5*
62	1	1	1	0.5	1	*4.5*

2. *Partially Satisfy*: The publication only simply mentioned the shortage or future work.

Question 5: 1. *Fully Satisfy*: The conclusion is consistent with the goal and accurately represented. 2. *Partially Satisfy*: The conclusion is consistent with the goal to a large degree and simply represented.

When the assessment score is above four, the publication is meaningful and valuable, and a score above three means reasonable. The assessment score distribution chart is shown as *Figure 3.5*. The average score for the overall 62 primary studies is 3.91, which means the overall quality of primary studies is persuasive. Only three primary studies have an assessment score below 3, but 7 works have the full score. The percentage of valuable (assessment score above 4) is 56.4, and the reasonable rate is 38.7%.

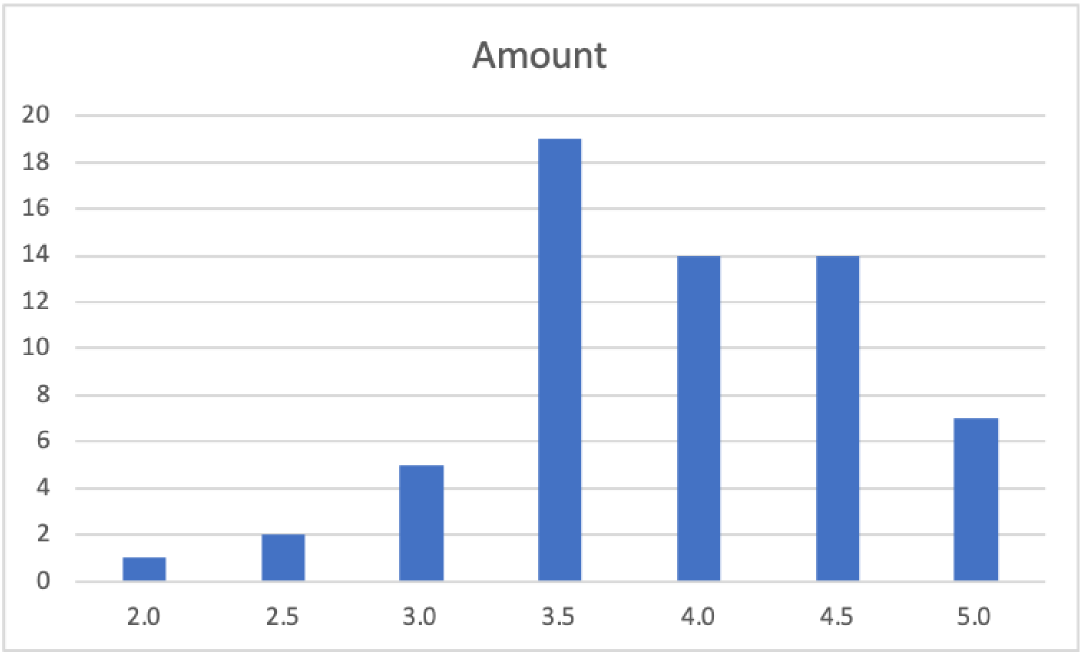


Figure 3.5: Assessment Score Distribution

3.1.5 Discussions

Some discussions related to this systematic literature review are presented in this part.

So far, this systematic literature review has presented the different approaches relevant to UML-based test case generation. The test cases generation step is a crucial phase in model-driven engineering, and the typical process in MBT is shown as *Figure 3.4*.

By reviewing 62 selected primary studies, the proposed approaches have similarities and differences. The review focuses on the model type, intermediate format and coverage criteria. On the one hand, the methods have similarities, and all studies follow the common process, even though some may skip or miss some steps. On the other hand, the proposed works are different, in which separate studies use various strategies to generate test cases and satisfy coverage criteria.

Most primary studies use behavioural models to perform the test case generation process because these models are more related to the implementation of SUT. However, the drawback of behavioural models is that they tend to concentrate on modelling either an individual function or a small group of interconnected functions. This specificity, while precise, may narrow the scope of the model, thereby overlooking the inherent interconnections among system attributes. The structural models or OCL specifications can generate test cases that cover more aspects of the SUT but need more detailed information to guide and help the generation process. Most of the approaches need an intermediate format. And then by transferring the original UML models to the intermediate structures to construct the corresponding test cases.

Most primary studies use the tree-like graph as the intermediate format to perform the test case generation process. Most coverage criteria are related to path coverage. The search-based algorithm traverses the intermediate form to find the corresponding paths according to the coverage criteria, and each route will be a single test case. The proposed approach applied some coverage criteria, like all path coverage, to construct test cases. If without any restriction, the test case explosion problem will be caused by the loops that existed in the intermediate format or the UML model. If the model contains loop(s), the number of all potential paths would be infinite. Many primary studies use expression condition coverage criteria to perform the test case generation process, for example, MC/DC and branch coverage. These criteria discover the potential condition combination within the decision expression, and these criteria are effective.

The MBT can apply almost to all types of systems, but MBT is especially powerful when facing embedded systems, distributed systems, concurrent systems, mission-critical systems and large-scale systems. The need for the testing phase is significant, with the scale of modern software systems increasing rapidly. Manually generating test cases is labour-intensive, and it seems impossible to create them exhaustively. In the meantime, testing after the implementation phase will delay the SDLC and increase the development budget. Generating test cases from the design model is an excellent approach to start the testing in the early stage of SDLC. The created test cases can guide and help the implementation process of the corresponding software system.

Following are some future orientations that may need further exploration by the researchers in the MBT community.

The state explosion is a tricky problem in many approaches, especially when generating test cases from the state chart diagram, activity diagram and sequence diagram. Most works use the search-based algorithm to traverse the UML model or intermediate format to create test cases, and each path corresponds to one specific test case. But when the model contains any loop-like event or transition, infinite paths can be found in theory. Although some approaches use techniques to avoid infeasible paths or use particular coverage criteria to prevent endless loops, these methods cannot always solve the problem well. And in some case studies are hard to determine the loop times when the model contains the specific trigger event. For example, input the wrong password three times and then trigger one particular event. The difference between the standard and this loop type is challenging to distinguish in this situation. How to minimise state space and promise the coverage standard is a valuable research aspect.

Concurrent and distributed systems are becoming increasingly popular, and the UML standard supports describing these systems. The interaction sequences of these systems are complicated and unpredictable. If the generated test cases exhaustively cover all possible combinations, there will be another state explosion problem. Some approaches tried to solve this problem by adding constraints on the software system or UML models. Still, the effectiveness of the method is limited and only available for particular types of systems. How to carry out test case generation in the concurrent and distributed system is a valuable and promising research area in MBT.

The TCO is a significant aspect of the testing process. Some approaches use coverage criteria to optimise generated test cases, the most frequently used one being MC/DC coverage. This method is powerful, but when dealing with complex conditions, the complexity is high while the efficiency is low. And some approaches use the stochastic algorithm, like the genetic algorithm, to create the most effective test suites. Most works applied to this method only proposed the approach in theory or applied it to simple case studies, which are challenging to reproduce. Efficiency optimising the generated test cases and maintaining feasibility still need more investigations.

The possible solution to optimise generated test cases is to prioritise (re-order) the generated test cases. Due to the balance between expensive time cost and software quality, executing all test cases is time-consuming. A small subset of test cases can find the majority of system faults. Different test cases have their own ability to reveal the defects within the software system. Executing the test cases earlier, which have the strong ability to find the error, will bring more advantages. Therefore, the right strategy to prioritise and order the test cases can make the testing more efficient. Those test cases that are most likely to find the faults will give high priority and will execute earlier in the test cases execution process. This process may be guided by the stochastic search or machine learning algorithm, like the genetic or ant colony algorithm. The possible validation method could be mutation testing. Mutation testing will assess the improvement of the TCP. The ordered and unsorted test suites will perform the same mutation testing together. If the prioritised test suite can achieve the same level of mutation score by using fewer test cases, this will show it is effective. And also, some indicators, like APFD (Average Percentage of Fault Detection), will assess how quickly the given test suite detects system defects. Moreover, the TCM will also benefit the testing process by eliminating redundant test cases to speed up software testing.

The discrepancy between abstract and executable test cases is still a gap, although the transformation is relatively easy to achieve. The entire MBT process needs transformation to complete the whole test chain. In the meantime, the test criteria are also an important research point, which determines whether the given test cases pass the tests or not. Combining the transformation process and test criteria is a necessary and significant process within the MBT.

Most works only validate the proposed approach on small-scale case studies, and there is still a gap between academia and industry. The majority of strategies only use a single or few numbers of simple models as experiments, but in reality, the models of the software project are much more complicated. Even though the work demonstrated a satisfactory investigation result, this cannot promise effectiveness on a real-world project. How to apply the test case generation techniques to industry needs constant exploration.

3.1.6 Threats to Validity

Each systematic literature review is vulnerable to numerous validation threats, which must be pointed out and dealt with as much as possible. In the following, some threats to validity are discussed.

The first threat is the search strings. The systematic literature review aims to analyse all possible works in the corresponding research area. Manually searching the potential works from numerous platforms seems impossible and unfeasible, even only focusing on a limited number of databases. The process of constructing a search string is an empirical method and is mainly based on experience to determine the content of strings. Even though constructing the search strings carefully and performing the automatic search of the relevant databases, there is no promise to include and select all possible research approaches. For example, the works published in technical reports, company journals or written in other languages. In this literature review, the neglected works also have crucial contributions and affect the completeness of this review. The inclusion search string and exclusion criteria must be amended and reviewed continually to deal with this threat. The step to confirm primary studies must read each approach and distinguish every work carefully. The inclusion and exclusion criteria for this review are systemically constructed, and the information to answer the research questions is extracted through reading all potential publications.

The second threat is about the primary studies themselves. Even select the works as carefully as possible, the quality of these studies is essential. The research publications most likely contain publication bias, which means the authors publish positive rather than negative results. This threat significantly affects the quality of the review but cannot control by us. The most feasible method to solve this problem is rigorous reading and analysis of the primary studies. By systemically literature review, the impacts of this threat can be minimised.

The other potential threat is the standard of assessing the quality of primary studies. In this systematic literature review, five questions are proposed to determine and evaluate the quality of each study, but there is no guarantee these questions will cover all quality standards. The proposed questions focus on the most necessary aspects of the assessment. The quality of primary studies is all on different standard levels, so the assessment process is critical. The quality assessment questions are carefully constructed and reviewed to ensure the evaluation of the primary studies. There may be some threats to the review, and here only present three main threats.

3.1.7 Conclusion

Typically, manual or exhaustive testing is unfeasible for the real-world software system because of the large number of inputs combination and system functions. Test cases, which can be used to detect the flaws within SUT, are an essential part of system testing, and test case generation is a crucial challenging phase in the testing phase. MBT provides several benefits to the generation process, for example, reducing generation time and making the testing start early. The UML has already become a de-facto standard both in academia and industry. Based on research trends and reality requirements, the review on this topic is meaningful and valuable.

The systematic literature review is based on UML-based test case generation, analysing 62 primary studies extracted from 443 papers, and the selected studies range from 1999 to 2019. The review mainly focuses on the model type, intermediate format and coverage criteria. Based on the review points, five research questions are proposed. And five assessment questions are used to assess the quality of selected primary studies.

The review results from the primary studies are presented in the review. This systematic literature review analysed each primary study in detail, and the different proposed approaches showed the consensus of the common process of MBT. But all primary reviews have differences in the model usage, coverage criteria, intermediate format, execute the test cases automatically or manually, and whether the proposed approach optimises the generated test cases.

Moreover, the threats to this SLR are also mentioned in this review. Through the review of the selected primary studies, the potential of MBT has yet to be fully explored, and many open questions need to be discovered. Some obstacles do not intuitively belong to the domain of MBT, but these have some interrelationships with MBT. And some other research area techniques may bring benefit to the research topic, like stochastic algorithm. Some future research orientations that may need to be investigated in the MBT community are presented in this review.

Finally, this systematic literature review brings the overall evaluation of the UML-based test case generation in MBT and may bring future research directions. This review is completed by reviewing selected primary studies and answering research and quality assessment questions to collect the related information. The test case generation is still a significant phase in system testing. The future work will not only focus on the test case generation process but also extend to the TCO processes.

3.2 Test Case Optimisation

There are three mainly used types of optimisation strategies, which are test case prioritisation, test case minimisation and test case selection. TCP aims to re-order the original test cases to detect the maximum defects as early as possible. TCM aims to reduce the number of test cases within the test suite by eliminating redundant test cases. TCS aims to categorise test cases to determine which test cases need to be executed for the new system.

The main difference between the three TCO techniques is that TCS aims to categorise the existing test suite into different groups and choose a specific subset of test cases to re-test the updated system. However, TCM concerns with identifying and removing redundant or unnecessary test cases within the original test suite. Both TCS and TCM select the subset of the previous test suite, which means the number of test cases will decrease. By contrast, TCP will preserve the whole previous test suite. The prioritisation process improves the testing ability by re-ordering the test cases within the test suite, so the faults within the software systems can be detected earlier.

In the following parts, we will discuss the overview, main approaches and related works for these three optimisation processes.

3.2.1 Test Case Prioritisation

TCP aims to re-order the sequence of test cases within the test suite that helps the system tester to achieve the maximum testing benefit, even if the testing procedure is prematurely halted [124].

The TCP problem can formally be defined as [125]:

Define: Test Case Prioritisation Problem

Given: T, a test suite, PT, the set of permutations of T, and f, a function from PT to the real numbers.

Problem: Find $T' \in PT$, such that $(\forall T'')$ $(T'' \in PT)$ $(T'' \neq T')$ $[f(T') \geq f(T'')]$.

And in this definition, PT is all possible combinations of prioritisation of set T. And f is a function that, applied to a specific ordering combination, returns an award value for that ordering.

From the above definition, to find the potential best solution T', the prioritisation algorithm must define the set of every permutation PT of test cases. And then, choose the T', which can maximise the function f.

Analysing every combination of test cases is almost infeasible in practice, especially when the test suite size is big. Assuming a test suite contains n test cases, the size of PT is $n!$. This situation may transfer the TCP problem to an instance of the Traveling Salesman Problem, which is a famous NP-complete problem [126]. Therefore, the TCP problem is always solved by a heuristics algorithm, like the genetic algorithm or ant colony algorithm, using their strong searching capacity to construct the sequence of test cases.

TCP can reduce the testing budget by re-ordering the sequence of test cases in the test suite so that the critical test cases can be executed earlier. In the test suite, test cases have different fault detection abilities. Due to time and resource constraints, executing all test cases within the test suite may be impossible. TCP increases the fault detection speed by scheduling the sequence of test case execution. Executing those test cases with the most potent capacity to find errors earlier will bring more advantages to the testing process. Therefore, the correct strategy to prioritise test cases to re-order the test cases can make regression testing more efficient.

TCP does not involve the selection activity to the test suite. It assumes that the whole test suite will be executed, but the testing may be terminated arbitrarily during the testing process.

In an empirical study of prioritisation techniques [125], the authors applied the same algorithm to different coverage criteria to compare the effectiveness. The criteria include branch-total, branch-additional, statement-total, state-additional, FEP-total and FEP-additional ^{2}.

The branch-total approach uses the number of the covered branches by each test case to prioritise the test suite, like the greedy algorithm, while the branch-additional uses the number of additional uncovered branches to prioritise the test suite. Similarly, statement-total and state-additional replaced the branch coverage with statement coverage. The FEP of a test case is measured using program mutation. The FEP-total approach prioritises test cases according to the mutation score of individual test cases, and FEP-additional by using additionally increased mutation scores provided by single test cases to prioritise the test cases.

The study results showed that no coverage criteria consistently outperformed others across different approaches. However, the FEP-additional approach always resulted in a higher fault detection ability.

Another widely used TCP technique is search-based prioritisation. The various implementations of search-based algorithms, for example, the genetic algorithm and ant colony algorithm. The experiment results proved that the test cases optimised by the search-based algorithm reached a generally good result [127]. However, applying a search-based algorithm may differ based on the selected test suite and fitness function. The advantage of search-based prioritisation is the strong search capacity, especially with an enormous search space.

Also, the requirement-based test cases prioritisation. A software system is built to meet pre-defined requirements. Therefore, by using the requirements specifications, the critical test cases can be classified. Test cases are mapped to

^{2}Fault Exposing Potential

the requirements tested by them, and then the prioritisation process can be performed based on various properties. For example, in [128], the authors used client priority, error-prone value, volatility value, and execution difficulties to re-order the corresponding test suite.

Another prioritisation technique is the history-based approach, which uses historical data as input to perform the TCP process. Like [129], they used history information for the test suite, like previous execution counts and which test cases revealed faults, to weight each test case. However, history-based prioritisation is restricted when the available historical execution and fault-detection information is limited.

According to the systematic literature review [130], model-based test cases prioritisation approaches only occupied 4% of selected studies. The model-based test cases prioritisation mainly focused on the state machine diagram and activity diagram using the branch or transition information to prioritise the test cases. There has been more and more attention to model-based prioritisation techniques in recent years, but the research on high-level UML models is still under exploration. Also, the model-based approach always hybrids with other methods, like a search-based algorithm, to perform the prioritisation process.

When the test suite is prioritised, we need a metric to validate the effectiveness of prioritisation, to check whether the new test suite better than the original one and to what extent. APFD is a metric to measure how early will the test suite detect the system faults [131]. APFD is used to calculate the average percentage of fault detection rate for the examined test suite and how early will the test suite detect the system faults. The value of APFD ranges from

0 to 1. A higher value indicates the earlier faults detection capacity. Intuitively, this metric measures how quickly the test suite detects the system faults. Then, to measure the effectiveness of the future TCP approach, we will use the APFD metric. At the same time, the equation of APFD is shown as *Equation 3.1*.

$$APFD = 1 - \frac{TF_1 + TF_2 + \cdots + TF_m}{nm} + \frac{1}{2n} \tag{3.1}$$

In this equation, TF_i indicates the test case which first detects i^{th} fault, n is the number of test cases within the test suite, while m is the number of faults [3] within the software system.

However, the APFD metric considers that every system fault has the same severity and executing each test case has the same testing efforts. In the actual testing process, the test case has a unique testing cost, and each fault may have a different degree of violation of the software system. So, the $APFD_c$ metric is introduced in [132]. This metric takes information about test cost and fault severity into account, and $APFD_c$ improves the accuracy of APFD. The equation of $APFD_c$ is defined as *Equation 3.2*.

$$APFD_c = \frac{\sum_{i=1}^{m}(f_i * (\sum_{j=TF_i}^{n} t_j - \frac{1}{2}t_{TF_i}))}{\sum_{j=1}^{n} t_j * \sum_{i=1}^{m} f_i} \tag{3.2}$$

Similarly to the APFD calculation, in this equation, f_i is the severity of the i^{th} fault. TF_i indicates Test Case which

[3]real or artificially generated defects

detects the i^{th} fault, and t_j is the testing cost of the corresponding test case. Also, n is the number of test cases within the test suite, while m is the number of faults.

Above definition of $APFD_c$ contains the cost and fault severity of the test cases. However, due to its unclear or system-specific estimation method of fault severity and test case cost, the practical feasibility is limited for this metric. Most researchers are willing to use $APFD$ to assess the effectiveness of their works. While the APFD is not suitable for mutation testing ideally, we will discuss more in Chapter 5, and a modified version of the APFD metric is also proposed.

The studies on the TCP process have been on-going for decades, and numerous approaches are proposed to explore this topic. Following are some related works closely related to this research work.

An approach using fuzzy logic to guide the test cases prioritisation process has been proposed in [133] by Rapos. They transferred the UML models to the symbolic execution tree. Then test suite size, symbolic execution tree size, relative test case size and output significant as inputs through 39 fuzzy rules to infer priority. They compared the proposed approach with the random method.

Shin [134] used an alternating variable method to perform a model-based TCP process. They customised a set of mutation rules, applied them to the state machine diagram, and then generated test cases according to mutants. The test cases prioritisation process is based on the fault coverage criteria and guided by the alternating variable method. Then the effectiveness of the proposed approach is examined by the APFD metric.

Pospisil [135] performed the TCP process by improved adaptive random prioritisation (ARP). Compared to the traditional ARP, they replace the original distance function with a multi-criteria decision-making method. Additional information, such as path complexity and coverage information, is used to determine priority.

Pan [136] presented the results of a systematic literature review regarding the application of machine learning techniques to test case optimisation. Also, Gupta [4] revised the multi-objectives and hybrid approaches to the test case prioritisation problem and proposed some future research trends.

Both [137][138] use GA to guide the TCP process. However, in the first article, the proposed approach by Ma is based on a control flow diagram and path information. The latter one, presented by Rattan, uses an extended system dependence graph to perform the prioritisation process.

Sornkliang [139] presented the work, which is directly related to the MDE scope. They assign weight to each node within the UML activity diagram and then prioritise test paths based on the calculation of the path scores. Through five case studies, Chaudhary [140] compared the TCP ability between four unsupervised clustering algorithms. The result showed that the DBK-mean [4] algorithm out-performance all others. Morozov [141] proposed a model-based TCP method based on fault activation analysis and error propagation analysis for the automotive system.

Xing [142] proposed an algorithm on artificial fish school algorithm to perform the TCP process. They define the coding way of the artificial fish, clustering, tail-chasing and foraging behaviours, and the effectiveness of the approach is examined by the APFD metric and EET [5]. The paper also

[4]Density based and Partition based K-Mean Algorithm
[5]Effective Execution Time

discusses how the number of iterations affects the proposed algorithm. Mann [143] also used an evolutionary algorithm to prioritise the generated test cases. They first constructed the test cases by generating random test data to satisfy condition coverage, then through a particle swarm optimisation-based algorithm to conduct the TCP process.

Miranda [144] presented FAST, which is a set of similarity-based techniques, to perform the TCP process. The similarity-based approach employs min-hashing and locality-sensitive hashing algorithms for quickly finding test cases within a big set with maximum diversities. They compared their method with various approaches to evaluate its effectiveness. Li [145] made a deep analysis of greedy additional algorithms for the TCP problem, and proposed an accelerated version of the original algorithm to improve the efficiency while preserving the effectiveness. The proposed approach uses a new data structure to store the previously selected test case information to achieve higher efficiency. The effectiveness of the method is examined by the APFD metric.

Sun [146] proposed a path-directed approach to the TCP process for metamorphic testing. Their approach first analyses feasible paths using symbolic execution and then generate corresponding test cases based on constraint solver. The constructed test cases are prioritised by calculating the distances between them based on the statement coverage.

How the parameters settings and genetic operators choices affect the effectiveness of the genetic algorithm is studied by Bajaj in [147]. Castro-Cabrera [12] conducted a survey on the TCP process, which ranges from 2017 to 2019. The results showed that TCP is still an activity research topic and has a great deal of interest in the literature. Most of the proposed works are based on search-based, coverage-based or similarity-based techniques, and a large proportion of them combined multiple methods to perform the TCP process.

In the domain of regression test case prioritisation, the seminal work [148] offers a comprehensive evaluation of various algorithmic strategies aimed at optimising the order of test cases. They systematically assessed the performance of five distinct algorithms: Greedy, Additional Greedy, 2-Optimal, Genetic Algorithms, and Hill Climbing, across a diverse set of software programs. The study highlighted the complex nature of choosing the right algorithm for test case ordering, especially as software and test suites grow. This research helps in understanding how to efficiently organise test cases in software development, making the testing phase more effective. Moreover, they validated the general effectiveness of the genetic algorithm, which is one of the adapted algorithms within our research.

In this work, five optimisation algorithms are implemented to conduct the TCP process. These algorithms are Genetic Algorithm (GA), Particle Swarm Optimisation (PSO), Firefly, Fish School and Cuckoo Search algorithm.

- *Genetic Algorithm (GA)*

Professor John Holland first introduced the GA in the 1960s, and through DeJong and Goldberg's further development, formed the basic GA [149]. This algorithm is a computational model simulating the natural selection of Darwinian evolution theory and the biological evolution process of genetic mechanism. It is a method to search for the optimal solution by mimicking the natural evolution process and trying to find the optimal solution to the problem base on the principle of survival of the fittest and gene exchange between individuals. The basic GA process is shown like *Figure 3.6*.

The GA starts with the population encoded by genes which

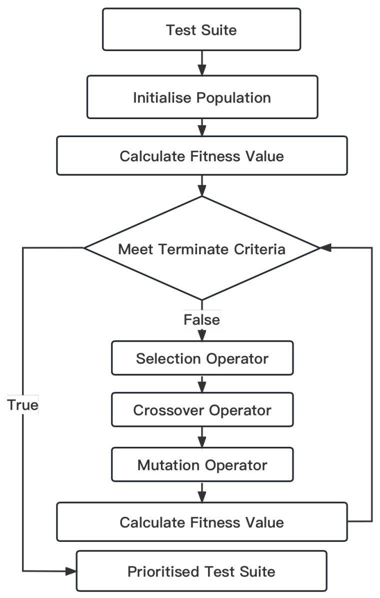


Figure 3.6: Process of Genetic Algorithm

represent the potential solution set to the problem. Through a series of evolutionary operators, the selection, crossover and mutation operators, to find the best result until meeting the terminate condition, typically reaching the maximum iterations or the solution has been found.

In the GA approach, each individual is associated with a fitness value that is used to evaluate the degree of fitness of the corresponding chromosome. The fitness value serves as a measure of the quality of the individual, where a higher fitness value indicates a better performance. Individuals are selected based on their fitness values to ensure that those with superior adaptive performance have a higher probability of reproducing and passing on their favourable characteristics to the next generation.

GA are optimisation algorithms that mimic the process of natural selection and evolution. GA relies on three primary operators, namely the selection operator, crossover operator, and mutation operator. The selection operator is utilised to determine which individuals from the parent population will be passed on to the next generation based on a specific strategy. This strategy may involve selecting the fittest individuals at random or using a combination of these methods.

The crossover operator is employed to determine the recombination or crossover of individuals, which involves combining the genetic material of parent individuals to produce new offspring. This process is based on the principle of genetic recombination, which is a natural process that occurs during sexual reproduction in organisms.

The mutation operator is utilised to introduce small random changes into the genetic material of individuals, thereby creating new genetic variations. This operator helps to prevent premature convergence of the algorithm by introducing diversity into the population.

- *Particle Swarm Optimisation (PSO)*

The PSO algorithm was first proposed by Eberhart and Kennedy in 1995 [150], and has been applied to a wide range of applications such as optimisation of engineering design, control systems, signal processing, and financial forecasting. PSO has been found to be particularly effective in solving problems that are nonlinear and have many local optima.

PSO is a meta-heuristic optimisation technique widely used to solve complex optimisation problems. It is inspired by the social behaviour of bird flocking and fish schooling, where the individuals in the group exhibit collective intelligence to achieve a common goal. In PSO, a population of particles move around in the search space, adjusting their positions and velocities based on their own experience and that of their neighbours to find the optimal solution. The PSO process shows like *Figure 3.7*.

The PSO algorithm begins by initialising a population of particles with random positions and velocities in the search

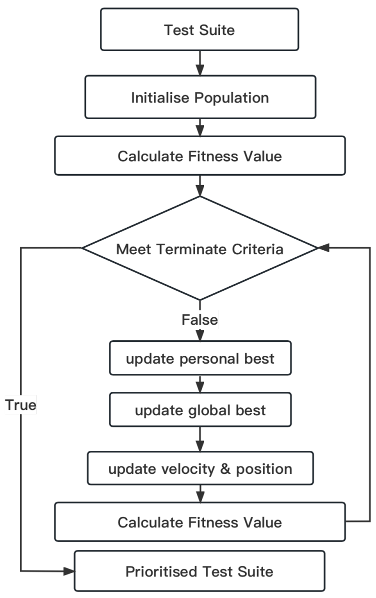


Figure 3.7: Process of Particle Swarm Optimisation

space. Different from GA, the individuals in PSO are called a particle.

Each particle is evaluated based on its fitness value, which measures how well it performs with respect to the objective function being optimised. At each iteration, the position and velocity of each particle are updated based on their own experience and that of the global best-performing particle. The particles then update their positions and velocities based on the following equations.

$$position_{new} = position_{current} + velocity_{current} \tag{3.3}$$

$$\begin{aligned} velocity_{new} = {} & w * velocity_{current} \\ & + c1 * rand() * (position_{pBest} - position_{current}) \\ & + c2 * rand() * (position_{gBest} - position_{current}) \end{aligned} \tag{3.4}$$

$$w_{new} = w_{current} * \alpha \tag{3.5}$$

In *Equation 3.4*, w, $c1$, and $c2$ are constant parameters that control the behaviour of particles, and rand() is a random number generator. The $position_{pBest}$ and $position_{gBest}$ represent the best position found by the particle itself and the entire population, respectively.

In detail, w is called inertia weight, and this factor encourages particles to maintain their current direction and speed. $c1$ is the cognitive learning factor, which encourages particles to move towards their personal best position. $c2$ is the social learning factor, which encourages particles to move towards the best position found by the swarm as a whole.

- *Firefly Algorithm*

The firefly algorithm is a more recent swarm-based optimisation algorithm proposed by Yang in 2009 [151], inspired by the flashing behaviour of fireflies. The algorithm mimics the way fireflies attract one another by emitting light, with brighter fireflies attracting others and weaker ones being attracted towards brighter ones. The process of firefly algorithm shows like *Figure 3.8*.

The basic idea of FA is to represent each potential solution as a firefly in the search space. The brightness of a firefly corresponds to the quality of the solution it represents, which is the original or modified APFD metric in the TCP problem.

The algorithm randomly generates the initial population, and then at each iteration, the algorithm updates the position of each firefly based on its brightness and attraction to other fireflies.

The movement of each firefly in FA is influenced by two

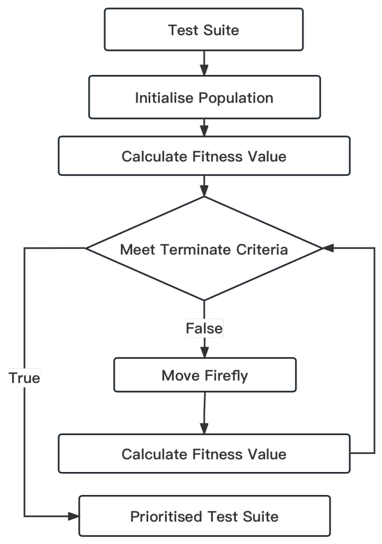


Figure 3.8: Process of Firefly Algorithm

factors, which are the brightness of the firefly itself and the brightness of other fireflies. Fireflies move towards brighter fireflies and are attracted to them with force inversely proportional to their distance apart. The movement of a firefly is described by the *Equation 3.6*.

$$x_i(t+1) = x_i(t) + \beta_0 e^{-\gamma r_{ij}^2}(x_j(t) - x_i(t)) + \alpha(rand - 0.5) \quad (3.6)$$

In this equation, $x_i(t)$ and $x_j(t)$ are the positions of firefly i and j at time t, r_{ij} is the Euclidean distance between the two fireflies, β_0 and γ are constant parameters that control the attractiveness of other fireflies, and α is a constant parameter that controls the randomisation factor. The term (rand - 0.5) is a random value between -0.5 and 0.5.

The algorithm continues to iterate through these steps until a stopping criterion is met, such as a maximum number of iterations or a satisfactory solution is found. One of the strengths of FA is its ability to explore the search space efficiently and converge to the optimal solution. It can also handle high-dimensional problems and problems with complex and nonlinear constraints.

- *Fish School Algorithm*

The fish school algorithm is a swarm intelligence optimisation algorithm proposed by Bastos in 2008 [152]. The algorithm is inspired by the collective behaviour of fish schools. This optimisation algorithm simulates the movement and interaction of a group of fish to search for the optimal solution in a given search space. The algorithm is based on two main components, which are collective behaviour and volitive behaviour. The process of fish school algorithm shows like *Figure 3.9*

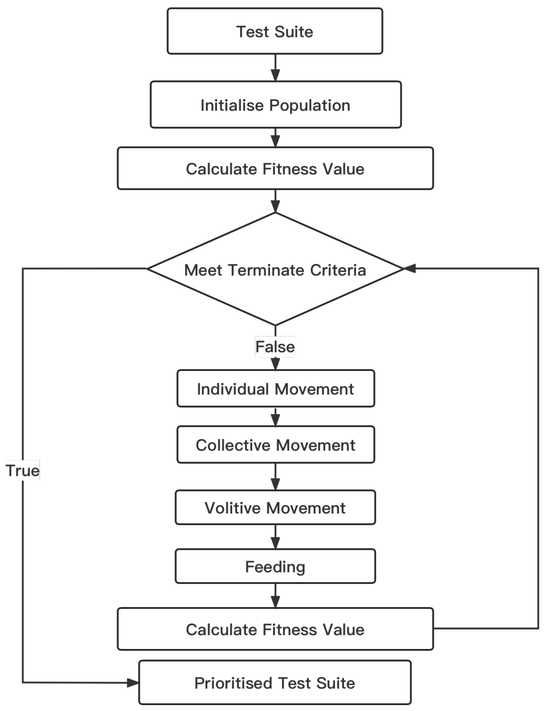


Figure 3.9: Process of Fish School Algorithm

In this algorithm, individuals are referred to as fish, and the main concept is for each fish to swim towards the best solution to gain weight through feeding. Each fish utilises three types of swims, including individual movement, collective movement, and volitive movement.

Each fish in the school conducts a local search (individual) to explore promising regions in the search space. And this local search can follow the *Equation 3.7*.

$$x_i(t+1) = x_i(t) + rand(-1,1) * step_{ind} \tag{3.7}$$

In this equation, $x_i(t+1)$ and $x_i(t)$ refers to the position after and before individual movement, and the rand(-1, 1) is used to determine the direction of movement. The $step_{ind}$ controls the displacement of the movement.

Collective behaviour refers to the tendency of fish to move in a group and adjust their positions and velocities based on the average position and velocity of the group. This behaviour ensures that the group stays together and moves in the same direction. In the fish school algorithm, collective movement is modelled by the *Equation 3.8* and *Equation 3.9.*

$$I = \frac{\sum_{i=1}^{N} \Delta x_i \Delta f_i}{\sum_{i=1}^{N} \Delta f_i} \tag{3.8}$$

$$x_i(t+1) = x_i(t) + I \tag{3.9}$$

In these equations, I represents the weighted average displacement of each fish, and the fish that gained a bigger improvement will attract other fish to move toward its position.

Volitive movement refers to the tendency of fish to make individual decisions based on their own experience and environmental cues. This movement enables fish to adapt to changes in the environment and find the best path to the optimal solution. The volitive movement is modelled by the

Equation 3.10, *Equation 3.11* and *Equation 3.12*. This movement is used to regulate the exploration/exploitation ability of the algorithm.

$$B(t) = \frac{\sum_{i=1}^{N} x_i(t)W_i(t)}{\sum_{i=1}^{N} W_i(t)} \tag{3.10}$$

$$x_i(t+1) = x_i(t) - step_{vol}\frac{x_i(t) - B(t)}{distance(x_i(t), B(t))} \tag{3.11}$$

$$x_i(t+1) = x_i(t) + step_{vol}\frac{x_i(t) - B(t)}{distance(x_i(t), B(t))} \tag{3.12}$$

The B in the equations is the barycentre of the school, which is calculated through the position and weight of each fish. If the overall weight of the school increases from the last to the current iteration, the fish would intend to move toward the barycentre. Otherwise, the fish will move away from the barycentre. And the distance function calculates the Euclidean distance between the fish and the barycentre. The $step_{vol}$ controls the distance of movement.

Beyond the movements, the fish school algorithm also has a feeding operator to update the weight of each fish, which follows the *Equation 3.13*.

$$W_i(t+1) = W_i(t) + \frac{\Delta f_i}{max(|\Delta f_i|)} \tag{3.13}$$

In this equation, the Δf_i is the fitness variation of the corresponding fish during the last and current iteration, and

$max(|\Delta f_i|)$ is the maximum absolute fitness variation among the whole school.

The fish school algorithm combines collective behaviour and volitive behaviour to explore the search space efficiently and converge on the global optima, which is a valuable optimisation technique that can provide high-quality solutions to complex problems.

- *Cuckoo Search Algorithm*

The cuckoo search algorithm is a nature-inspired optimisation technique developed by Yang in 2009 [153]. The algorithm is based on the behaviour of cuckoo birds, which lay their eggs in the nests of other bird species, leading to the survival of the fittest scenario where the cuckoo's egg competes with the host bird's eggs for survival. This strategy of the cuckoo bird has inspired the development of a meta-heuristic optimisation algorithm that aims to search the entire solution space to find the optimal solution to a given problem. The process of cuckoo search algorithm shows like *Figure 3.10*

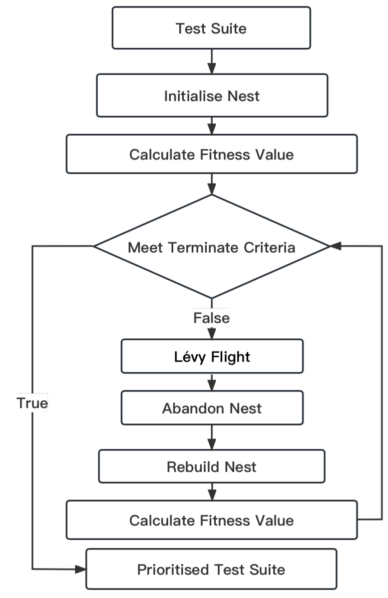


Figure 3.10: Process of Cuckoo Search Algorithm

The cuckoo search algorithm generates an initial population of nests randomly and then evaluates the fitness of each nest. The fitness function is a measure of how well a particular solution performs with respect to the problem at hand.

For each nest, generate a new solution, called a cuckoo, by randomly selecting a nest from the population and performing a random walk in the solution space. This step is inspired by the behaviour of cuckoo birds, where they lay their eggs in the nests of other birds. The new solution is generated by modifying the current solution in a random direction, which is determined by a random number generator. The random walk is called Lévy flight, which shows like *Equation 3.14*.

$$x_i^{(t+1)} = x_i^{(t)} + \alpha \oplus \textit{Lévy}(\lambda) \tag{3.14}$$

After Lévy flight, evaluate the fitness of the cuckoo solution. Replace the nest with the cuckoo solution if the fitness of the cuckoo is better than the fitness of the nest. This step aims to keep the best solutions in the population and replace the weaker ones with better solutions.

Abandon a fraction of the worst nests, and generate a new nest randomly. These steps are also inspired by the behaviour of cuckoo birds. If a host bird discovers a foreign egg in its nest, it may abandon the nest and build a new one elsewhere. In the cuckoo search algorithm, the fraction of abandoning a nest is proportional to its population. If a nest is abandoned, a new one is generated randomly. Repeat the iterations until the stop criteria are met.

3.2.2 Test Case Minimisation

The TCM problem can formally be defined as [3]:

Define: Test Case Minimisation Problem

Given: A test suite, T, a set of test requirements r_1,...,r_n, that must be satisfied to provide the desired 'adequate' testing of the program, and subsets of T, T_1 , . . . , T_n , one associated with each of the r_is such that any one of the test cases t_j belonging to T_i can be used to achieve requirement r_i.

Problem: Find a representative set, T', of test cases from T that satisfies all r_is.

When each test requirement in r_1,...,r_n is satisfied by one of the test cases, the testing criteria will also be satisfied. The newly formed test suite, T' is the representative set of test cases selected from T_is. Moreover, to maximise the effectiveness of the TCM process, the T' should be the minimal representative set of T_is.

The TCM technique aims to reduce the size of the test suite by eliminating redundant test cases in the test suite to improve the testing ability. TCM also refers to test case reduction, which means the elimination is permanent. However, the two concepts are essentially interchangeable.

A test suite consists of all test cases that meet all predefined test requirements. As the size and complexity of the software grow, the test suite becomes larger and larger. Rerunning all the test cases within the test suite seems unfeasible and increases the test budget. The TCM technique generates a representative set from the original test suite that satisfies all the requirements covered by the original test suite but contains fewer test cases. A test case is redundant if other test cases can satisfy the same requirements, then removing this kind of test case will not decrease the fault detection ability in most circumstances.

The heuristic-based approaches are the most widely used techniques to perform the TCM process. These approaches include GE, GRE heuristic, etc. [6]

GE heuristic is based on the greedy and essential strategy. An essential test case is such that the requirement r_i cannot be satisfied by any other test case. GE heuristic first selects all essential test cases into the new test suite. Then, by using greedy additional to choose test cases into the new test suite, for example, select the test case that satisfies the maximum unsatisfied requirements.

GRE heuristic [154] is the improved version of the GE heuristic. GRE heuristic first removes all obvious redundant test cases within the test suite, then performs a GE heuristic to generate the test suite.

There are also many other TCM techniques, for example.

Chen [155] proposed a divide-and-conquer approach to perform the TCM process. Divide-and-conquer decomposes the original problem into a set of sub-problems, finds the optimal solution to each sub-problem, and then, bottom-up combines these optimal solutions to solve the actual problem. They extracted the essential and redundant test cases from the original test suite. The essential set contains the essential test cases, while the redundant set contains the test cases, which satisfied requirements can be satisfied by other test cases. The essential subset should be included, and the redundant subset will be discarded to form the new test suite.

In [156], Tallam proposed a delayed greedy strategy. One potential weakness of the greedy strategy is that the early selected test cases may eventually be redundant by subsequently selected test cases. The authors overcame this weakness by constructing a concept lattice, a hierarchical cluster-

[6]GE - Greedy Essential; GRE - Greedy Redundant Essential

ing which recorded the relationship between the test cases and test requirements. The experiment result showed that the delayed greedy strategy could select fewer test cases compared to the greedy strategy.

In [157], Nachiyappan proposed a TCM based on a genetic algorithm. The proposed approach used a mathematical model for reduction and the initial population by test history in this model. The fitness value within the genetic algorithm is calculated by the block-based coverage and execution time for test cases. Test cases which violate the defined fitness constraints will be rejected. The results showed that the reduced test suite has the same coverage ability as the original test suite. In [158], Zhang proposed an improved quantum genetic algorithm approach that fasts the convergence speed by using individual fitness values to adjust the chromosomes dynamically.

In the empirical study [159], Wong examined whether reducing the size of the test suite will affect the fault detection ability through ten common Unix programs and randomly generated test suites. The results showed only a little or no reduction in fault detection effectiveness. In this study, two assessment metrics are used to validate the effectiveness of TCM techniques.

$$(1 - \frac{number\ of\ test\ cases\ in\ the\ reduced\ test\ suite}{number\ of\ test\ cases\ in\ the\ original\ test\ suite}) * 100\% \tag{3.15}$$

and

$$(1 - \frac{number\ of\ faults\ detected\ by\ the\ reduced\ test\ suite}{number\ of\ faults\ detected\ by\ the\ original\ test\ suite}) * 100\% \tag{3.16}$$

The *Equation 3.15* describes how many test cases are eliminated after the reduction. The *Equation 3.16* demonstrates the percentage of decreased effectiveness to fault detection ability after applying the test cases minimisation technique.

The decreased effectiveness of fault detection ability is not excessive. To reduce the testing budget, the TCM process can be applied to the SDLC. Moreover, if the testing cost is directly related to the number of test cases within the test suite, the minimisation technique is a necessary optimisation process.

Palomo [160] used an exact search-based technique to perform the minimisation process and maintain the mutation coverage meanwhile. The approach mutated the original specifications and compared their behaviours versus the original ones under the test suite. If the mutants cannot be killed, the proposed mutated will classify the test case as redundant.

A genetic algorithm approach is demonstrated in [161] by Bhatia. The proposed algorithm aims to provide optimal fitness value by combining the genetic algorithm and class partition. The modified GA minimise the number of test cases by finding the most error-prone test cases based on the priority of the path information.

Lin [162] proposed an approach to minimise test suite for composition service by modification impact analysis. They compared structural and variable changes and located the impacted nodes by dependency analysis. Through the influenced information, they excluded redundant test cases to perform the TCM process.

Hashim [163] proposed a TCM approach based on the firefly algorithm. The optimisation process is combined with the

UML state machine diagram, by analysing the path coverage for the state machine to calculate the fitness value for each test case. Then, through a firefly algorithm to minimise the test suite.

Deneke [164] also proposed a TCM approach by using an evolutionary algorithm, which is based on particle swarm optimisation. The proposed approach through requirements coverage information to guide the particle swarm optimisation process. They validate the effectiveness of their method by comparing various benchmark techniques.

Li [165] proposed a mutation-based TCM and TCP approach. Based on the fault detection information of each test case, the hierarchical clustering process is conducted. Then the test case with the highest priority in each cluster is selected according to the cut-level threshold. If there exists any edge test or edge mutation program, the corresponding one will be chosen. The edge mutation program is the faulty mutation version that can only be killed by one specific test case, and that test case is the edge test.

Bajaj [166] suggested an improved quantum-behaved particle swarm optimisation for TCO processes, which include the TCM process. The proposed approach has been validated against various evolutionary algorithms. They used fault coverage or statement coverage as the fitness function to guide the optimisation process.

Huang [167] improved the efficiency of the adaptive random testing technique for the TCM process by introducing the fixed-size candidate set. Compared with the original algorithm, which selects one candidate from the candidates and generates new candidates, the proposed algorithm selects the best candidates. According to the distance, some poor candidates are discarded, and new randomly generated candidates are added to improve efficiency. Compared with the original algorithm, the efficiency is improved, and the effectiveness is almost the same.

Gupta [168] proposed a TCM approach that aims to help fault detection and localisation efficiency. They combined the NSGA-II [7] algorithm with statement, branch coverage, and diversity-aware mutation adequacy criteria to perform the optimisation process.

Turner [169] also used the NSGA-II algorithm to conduct the TCM process, in which the aim is to optimise the code coverage and execution time.

The requirements that will be used in the TCM process can either be the real system defects or the artificially simulated faults. The real faults are always unavailable in the MBT process since the actual system may still not be implemented. Then, mutation testing is a powerful technique that helps to guide the TCM process.

In this work, five optimisation algorithms are implemented to conduct the TCM process. These algorithms are Non-dominated Sorting Genetic Algorithm II (NSGA-II), Particle Swarm Optimisation (PSO), Multi-objective Evolutionary Algorithm based on Decomposition (MOEA/D), Strength Pareto Evolutionary Algorithm II (SPEA2) and Cuckoo Search algorithm.

- *NSGA-II*

Non-dominated Sorting Genetic Algorithm II (NSGA-II) is an optimisation algorithm proposed by Deb in 2002 [170]. The NSGA-II is based on GA and specifically designed for multi-objective optimisation problems.

NSGA-II aims to obtain a collection of solutions that are Pareto-optimal, indicating that no other solution can en-

[7]Non-dominated Sorting Genetic Algorithm II

hance one objective without compromising at least one other objective. To accomplish this objective, NSGA-II adopts a fast non-dominated sorting method for identifying the top solutions and employs a crowding distance metric to retain diversity among the chosen solutions.

This algorithm is based on the GA, so the general process of NSGA-II is quite similar to the GA. Start with the randomly generated population, then through the evolutionary operators iteratively to solve the corresponding problem.

- *Particle Swarm Optimisation (PSO)*

The multi-objective PSO algorithm is a variant of the PSO algorithm used to optimise multiple objectives simultaneously, which is commonly used in complex optimisation problems when more than one objective need to be optimised at the same time.

The multi-objective PSO algorithm works by using a fitness function that considers all the objectives that need to be optimised. In a multi-objective PSO algorithm, instead of trying to find a single solution that optimises a single objective, the algorithm attempts to find a set of optimal solutions, which are the Pareto optimal set.

- *MOEA/D*

Multi-Objective Evolutionary Algorithm based on Decomposition (MOEA/D) is a popular multi-objective optimisation algorithm that was first proposed by Zhang in 2007 [171]. MOEA/D is a population-based meta-heuristic algorithm that optimises multiple objectives simultaneously.

The overall idea of MOEA/D optimisation is to decompose the original multi-objective problem into single-objective sub-problems. A local search algorithm then solves each subproblem, and the solutions obtained from each sub-problem are combined to form the final Pareto optimal set. The gen-

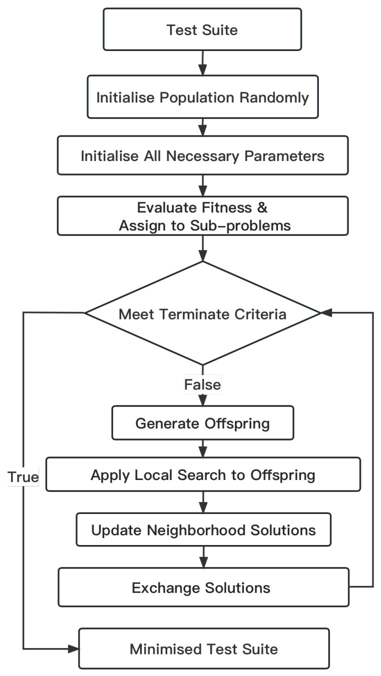

Test Suite
Initialise Population Randomly
Initialise All Necessary Parameters
Evaluate Fitness & Assign to Sub-problems
Meet Terminate Criteria
False
Generate Offspring
True
Apply Local Search to Offspring
Update Neighborhood Solutions
Exchange Solutions
Minimised Test Suite

Figure 3.11: Process of MOEA/D Algorithm

eral process of this algorithm is shown as *Figure 3.11.*

During the initialisation process, the MOEA/D algorithm randomly generates a set of solutions as the initial population. The original multi-objective problem will be decomposed into a set of sub-problem by selecting a set of weight vectors. Each weight vector defines a specific linear combination between the objectives. The sub-problem optimisation phase involves applying a local search algorithm to each sub-problem to find the optimal solution. In order to improve diversity and convergence, the solutions between the neighbourhoods of sub-problems will be exchanged. The algorithm terminates when a stopping criterion is met, and the final Pareto optimal set is returned as the output of the algorithm.

The key of MOEA/D is the decomposition process, which allows the multi-objective problem to be solved as a set of single-objective subproblems. And using a neighbourhood search mechanism further enhances the diversity and convergence of the algorithm, which results in a highly efficient and effective optimisation procedure.

- *SPEA2*

Strength Pareto Evolutionary Algorithm 2 (SPEA) is a multi-objective optimisation algorithm which has been proposed by Zitzler in 2001 [172]. This algorithm has already been widely used in various fields like finance and software engineering. The general process of SPEA2 is demonstrated as *Figure 3.12*.

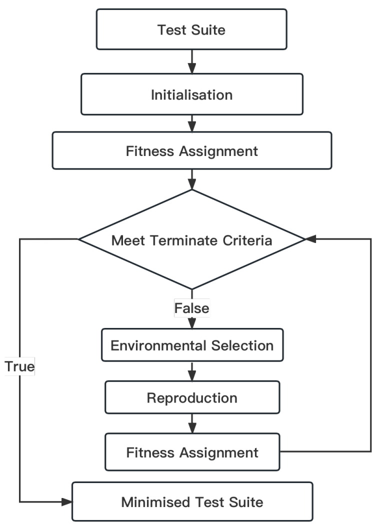


Figure 3.12: Process of SPEA2 Algorithm

In the initialisation process, the algorithm constructs the initial population by randomly generating candidate solutions. Each candidate individual contains a set of decision variables according to the objectives of the problem.

In the fitness evaluation process, the fitness of each individual is calculated based on its proximity to other individuals within the population by using a metric known as crowding distance. The fitness value is the sum of raw fitness value and crowding distance. The raw fitness value measures the domination information of the corresponding candidate individual. The crowding distance of the candidate individual is calculated through the distances between the candidate and the related neighbours within the search space. Introducing the crowding distance is one of the main differences between the SPEA and SPEA2 optimisation algorithms, which helps to improve the diversity within the population.

During the environmental selection process, a subset of the population will be passed to the next generation. The selection process is mainly based on the results of the fitness assignment process and aims to maintain a certain level of diversity to cover more parts of the search space. This selection process combines elitism and diversity preservation to ensure the exploration ability.

The reproduction will fulfil the gap in size between the selected subset of the population and the population. This step is similar to the GA process. Certain parent individuals, usually 2, will be selected. Then, through crossover operator to produce offspring individuals. In this step, the mutation operator will also be included to construct the new offspring, which helps search space more effectively. The reproduction process will introduce new individuals to the population and benefit the exploration ability across the search space.

- *Cuckoo Search*

To solve the TCM problem, we adapted the cuckoo search algorithm into a multi-objective scenario. The multi-objective cuckoo search algorithm is a variant of the cuckoo search algorithm that is designed to solve multi-objective optimisation problems. The overall process of the algorithm used in TCM optimisation is exactly the same as that used in the TCP process. The algorithm starts with the randomly generated initial population and then by using the Lévy flight to explore the search space.

3.2.3 Test Case Selection

The TCS problem can formally be defined as [3]:

Define: Test Case Selection Problem

Given: The program, P, the modified version of P, P' and a test suite, T.

Problem: Find a subset of T , T', with which to test P'.

When applied changes or updates to software, there is a need to perform regression testing to check the improvements do not bring any new defects into to system or violate the system requirements. Re-testing all previous test cases is time-consuming and seems unnecessary in most scenarios because the improvement will only affect part of the system usually. TCS aims to reduce the test suite size by choosing the test cases related to the changed part. The majority of selection techniques are modification-aware.

Both TCS and TCM processes reduce the test case amount within the test suite. The key difference between these two techniques is whether they focus on the changed part of system specifications. TCM is often based on one system version, but TCS is required to identify the improvements between the previous and current versions of system specifications.

By considering the following notations, the subset of fault-revealing test cases, T_{fr}, the subset of modification-revealing test cases, T_{mr}, the subset of modification-traversing test cases, T_{mt}, and the original test suite, T. The following relationship can be established.

$$T_{fr} \; = \; T_{mr} \; \subset \; T_{mt} \; \subset \; T$$

TCS techniques are designed to find the subset of test cases which belong to T_{mt}, due to direct finding the T_{mr} is hard to perform. The subset T_{mt} is the closest set to T_{mr} without executing the whole set of the test suite. In other words, by finding T_{mt}, it is possible to determine which test cases do *not* able to reveal any fault within the new version system P', then to reduce the test suite size.

There are various TCS techniques based on incomplete statistics. The techniques include integer programming, data-flow analysis, dynamic slicing, CFG graph-walking, textual difference comparison, SDG slicing, clustering method, design-based testing, etc. [8] Following are the partial discussions of these techniques.

In [173], Agrawal proposed a set of TCS techniques based on different program slicing methods. The execution slice of a program is the execution traces to the corresponding test case, which is the set of statements executed by the given test case. A dynamic slicing technique determines the statements by executing the given test cases. The slicing result contains the statements which will influence the output of the program. The difference between execution slicing and dynamic slicing is that execution slicing may contain statements that do *not* affect the output. Meantime, dynamic slicing is a specific type of execution slicing. If the slicing result contains the modified system specification, the corresponding test case will be included in the TCS result set.

The similarity-based TCS strategy is conducted by Nachiyappan [157]. The general idea of this strategy is that assume that the diversity of test cases tends to maximise the ability to detect faults because the dissimilar test cases will cover different parts of the system in theory. And this

[8] CFG - Control Flow Graph; SDG - System Dependence Graph

method by using similarity functions to calculate distances between test cases. And then, the size of the test suite is reduced by selecting a subset of test cases according to the gathered distances.

The textual difference comparison method is a TCS strategy that uses the difference between the current and previous program specifications to perform the selection process. In [174], Vokolos identified the modified parts of the system by applying the textual comparison tool, *diff*, to extract the differences between various versions of source code. The source code is converted into canonical forms to eliminate the impact of unnecessary differences. The test cases which covered the distinctions between source codes will be selected into the result set.

The clustering strategy is to group test cases according to their characteristics, like the work proposed by Chen [175]. The fundamental idea of this kind of approach is that minimise intra-group variance and maximise inter-group variance. The advantage of the clustering method is that decreasing the test suite size through the grouped data then to save the testing effort, and most of the clustering methods use similarity or distance to categorise the clusters. This method may not be necessary to involve information about the program changes.

The design-based or model-based testing method uses design-level specifications to perform the TCS process. In [176], Briand proposed a black-box UML-based approach. The author categorised the previous test suite into obsolete, re-testable and re-usable. This approach used behavioural models, and sequence diagrams, to select the test cases. The authors implemented an automated impact analysis tool for UML and empirically evaluated the effectiveness of the approach through case studies.

Al-Refai [177] submitted the PhD dissertation about model-based regression TCM. The proposed approach used reverse engineering to extract activity and class diagrams from the different software system versions and compare the models to find the information on model changes. Then, using the extracted information to perform the test cases selection process and support inheritance hierarchy changes of dependence relationship.

Alkhazi [178] published a work on the TCS for model transformation. This article performed the TCS process for model transformation, and the proposed method adopted the genetic algorithm to complete the selection process. Rules coverage and execution time are combined as the reference values of fitness value to the genetic algorithm. The proposed approach is compared with random testing, test all and other approaches. The proposed approach is not as good as the mono-objective one in terms of time reduction, but the proposed method has a stronger advantage in transition coverage. This work is not directly relevant to the TCO methods for the software system, but the underlying techniques are mutually connected.

Dorcis [179] customised a clustering algorithm for usage traces to perform the selection process. The clustered test case reduces the size and redundancy of the test suite to save the testing efforts. Based on the selection criteria, the representative test cases that are necessary to rerun are selected from the clusters.

Guizzo [180] experimented with the process of using the TCS process to accelerate the genetic improvement process. The results showed that when applied TCS process, the efficiency gains are up to 68%. This result demonstrated the

TCS process not only benefits the testing process, but also helps various activities within software engineering.

d'Aragona [181] combined with JUnit to propose a TCS method for the Java program. For each *@Test* within JUnit, a fake test case is invoked to construct the call graph of the corresponding test case. If the call graph traverses the modified parts of the program, that test case will be selected for the rerun. And the proposed method is integrated into IntelliJ Idea as a plug-in.

Arrieta [182] proposed a set of seeding strategies for TCS algorithms. The general idea is to improve the efficiency of the optimisation process through the initial population of the search-based algorithm. The proposed strategies are evaluated with four multi-objective search algorithms, two different application domains and several case studies in each of these application domains.

3.3 Mutation Testing

How can we generate test cases that reveal faults? How confident are we with our test suite? Mutation testing could be a possible solution to answer these questions by checking the effectiveness of the test suite through artificial defects. If the test suite cannot detect the designed faults, we reduce confidence with the existing test suite [183].

Mutation testing provides a range of methods, tools, and reliable results for the software testing process. There are two basic hypotheses within mutation tests designed to find valid test cases and real errors in the program. In a project, the number of potential bugs is enormous, and it is impossible to generate mutants to cover all the bugs. Therefore, traditional mutation testing aims to find a subset of these errors that approximate the bugs as closely as possible.

There are two primary hypotheses within mutation tests designed to find valid test cases and real errors in the program. The two hypotheses are: Competent Programmer Hypothesis (CPH) and Coupling Effect (CE). CPH means: assuming that programmers are capable and they try to develop programs better and achieve the right results, not faulty ones. It focuses on the behaviour and intentions of the programmers. CE is more concerned with the category of errors in the variation test. A simple error is often caused by a single variation, while a large and complex error is often caused by more variation. Complex variants are usually composed of many simple variants.

Software testing only can prove the existence of system faults and software failure. Still, testing can never confirm the absence of defects because it is nearly impossible to perform exhaustive testing under the time and resource constraints[184]. Also, the system faults are the aggregation of simple defects. We can validate the correctness of the SUT by mutation testing based on the assumption that we can change the specifications, also known as mutants, to simulate the real system defects.

In mutation testing, from a program p, a set of faulty programs p' called mutants, is generated by a simple single change to the original program p. And in the next step, a test suite T is used to validate the corresponding systems. Before the mutation analysis, the original system p needs to be checked by the test suite T. This step collects the expected behaviours of the original program p. Then, each mutated program p' will be executed against the test suite T. If p' returns a different result compared to the original program p, this means the mutant p' is *killed*. Otherwise, it is *survived.*

One of the most crucial usage of mutation testing is to measure the mutation score of test cases by *mutation score (MS)*, which is used to describe the percentage of the killed mutants [185]. The equation of MS is formally defined as *Equation 3.17.*

$$MS = \frac{K_m}{T_m - E_m} \tag{3.17}$$

In this equation, K_m refers to the number of killed mutants, T_m is the number of generated mutants and E_m is the number of equivalent mutants. The range of mutation scores is from 0 to 1, and a higher mutation score means higher confidence in the corresponding test suite.

Mutation testing is an effective method to assess the quality of the test suite. But it still has some defects. One of the problems is mutation testing needs a high computational cost of executing an enormous number of test cases against all mutants of the original system. Another drawback is mutation testing needs many human efforts to be involved in the mutation testing process, like the test oracle and equivalent mutant problem [186].

The test oracle problem refers to the checking process for the output of the original and mutated program when executing the test suite. And strictly speaking, this problem not only belongs to mutation testing alone. For all forms of testing, when executing the given test suite, the only remaining effort is to check the testing output. However, more mutants mean more accuracy checks are required for the mutation testing process, which will highly increase the testing efforts, even though there may have some tools to check the results automatically. This oracle cost is usually the most expensive part of the overall test activity. In addition, due to the indeterminacy of mutant equivalence, detection of equivalent mutants usually requires additional human effort [187].

In the MBT area, researchers always use mutation testing to validate the effectiveness of the proposed approach. Mutation testing usually appears in the validation process of the proposed approach. They use the proposed method to construct the test suite and inject mutants into the corresponding system or design specification. Then researchers use the mutants to execute the test suite, and the mutation score will reflect the effectiveness of the related test suite. Model-based mutation testing always adapts to low-level behavioural models, like state chart diagrams and activity diagrams. How to better adapt mutation testing to the high-level structural model still need further exploration. In Chapter 4, We will demonstrate the mutation testing process for specifications expressed in OCL.

Many publications and surveys applied mutation testing to various application scenarios, like [188] from Ghiduk and [189] from Ma. This research is not trying to perform an exhaustive survey on mutation testing while mainly focusing on performing TCO processes to the system expressed in OCL. So, some related works which are closely associated with this research work are discussed in the following.

OCL is already a highly active research area, [190] and [46] proposed to add *Map* type and *Regular Expressions* to OCL standard by Lano, also [45] tried to refactor collection-related types. In [191] and [192], the authors worked on the mutation operators for the UML diagram, and they mentioned a few mutation operators to OCL standard, although far from enough to cover all OCL standard library.

After Ahmed [193] conducted a survey for mutation operators to the object-oriented system, Papadakis [183] presented a detailed survey about mutation testing, which included the descriptions of the problem, method, tool and common practices within the field of mutation testing. This work highlighted that mutation testing within the scope of MDE has not been well-studied compared to the code-based approaches in recent years, but there is a growing interest in this direction. [185] proposed the guidelines for mutation operators and the corresponding classification.

The classification of mutation operators to design models, mainly focused on UML models, is proposed by Strug in [191]. Also, Hassine [194] designed a set of mutation operators for the abstract state machines and proposed their classification.

In [192] Granda proposed a set of mutation operators to the OCL standard and validated their effectiveness through case studies. In [195], Ascari used OCL specification and related mutation operators to validate the correctness of the object-oriented programs. However, these papers only mentioned a limited number of mutation operators that can be applied to OCL specifications, and the classification of these operators is not considered.

Chapter 4

Mutation Testing for OCL

4.1 Introduction

OCL is a type of declarative language that adds constraints to UML models or expresses the software systems by pre- and post- conditions. Since MDE became increasingly popular in recent years and OCL acts as a standard MDE language, there has been a rapid growth in the number of system specifications expressed in OCL [196]. Due to these facts, the research on OCL-based mutation testing is currently receiving more and more attention.

In mutation testing, from a program or specification p, a set of faulty versions p' called mutants are generated by making, for each p', a single simple change to the original program p. Mutation operators are the transition rules defining how to perform these changes and represent the mutants. Mutant versions of expressions are syntactically and type-correct, but should have a distinct semantics from their source.

There are plenty of studies of mutation testing established on various programming languages. However, these approaches are language-dependent, which means the proposed mutation operators are only suitable for the corresponding programming language [187]. Suppose we can propose the mutation operators above the platform-dependent level, like OCL, which is at the same level as the system model. In that case, the mutation testing process will be benefited from the platform-independent operators. Due to the abstraction level of the OCL specification, the mutation operators to OCL specification will allow the mutated specifications to be generated once, and the results could be used for all implementations of corresponding OCL specifications in different programming languages or platforms.

In our previous work [197], we evaluated the feasibility of OCL-based mutation testing through limited mutation operators on a real-world Android system expressed in OCL. Here, we extend our previous work to the full set of mutation operators based on the OCL standard library.

This work is mainly according to Clause 11 *"OCL Standard Library"* within the OCL 2.4 standard [11]. This clause describes the predefined types, their operations, and predefined expression templates in the OCL. The proposed mutation operators are presented in the same order as the elements of Clause 11, although the standard presentation may need further refinement [45].

Based on the standard library of OCL standard version 2.4, there are two main groups of types, primitive types and collection-related types and predefined iterator expressions. The primitive types contain five individual types, *Real*, *Integer*, *String*, *Boolean* and *UnlimitedNatural*. Meanwhile, the collection-related types include one supertype *Collection*, and four sub-types, *Set*, *OrderedSet*, *Bag* and *Sequence*. The four sub-types depend on whether duplicated elements are allowed and whether the elements are ordered. Also, some predefined iterator operations are included in the standard library. The proposed mutation operators are applied to these OCL specifications and follow the order they appear in the standard library.

Within the OCL standard Library, the special types, the primitive types, the collection-related types and predefined iterator expressions are demonstrated. For the proposed mutation operators within this work, all these sub-clauses have corresponding mutation operators except the special types (OclAny, OclVoid, OclInvalid and OclMessage). We omit the special types because the OCL standard mainly focuses on the remaining types and operations. The details of these operators are in the following sections.

OCL standard has various variable types with corresponding operations, leading to the full set of mutation operators containing many operators. There are more than one hundred operators within the OCL standard library, and the corresponding mutation operators contain hundreds of transition rules. These operators have different impacts and results for mutation testing [191]. Based on the specific testing purposes, the OCL participators may not need to apply all these operators. Only the subset of these operators needs to be introduced.

There is a need to classify these mutation operators into different groups, allowing users to select particular groups based on their preferences. Hence, to manage the mutation operators, we also proposed the classification of the OCL mutation operators in this work.

The classification design in this work is inspired by [198] and [199], incorporating their core methodologies to organise our proposed mutation operators for the OCL standard library. The primary strategy focuses on the different facets of OCL standards and expressions, including arithmetic, conditional, relational, logical, etc. The various groups of mutation operators will enable the mutation testing process and the generated mutants to be more manageable, and help to confirm which specific parts of the specifications need to be explored further. Moreover, users can choose which aspects they care about more, or perform partial mutation testing based on their priorities and time constraints.

In addressing mutation testing in the context of OCL, it is crucial to understand its role within a broader testing strategy and its specific benefits for software development. This technique, which involves generating mutants or modified versions of the OCL specifications to simulate potential faults, is aimed at evaluating the comprehensiveness of the corresponding test suite. This approach is particularly valuable in enhancing the fault detection ability of the test suite, thereby ensuring a higher quality of software development.

The application of this methodology is particularly advantageous in contexts such as regression testing, where the primary goal is to ensure that new code changes do not introduce errors into previously verified parts of the software. Meanwhile, mutation testing can provide essential information to guide the TCO processes under the MBT context. For developers, mutation testing offers a systematic way to evaluate and enhance the quality of their test suites. This is crucial for maintaining high software quality across various scenarios.

Moreover, the broader significance of employing mutation testing lies in their contribution to the reliability and maintainability of software systems. Developers can ensure the software remains resilient against evolving challenges through iterative refinements prompted by mutation testing feedback, thereby enhancing its long-term stability.

In essence, mutation testing within the OCL context enriches the software development life cycle by providing a proactive framework for identifying and addressing potential model-based issues. This method not only aids in preemptively securing the software against potential faults but also supports a culture of continuous quality assurance and improvement, which is crucial for developing reliable and error-resistant software applications, particularly under the context of model-based testing.

The structure of this chapter is as follows. After the introduction, Sections 2, 3 and 4 demonstrate the full set of mutation operators for the OCL standard library, which include primitive types, collection-related types and the remaining operators. The classification of these operators is proposed in Section 5. Because it is hard to find the OCL specifications that cover all proposed operators, Section 6 partially validates the proposed mutation operators. Also, some discussions that need to be considered for this chapter are included in this section. Finally, Section 7 concludes this chapter with an outline of future works related to this chapter.

4.2 Primitive Types

This section presents the mutation operators for primitive types within the OCL standard library, which include *Real*, *Integer*, *String*, *Boolean* and *UnlimitedNatural*.

4.2.1 Real

This part contains the mutation operators for *Real*.

- **+ (r: Real): Real**

  MO_1 : -　　$MO_2 : *$　　$MO_3 : /$
- **- (r: Real): Real**

  $MO_1 : +$　　$MO_2 : *$　　$MO_3 : /$
- **$*$ (r: Real): Real**

MO_1 : + MO_2 : - MO_3 : /

- **-: Real** % negation of the original value (Unary)
  MO_1 : $\varnothing$ % eliminate the original operator
- **/ (r: Real): Real**
  MO_1 : + MO_2 : - MO_3 : $*$
- **abs(): Real**
  MO_1 : - % negation (regardless original value)
- **floor(): Integer**
  MO_1 : floor() + 1 MO_2 : floor() - 1
- **round(): Integer**
  MO_1 : floor()
- **max(r: Real): Real**
  MO_1 : min(r)
- **min(r: Real): Real**
  MO_1 : max(r)
- **< (r: Real): Boolean**
  MO_1 : >= MO_2 : > MO_3 : <=
- **> (r: Real): Boolean**
  MO_1 : <= MO_2 : < MO_3 : >=
- **<= (r: Real): Boolean**
  MO_1 : > MO_2 : >= MO_3 : <
- **>= (r: Real): Boolean**
  MO_1 : < MO_2 : <= MO_3 : >
- **toString(): String**
  NULL % no corresponding mutation operator

4.2.2 Integer

This part contains the mutation operators for *Integer*.

- **-: Integer** % negation of the original value (Unary)
  MO_1 : $\varnothing$ % eliminate the original operator
- **+ (i: Integer): Integer**

MO_1 : - MO_2 : $*$ MO_3 : /

- **- (i: Integer): Integer**
  MO_1 : + MO_2 : $*$ MO_3 : /
- **$*$ (i: Integer): Integer**
  MO_1 : + MO_2 : - MO_3 : /
- **/ (i: Integer): Real**
  MO_1 : + MO_2 : - MO_3 : $*$
- **abs(): Integer**
  MO_1 : - % negation (regardless original value)
- **div(i: Integer): Integer**
  MO_1 : mod(i)
- **mod(i: Integer): Integer**
  MO_1 : div(i)
- **max(i: Integer): Integer**
  MO_1 : min(i)
- **min(i: Integer): Integer**
  MO_1 : max(i)
- **toString(): String**
  NULL % no corresponding mutation operator

4.2.3 String

This part contains the mutation operators for *String*.

- **+ (s: String): String**
  MO_1 : $\varnothing$ % eliminate the original operator
- **size(): Integer**
  MO_1 : size() - 1 MO_2 : size() + 1
- **concat(s: String): String**
  MO_1 : $\varnothing$ % eliminate the original operator
- **substring(lower: Integer, upper: Integer): String**
  % in short lower => l, upper => u
  MO_1 : substring(l - 1, u) MO_2 : substring(l + 1, u)

MO_3 : substring(l, u - 1) MO_4 : substring(l, u + 1)

- **toInteger(): Integer**
  NULL   % no corresponding mutation operator
- **toReal(): Real**
  NULL   % no corresponding mutation operator
- **toUpperCase(): String**
  MO_1 : toLowerCase()
- **toLowerCase(): String**
  MO_1 : toUpperCase()
- **indexOf(s: String): Integer**
  MO_1 : indexOf(s) - 1 MO_2 : indexOf(s) + 1
- **equalsIgnoreCase(s: String): Boolean**
  MO_1 : =   % equal
- **at(i: Integer): String**
  MO_1 : at(i - 1) MO_2 : at(i + 1)
- **characters(): Sequence(String)**
  NULL   % no corresponding mutation operator
- **toBoolean(): Boolean**
  NULL   % no corresponding mutation operator
- **$<$ (u: String): Boolean**
  MO_1 : $>=$ MO_2 : $>$ MO_3 : $<=$
- **$>$ u: String): Boolean**
  MO_1 : $<=$ MO_2 : $<$ MO_3 : $>=$
- **$<=$ (u: String): Boolean**
  MO_1 : $>$ MO_2 : $>=$ MO_3 : $<$
- **$>=$ (u: String): Boolean**
  MO_1 : $<$ MO_2 : $<=$ MO_3 : $>$

4.2.4 Boolean

This part contains the mutation operators for *Boolean*.

- **or (b: Boolean): Boolean**   % e.g.   *a or b*

MO_1 : *not a and not b* MO_2 : *a xor b*

- **xor (b: Boolean): Boolean** % e.g. *a xor b*
  MO_1 : *a or b*
- **and (b: Boolean): Boolean** % e.g. *a and b*
  MO_1 : *not a or not b*
- **not: Boolean** (Unary)
  MO_1 : $\varnothing$ % eliminate the original operator
- **implies (b: Boolean): Boolean** % e.g. *a implies b*
  MO_1 : *a and not b*
- **toString(): String**
  NULL % no corresponding mutation operator

4.2.5 UnlimitedNatural

This part contains the mutation operators for *UnlimitedNatural*.

- **+ (u: UnlimitedNatural): UnlimitedNatural**
  $MO_1 : *$ $MO_2 : /$
- **∗ (u: UnlimitedNatural): UnlimitedNatural**
  $MO_1 : +$ $MO_2 : /$
- **/ (u: UnlimitedNatural): Real**
  $MO_1 : +$ $MO_2 : *$
- **div(u: UnlimitedNatural): UnlimitedNatural**
  MO_1 : mod(u)
- **mod(u: UnlimitedNatural): UnlimitedNatural**
  MO_1 : div(u)
- **max(u: UnlimitedNatural): UnlimitedNatural**
  MO_1 : min(u)
- **min(u: UnlimitedNatural): UnlimitedNatural**
  MO_1 : max(u)
- **< (u: UnlimitedNatural): Boolean**
  $MO_1 : >=$ $MO_2 : >$ $MO_3 : <=$
- **> u: UnlimitedNatural): Boolean**

$MO_1 : <=$ $MO_2 : <$ $MO_3 : >=$

- **<= (u: UnlimitedNatural): Boolean**
  $MO_1 : >$ $MO_2 : >=$ $MO_3 : <$
- **>= (u: UnlimitedNatural): Boolean**
  $MO_1 : <$ $MO_2 : <=$ $MO_3 : >$
- **toInteger(): Integer**
  NULL % no corresponding mutation operator
- **toString(): String**
  NULL % no corresponding mutation operator

4.3 Collection-Related Types

This section presents the mutation operators for collection-related types within the OCL standard library, which include one supertype *Collection*, and four sub-types, *Set*, *Ordered-Set*, *Bag* and *Sequence*.

4.3.1 Collection

This part contains the mutation operators for the supertype *Collection*.

- **= (c: Collection(T)): Boolean**
  $MO_1 : <>$
- **<> (c: Collection(T)): Boolean**
  $MO_1 : =$
- **size(): Integer**
  MO_1 : size() - 1 MO_2 : size() + 1
- **includes(object: T): Boolean**
  MO_1 : excludes(object)
- **excludes(object: T): Boolean**
  MO_1 : includes(object)
- **count(object: T): Integer**
  MO_1 : count(object) - 1 MO_2 : count(object) + 1

- **includesAll(c2: Collection(T)): Boolean**
  $MO_1 : c2 \to exists(e \mid self \to excludes(e))$
  (Equivalent to not excludesAll(...))
- **excludesAll(c2: Collection(T)): Boolean**
  $MO_1 : c2 \to exists(e \mid self \to includes(e))$
  (Equivalent to not includesAll(...))
- **isEmpty(): Boolean**
  MO_1 : notEmpty()
- **notEmpty(): Boolean**
  MO_1 : isEmpty()
- **max(): T**
  MO_1 : min()
- **min(): T**
  MO_1 : max()
- **sum(): T**
  $NULL$ % no corresponding mutation operator
- **product(c2: Collection(T2)): Set(Tuple(first: T, second: T2))**
  $NULL$ % no corresponding mutation operator
- **selectByKind(type: Classifier): Collection(T)**
  MO_1 : selectByType(type)
- **selectByType(type: Classifier): Collection(T)**
  MO_1 : selectByKind(type)
- **asSet(): Set(T)**
  $NULL$ % no corresponding mutation operator
- **asOrderedSet(): OrderedSet(T)**
  $NULL$ % no corresponding mutation operator
- **asSequence(): Sequence(T)**
  $NULL$ % no corresponding mutation operator
- **asBag(): Bag(T)**
  $NULL$ % no corresponding mutation operator
- **flatten(): Collection(T2)**

NULL % no corresponding mutation operator

Since operations *selectByKind*, *selectByType*, *asSet*, *asOrderedSet*, *asSequence* and *asBag* are included in all four subtypes and have the same mutation operators, they are omitted in the following subsections. Also, the operators = and *count* are omitted since only *OrderedSet* not implemented these operations.

The following parts are the mutation operators for *Set*, *OrderedSet*, *Bag* and *Sequence*.

4.3.2 Set

- **union(s: Set(T)): Set(T)**
  MO_1 : intersection(s)
- **union(bag: Bag(T)): Bag(T)**
  MO_1 : intersection(bag)
- **intersection(s: Set(T)): Set(T)**
  MO_1 : union(s)
- **intersection(bag: Bag(T)): Set(T)**
  MO_1 : union(bag)
- **- (s: Set(T)): Set(T)**
  MO_1 : union(s) MO_2 : intersection(s) MO_3 : symmetricDifference
- **including(object: T): Set(T)**
  MO_1 : excluding(object)
- **excluding(object: T): Set(T)**
  MO_1 : including(object)
- **symmetricDifference(s: Set(T)): Set(T)**
  MO_1 : union(s) MO_2 : intersection(s) MO_3 : -
- **flatten(): Set(T2)**
  NULL % no corresponding mutation operator

4.3.3 OrderedSet

- **append(object: T): OrderedSet(T)**
  MO_1 : prepend(object)
- **prepend(object: T): OrderedSet(T)**
  MO_1 : append(object)
- **insertAt(index: Integer, object: T): OrderedSet(T)**
  MO_1 : insertAt(index - 1, object) MO_2 : insertAt(index + 1, object)
- **subOrderedSet(lower: Integer, upper: Integer): OrderedSet(T)**
  MO_1 : subOrderedSet(lower - 1, upper) MO_2 : subOrderedSet(lower + 1, upper)
  MO_3 : subOrderedSet(lower, upper - 1) MO_4 : subOrderedSet(lower, upper + 1)
- **at(i: Integer): T**
  MO_1 : at(i - 1) MO_2 : at(i + 1)
- **indexOf(obj: T): Integer**
  MO_1 : indexOf(obj) - 1 MO_2 : indexOf(obj) + 1
- **first(): T**
  MO_1 : last()
- **last(): T**
  MO_1 : first()
- **reverse(): OrderedSet(T)**
  NULL % no corresponding mutation operator
- **sum(): T**
  NULL % no corresponding mutation operator

4.3.4 Bag

- **union(bag: Bag(T)): Bag(T)**
  MO_1 : intersection(bag)

- **union(set: Set(T)): Bag(T)**
  MO_1 : intersection(set)
- **intersection(bag: Bag(T)): Bag(T)**
  MO_1 : union(bag)
- **intersection(set: Set(T)): Set(T)**
  MO_1 : union(set)
- **including(object: T): Bag(T)**
  MO_1 : excluding(object)
- **excluding(object: T): Bag(T)**
  MO_1 : including(object)
- **flatten(): Bag(T2)**
  $NULL$ % no corresponding mutation operator

4.3.5 Sequence

- **union (s: Sequence(T)): Sequence(T)**
  $NULL$ % no corresponding mutation operator
- **flatten(): Sequence(T2)**
  $NULL$ % no corresponding mutation operator
- **append(object: T): Sequence(T)**
  MO_1 : preappend(object)
- **prepend(object: T): Sequence(T)**
  MO_1 : append(object)
- **insertAt(index: Integer, object: T): Sequence(T)**
  MO_1 : insertAt(index - 1, object) MO_2 : insertAt(index + 1, object)
- **subSequence(lower: Integer, upper Integer): Sequence(T)**
  % in short lower => l, upper => u
  MO_1 : subSequence(l - 1, u) MO_2 : subSequence(l + 1, u)
  MO_3 : subSequence(l, u - 1) MO_4 : subSequence(l, u + 1)

- **at(i: Integer): T**
  MO_1 : at(i - 1) MO_2 : at(i + 1)
- **indexOf(obj: T): Integer**
  MO_1 : indexOf(obj) - 1 MO_2 : indexOf(obj) + 1
- **first(): T**
  MO_1 : last()
- **last(): T**
  MO_1 : first()
- **including(object : T): Sequence(T)**
  MO_1 : excluding(object)
- **excluding(object : T): Sequence(T)**
  MO_1 : including(object)
- **reverse(): Sequence(T)**
  NULL   % no corresponding mutation operator
- **sum(): T**
  NULL   % no corresponding mutation operator

4.4 Other Operators

This section contains the remaining mutation operators and mainly focuses on *predefined iterator expressions.*

4.4.1 Predefined Iterator Expressions

% expr is the expression within the *body*. Similarly for the operator versions with local variables $\rightarrow any(v \mid expr)$, etc.

- $\rightarrow$ **any**(**expr**)
  $MO_1 : \rightarrow any(not\ expr)$
- $\rightarrow$ **closure**(**expr**)
  NULL   % no corresponding mutation operator
- $\rightarrow$ **collect**(**expr**)
  $MO_i : \rightarrow collect(M_i)$ % each mutant M_i of expr
- $\rightarrow$ **collectNested**(**expr**)

NULL % no corresponding mutation operator

- → **exists**(**expr**)
  $MO_1 : \rightarrow forAll(not\ expr)$
- → **forAll**(**expr**)
  $MO_1 : \rightarrow exists(not\ expr)$
  (Equivalent to *not (iterator → forAll(expr))*)
- → **isUnique**(**expr**)
  NULL % no corresponding mutation operator
- → **one**(**expr**)
  $MO_1 : \rightarrow select(expr)\ \rightarrow\ size()\ <>\ 1$
- → **reject**(**expr**)
  $MO_1 : \rightarrow select(expr)$
- → **select**(**expr**)
  $MO_1 : \rightarrow reject(expr)$
- → **sortedBy**(**expr**)
  MO_1 : randomShuffle() % shuffle the original order

4.4.2 Structural Operator

This subsection proposes the mutation operators for structural modifier *if*.

- **if(expression)**
  MO_1 : if(true) MO_2 : if(not expression)
- **if** con **then** exp1 **else** exp2 **endif**
  MO_1 : ***if*** *con* ***then*** *exp2* ***else*** *exp1* ***endif***

4.5 Classification of Mutation Operators

4.5.1 Supported Groups

This subsection proposes the supported groups that can be supported by OCL standard library, and we plan to implement them within the AgileUML tool.

- **Bound of Relational Operator (BRO)**

Original Operator	Mutated Operator
$<$	$<=$
$<=$	$<$
$>$	$>=$
$>=$	$>$

Explanation: This group contains the mutation operators that replace the relational operators $<$, $<=$, $>$, $>=$, and only considers the boundary condition. For each operator, the mutated version only considers the corresponding transition rule.

- **Numerical Value Negation (NVN)**

Original Value	Mutated Value
- i	i
i	- i

Explanation: The negation to the numerical value for *Real* and *Integer* types, this group does not apply to the *UnlimitedNatural* type due to it only having the positive value.

- **Simple Arithmetic Operator (SAO)**

Original Operator	Mutated Operator
$+$	-
-	$+$
$*$	$/$
$/$	$*$

Explanation: The SAO simply replaces the binary operator for numerical value types, and the transition rules are shown in the above table. For each original operator, only one corresponding mutated operator will be applied.

- **Negation of Relational Operator (NRO)**

Original Operator	Mutated Operator
=	<>
<>	=
<	>=
<=	>
>	<=
>=	<

Explanation: The NRO will negate the relational operators in condition expressions, which is the opposite operator to the original one, but the contained mutation operators are typically easy to detect during mutation testing.

- **Return Empty Mutation (REM)**

Return Type	Return Result
Real	result = 0.0
Integer	result = 0
String	result = $''$
Boolean	result = false
UnlimitedNatural	result = 0

Explanation: This group relates to the return value. For each primitive type, the corresponding return value is designed, mainly return 0 to the numerical value types, false to the *Boolean* type, and empty string to *String* type. This group will only consider the return values, while other places will not be considered.

- **Return False Mutation (RFM)**

Original Expression	Mutated Expression
result = *value*	result = false

Explanation: This mutation operator will be performed when the return type (result) is the *Boolean* type. Directly replace the original expression with *false.*

- **Return True Mutation (RTM)**

Original Expression	Mutated Expression
result = *value*	result = true

Explanation: This mutation operator will be performed when the return type (result) is the *Boolean* type. Directly replace the original expression with *true.*

- **Return Null Mutation (RNM)**

Return Type	Mutated Expression
Non-Primitive Types	result = null

Explanation: This mutation operator will be performed when the return type (result) is the *Collection-Related* type (non-primitive types). Directly replace the original expression with *null.*

- **Remove Conditional Expression (RCE)**

Original Expression	Mutated Expression
if (*expression*)	if (true)

Explanation: This mutation operator directly changes the conditional expression to *true* and guarantees the guarded expression will always be executed, which is careless about the evaluation result of the original expression.

- **Random Arithmetic Operator (RAO)**

Original Expression	Mutated Expression
a *op* b	a + b
	a - b
	a ∗ b
	a / b

Explanation: The RAO will also replace the binary operator for numerical value types. Compared to *Simple Arithmetic Operator (SAO)*, the RAO randomly chooses one of the operators demonstrated in the table, or more than one mutation will be generated. In this category, operators are specifically designed to prevent self-replacement, thereby reducing the occurrence of equivalent mutants that arise when mutations result in no actual change from the original.

- **Constant Replacement Mutation (CRM)**

Original Value	Mutated Value
i	1
	0
	-1
	- i
	i + 1
	i - 1

Explanation: The CRM will replace the original numerical value *i* with following values, *1*, *0*, *-1*, - *i* (negation), *i + 1* and *i - 1*. *UnlimitedNatural* type does not have negative values, so some mutation operators will not be applied. Users can generate one mutant that randomly applies one of the mutation operators or create various mutants by involving more operators.

- **Logical Operator Replacement (LOR)**

Original Expression	Mutated Expression
a *op* b	a or b
	a xor b
	a and b
	a implies b

Explanation: The LOR will replace the binary relational operator. Like *Random Arithmetic Operator (RAO)*, the original operator will be replaced by a random logical operator mentioned above, or more than one mutation will be generated. In this category, like RAO, operators are specifically designed to prevent self-replacement.

- **Iterator Expression Mutation (IEM)**

Original Expression	Mutated Expression
$\rightarrow exists(expr)$	$\rightarrow forAll(not\ expr)$
$\rightarrow forAll(expr)$	$\rightarrow exists(not\ expr)$
$\rightarrow one(expr)$	$\rightarrow select(expr)\ \rightarrow\ size\ <>\ 1$
$\rightarrow any(expr)$	$\rightarrow any(not\ expr)$
$\rightarrow reject(expr)$	$\rightarrow select(expr)$
$\rightarrow select(expr)$	$\rightarrow reject(expr)$
$\rightarrow sortedBy(expr)$	randomShuffle()

Explanation: The IEM will replace the operations to the *pre-defined iterator expressions*. And the *expr* is the expression within the *body* of the original operation.

Similarly for the operator versions with local variables $\rightarrow any(v \mid expr)$, etc.

- **Collection Related Operator (CRO)**

Original Expression	Mutated Expression
isEmpty()	notEmpty()
notEmpty()	isEmpty()
union	intersection
intersection	union
symmetricDifference	union
	intersection
includes(obj)	excludes(obj)
excludes(obj)	includes(obj)
including(obj)	excluding(obj)
excluding(obj)	including(obj)
includesAll	$c2 \rightarrow exists(e \mid self \rightarrow excludes(e))$
excludesAll	$c2 \rightarrow exists(e \mid self \rightarrow includes(e))$
selectByKind	selectByType
selectByType	selectByKind

Explanation: The CRO will replace the original expression to the *Collection-Related* types with the mutated expression presented in the above table.

- **IF Structure Mutation (ISM)**

Original Structure	Mutated Structure
if con	**if** con
then expr1	**then** expr2
else expr2	**else** expr1
endif	**endif**

Explanation: This group corresponds to the conditional structure *if*. The contained mutation operator exchanges the if-else expressions.

- **Index Based Operator (IBO)**

Original Expression	Mutated Expression
size()	size() + 1
	size() - 1
substring(lower, upper)	substring(lower + 1, upper)
	substring(lower - 1, upper)
	substring(lower, upper + 1)
	substring(lower, upper - 1)
indexOf(s)	indexOf(s) + 1
	indexOf(s) - 1
at(i)	at(i + 1)
	at(i - 1)
insertAt(index, obj)	insertAt(index + 1, obj)
	insertAt(index - 1, obj)
subSequence(l, u)	subSequence(l + 1, u)
	subSequence(l - 1, u)
	subSequence(l, u + 1)
	subSequence(l, u - 1)
subOrderedSet(l, u)	subOrderedSet(l + 1, u)
	subOrderedSet(l - 1, u)
	subOrderedSet(l, u + 1)
	subOrderedSet(l, u - 1)

Explanation: Unlike the most common programming languages which are 0-based indexing, OCL is 1-based indexing. These mutation operators simulated the differences between the 0-based and 1-based indexing. The IBO will modify the original expressions by + 1 and - 1. This is related to the common misunderstanding when using OCL instead of other programming languages.

- **Reverse Original Operator (ROO)**

Original Expression	Mutated Expression
not	∅
min()	max()
max()	min()
append(obj)	prepend(obj)
prepend(obj)	append(obj)
first()	last()
last()	first()
toUpperCase()	toLowerCase()
toLowerCase()	toUpperCase()
abs()	- (negation)

  Explanation: The original operator is replaced by a syntactically similar operator which however has a different semantics.

4.5.2 Possible Groups

This subsection proposes the groups that possibly can be supported by the library, and the feasibility needs further evaluation.

- **Remove Member Variable (RMV)**

  Explanation: Change the value of the member variable (defined in the method) to the initial value according to the variable type, *Real*(0.0), *Integer*(0), *String*(''), *Boolean*(false), *UnlimitedNatural*(0). Different to *Return Empty Mutation (REM)*, this group does not change the return value and only focuses on the value of the member variable. The return value remains in the original format, and the exact return value will be determined during the calculations within the post-conditions.

- **Delete Method Call (DMC)**

  Explanation: The DMC will replace the method call with the initial value according to the return type of the original method. *Real*(0.0), *Integer*(0), *String*(”), *Boolean*(false), *UnlimitedNatural*(0), *Collection-Related Types*(null).

- **Replace Method Call (RMC)**

  Explanation: The RMC will replace the method call with another possible one with the same return type and parameter list. Instead of *Delete Method Call (DMC)* using the actual return value, this group uses the method call with the same return type.

- **Arithmetic Operator Deletion (AOD)**

Original Expression	Mutated Expression
a *op* b	a
	b

  Explanation: The AOD will replace the original expression of a binary operator with one of its members, and the example is shown in the table.

- **Ungrouped Mutation Operators**

OCL Operation	Mutated Specification
floor()	floor() + 1
	floor() - 1
round()	floor()
div()	mod()
mod()	div()
concat(s: String)	∅
equalsIgnoreCase(s: String)	=

Explanation: This group contains the mutation operators that are defined according to the OCL standard library. However, according to our classification strategies, these mutation operators have not been classified into any group, as mentioned earlier.

4.6 Evaluation & Discussions

In this section, we evaluated the proposed mutation operators using three real-world OCL specifications from our research group, along with additional manually created OCL specifications. The real-world specifications include Bond, Interest Rate, and String Processing, which facilitate the computation of bond-related data, interest rate calculations, and string comparisons, respectively. These specifications vary in complexity, with their lengths ranging from 10 to 93 lines, and are considered to be small to medium in scale. More detailed descriptions of these OCL specifications will be introduced in Chapter 7 during the evaluation process. Due to the challenge of locating existing specifications that include all proposed mutation operators, we created specific benchmarks manually to extend the scope of our assessment. However, we still cannot perform a full-scale evaluation of these operators.

To these three specifications, the system details are expressed in OCL, which means OCL specifications are used to represent the expected behaviours of these systems through pre- and post- conditions. *Figure 4.1* shows the example specification, which contains one function *bisection* and the corresponding pre- and post- conditions.

In order to evaluate the proposed mutation operators, AgileUML treats these OCL specifications as input files. The

```
operation bisection(r : double , rl : double , ru : double ) : double
pre: r > -1 & rl > -1 & ru > -1 & ru <= 1 & rl <= 1 & r <= 1
post: v = value(r) & result = (if ru - rl < 0.001 then r else (if v > price
then self.bisection(( ru + r ) / 2,r,ru) else self.bisection(( r + rl ) /
2,rl,r) endif) endif);
```

Figure 4.1: OCL Example - bisection

corresponding test cases, mutated OCL specifications and corresponding executable Java files can be generated. We first examine the original specification by the generated test suite and Java programs to collect the corresponding outputs. Then, the same test suite will be executed against the mutated specifications to determine whether the corresponding mutants are killed. The results show in *Table 4.1*.

Table 4.1: Evaluation of Mutation Operators

Type	Generated Mutants	Mutation Score
Primitive Type	47	91.48%
Collection-Related Type	27	85.19%
Predefined Iterator Expressions	12	58.33%
Structural Operator	5	100%

In *Table 4.1*, how many mutants are generated and the corresponding mutation scores to the primitive type, collection-related type, predefined iterator expressions and structural operator are recorded.

From the results, the mutants for primitive type and *if* structure are most likely to be killed during the testing process. However, only around half (7) mutants for predefined iteration expressions have been killed. Some of them are quite tricky but possible scenarios. For example, when changing the operation *first()* to *last()*, in some cases, there may return the same element.

Some mutants have not been killed during the evaluation process, and there are two main reasons for this. First, some mutants are equivalent mutants, which cannot be killed during the mutation testing process. Second, regarding the test suite itself, the generated test suite failed to detect some mutants. How to generate a high-quality test suite still need to be explored further.

There is a threat to validity in this evaluation process, which is the generalisation of our selected specifications. As this threat always happens in most of the evaluation process, the set of chosen specifications in this work may not represent the whole population, and the distributions of OCL operations vary across different specifications. In order to reduce this impact, we choose three different OCL specifications with different scales.

In this chapter, the mutation operators for the standard OCL library and the corresponding classification of these operators have been proposed. Unfortunately, we are unable to perform the full-scale evaluation process of these operators like in our previous work [197] because it is hard to find the OCL specifications covering all these operations. However, some points for the mutation operators and classification need to be discussed [200].

First, the strategies that we used to design these mutation operators. We employed the following strategies. 1) The negation of the original operators. 2) The difference between OCL standard and common programming languages, like collection-related types, most common programming languages are 0-based, but OCL is 1-based. 3) The common mistakes, such as semantic misunderstanding (confusing $\rightarrow$ *including* and $\rightarrow$ *excluding*, for example). And we only design the operators for those operations presented in OCL standard library. Although some possible future OCL types or operations are proposed, like [44], the mutation operators for these types or operations are skipped due to not all OCL tools supporting them. These strategies may not be optimal, and they need further improvements during the future research period. Some of the operations do not have corresponding mutation operators in this work. Because, based on our strategies, we cannot find a suitable mutation operator for these operations, then we neglect them. Hopefully, this gap will be solved with the help of other OCL users or our further investigations.

In the context of primitive types in the OCL standard library, most operators are binary. The notable exception is the unary negation (-) operator for Real and Integer types, which is marked in the corresponding locations. For the specific instances in the proposed operators, like includesAll, the proposed mutation operators have the equivalent format as indicated. The condition is negated because this is a typical logic error made by specifiers, especially those inexperienced in OCL. However, this consistency in format is intended solely for illustrative purposes and does not signify any difference in their effect on the mutation testing process.

Second, the problem of equivalent mutants needs to be discussed. Equivalent mutants are the mutated specifications that already applied the mutation operator but did not modify the behaviour of the original specification, which makes these mutants not possible to be killed. Although the detection of equivalent mutants is an undecidable question [201], the existence of these mutants will affect the mutation testing process by the results of the mutation score. If too many of these mutants exist, the mutation score will be lower since the dividend in *Equation 3.17* will become larger than expected, and the test cases may be underestimated. In theory, some groups or mutation operators may generate equivalent mutants, like *Return False Mutation (RFM)* and *Return True*

Mutation (RTM), the mutation operators replace the value with false or true. When the original value is always evaluated as false or true, applied these mutation operators will not change the behaviour of the corresponding specifications. We need further experiments to evaluate which operators are more likely to produce equivalent mutants for the proposed mutation operators.

Furthermore, the usage of these mutation operators. Mutation testing is a fault-based technique that can be applied to many optimisation problems, like test case prioritisation and test case minimisation. The fault detection information can guide the corresponding optimisation process. Also, another possible usage for these operators is that the fault detection information can be used to validate the effectiveness of the test suite through many evaluation metrics. For example, APFD (Average Percentage of Fault Detection, which measures how quickly the test suite can detect faults) [131]. Indeed, how to generate a high test coverage test suite is an important topic. Still, for a given test suite, the mutation operators can be used to optimise this test suite, which will benefit the MBT process. In this research work, the mutation operators will be used to guide the TCO processes and also be used to evaluate the optimised test suite through (modified) APFD metric.

Then, some mutation operators may generate invalid mutants. For example, in expression $a + b$, we can generate the mutant a/b. The mutated expression is a valid mutant in most cases, but when b equals 0, this mutant will be invalid. Although the sample scenario can be detected easily by static analysis, it is hard to determine when b results from a function call. How to avoid this kind of scenario needs to be treated carefully. Also, some mutation operators may generate mutants that can be killed easily.

Also, the detection possibility for different groups of mutation operators. After various groups of mutants are generated based on detailed test strategies and specific test cases, these mutants do not have a uniform detection possibility. Some may be identified as weak mutants (most test cases can kill the mutant), and some may be hard to kill (only a few, even no test case, can kill the mutant). However, when we evaluate the mutation testing process, mutation scores treat them all the same and only reflect the proportion of killed mutants. So, how to improve this evaluation metric still need further consideration.

In the development of our mutation operator classification, our principal method involved grouping the proposed mutation operators in relation to the diverse facets of the OCL standard library. This primarily arises from our focus on altering the original operators, while the classification procedure is directed towards organising these modifications based on various strategic criteria. This difference in focus might result in the omission of some operators. However, this exclusion should not be seen as undermining the value or applicability of the proposed mutation operators.

Moreover, there may be questions about the exclusion of certain operators from the proposed set. For example, the Arithmetic Operator Deletion within the classification, where a *op* b is mutated to either a or b. This exclusion is intentional, as the proposed mutation operators primarily aim at modifying the original expressions rather than eliminating them. The elimination is a strategy that can be consistently implemented in actual mutation testing practices. For instance, with types such as Real, Integer, and UnlimitedNatural, it is often possible to retain the original value for unary expressions and one of the operands for binary expressions. Similarly, for Boolean type, one of the operands can usually be preserved. In the case of String types, if the original operator would return a String, the original expression can often be maintained. Additionally, methods can be omitted for Collection-Related Types without compromising syntactical correctness. These scenarios were excluded from the proposed operators for simplicity, but they remain viable strategies during the mutation testing process.

Still, about the classification of mutation operators, we may notice that some groups have an overlap between operators. For example, both RFM and REM return false when the type is Boolean. The reason for this scenario is different groups cluster mutation operators together based on the logic in common, which leads to this overlap. But when users choose to apply the particular group or groups based on their own needs, this overlap will not affect the mutation testing process.

Finally, the evaluation of the proposed mutation operators and the corresponding classification. We are currently experimenting with the effectiveness of these operators. However, there is a threat to validity if all assessments are performed by ourselves. Since all these mutation operators are designed based on our understanding of the OCL standard, there may have some biases and some operators still need further improvements. In this work, we also propose the classification of OCL mutation operators. The proposed classification introduces a variety of mutation operator groups designed to enhance the management of the testing process. It includes 17 supported groups that are compatible with the OCL standard library, with plans to integrate these into the AgileUML tool. Additionally, five categories (possible groups) are identified as having potential for implementation, though they require further investigation to assess their feasibility.

4.7 Conclusion

This chapter proposes a full set of mutation operators for the OCL standard version 2.4, mainly based on Clause 11 *"OCL standard Library"*. These operators will benefit the mutation testing process. We also propose the classification of the proposed operators, which provides more options to the users of OCL in languages such as ATL, QVT [1], etc. The classification will benefit the OCL participators in mutation testing, which allows them to choose which kinds of mutants will be generated based on their testing purposes. We have already implemented most of these operators within AgileUML. However, there still needs further implementation. Also, more evaluations of the effectiveness of the proposed operators and classification need to be performed in the future.

[1]ATL - ATLAS Transformation Language; QVT - Query/View/Transformation

Chapter 5

Test Case Prioritisation

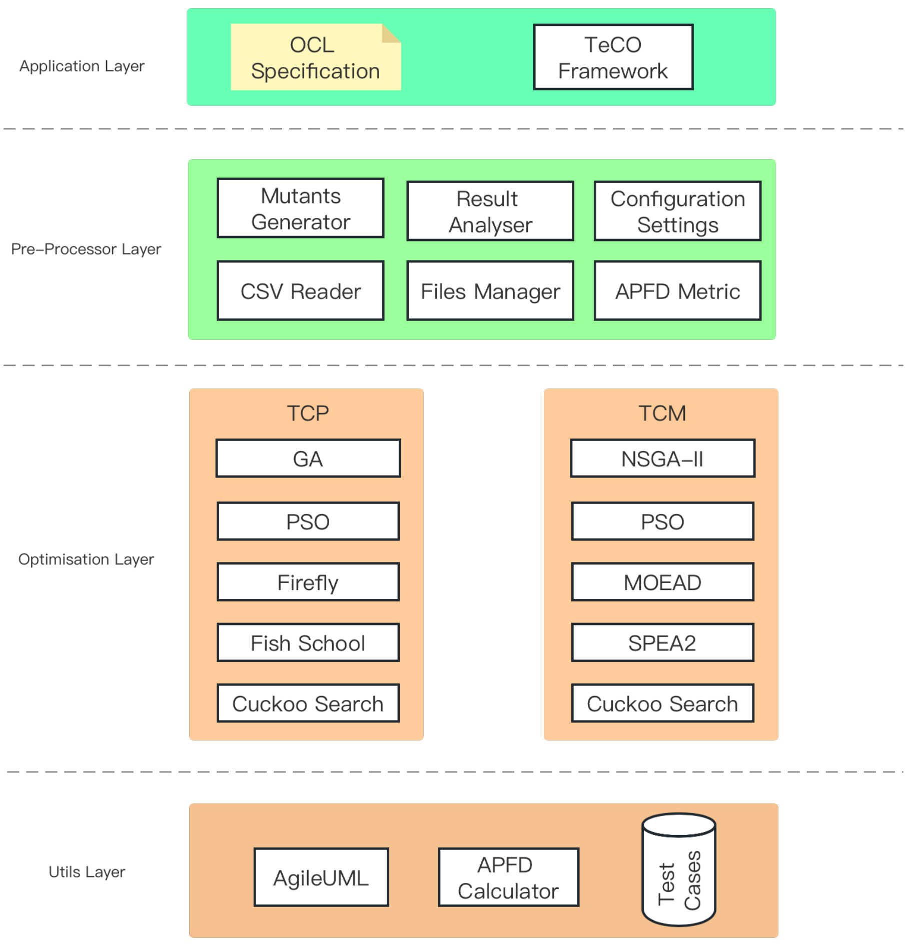


Figure 5.1: Structure of the Proposed Framework

In order to perform the TCO processes for the systems whose specifications are expressed in OCL, we proposed the TeCO (Test Case Optimisation) framework. The proposed framework supports the mutation testing, TCP and TCM processes for the corresponding OCL specifications and existing test cases. Combined with the strong MDE ability of the AgileUML tool, the proposed framework also provides the operations to generate test cases from the OCL specification and transfer the OCL specifications to the targeting real implementations. The overall structure of the TeCO framework is demonstrated by *Figure 5.1*.

The application layer includes two components which are OCL specifications and the TeCO framework. The OCL specification is used to describe the corresponding system by pre- and post- conditions, which acts as the input of the TeCO framework. The TeCO framework is the proposed framework that directly interacts with the users to provide various MBT activities.

The pre-processor layer provides the services that are required to perform the TCO processes. There are six components within this layer, which are the mutants generator, result analyser, configuration settings, CSV [1] reader, files manager and APFD metric. The mutants generator combines with AgileUML to generate the mutated OCL specifications. These mutants are used to guide the optimisation process by the corresponding faults (mutants) detection information. The detailed transition rules to generate these mutants are explained in *Chapter 4*. In this work, all mutants are first-order mutants, which means each mutant only applied a single transition rule once.

The result analyser collects the mutant detection ability for each test case and categorises this information into the corresponding files according to detailed OCL specifications. Configuration settings let users specify which optimisation algorithm, which TCO process or processes, original APFD

[1]Comma-Separated Values

metric or modified APFD metric will be used. Due to the results information being stored in a CSV format, the CSV reader component provides the necessary services to access these results. Finally, the last component, the APFD metric, accesses the APFD calculators in the utils layer to calculate the required evaluation result.

The optimisation layer contains the specific algorithms to perform the TCP and TCM processes. We adapted five distinct evolutionary algorithms for each of these activities. The details of these optimisation algorithms will be demonstrated in this (TCP process) and the next chapters (TCM process).

The last layer is the utils layer, which includes AgileUML, APFD calculator and test cases. AgileUML provides some fundamental MDE supports, such as test case generation and mutants generation, to the proposed TeCO framework. APFD calculator computes the original or modified APFD value for the provided test suite according to the user configuration. The test cases component is used to store the generated or existing test cases.

In the following parts of this chapter, the five TCP algorithms implemented in the TeCO framework will be described. And the corresponding evaluation metric to the TCP process will also be discussed. Moreover, we modified one well-known metric, APFD, to make this metric more suitable for the MBT and mutation testing environment.

5.1 TCP Process

In the TeCO framework, as demonstrated in Chapter 3, five optimisation algorithms are implemented to conduct the TCP process. These algorithms are Genetic Algorithm (GA), Particle Swarm Optimisation (PSO), Firefly, Fish School and

Cuckoo Search algorithm. The following are the details about how they are applied to the TCP problem.

The following are essential aspects that need to be highlighted regarding these prioritisation algorithms.

- *Genetic Algorithm*

Within the genetic algorithm, the population is a set of individuals. The solution to an optimisation problem is called an individual, which is represented as a sequence of variables called chromosomes or gene strings. In TCP problems, the chromosome is the permutation of the test cases, so the chromosome length is precisely the same number of test cases within the test suite, like *Figure 5.2*.

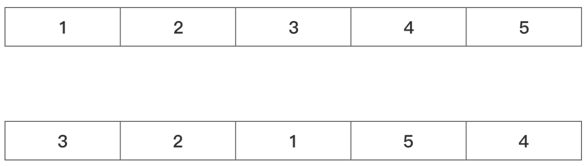


Figure 5.2: Example of Individual

In this example, there are two individuals, which are {1, 2, 3, 4, 5} and {3, 2, 1, 5, 4}. The sequence of numbers is the execution order of the corresponding. The first individual will execute the first test case at the beginning, then the second, third, fourth, and fifth. However, the second individual will start with the third test case, then the second, first, fifth and finally the fourth one. Due to the TCP problem being modelled as a permutation problem, each distinct number will only appear once in the individual.

In the context of the TCP problem, the fitness value can be represented by the evaluation result of the original or modified APFD metric. Further details regarding the APFD metric will be discussed in subsequent sections.

In this thesis, the selection operator is tournament selection. This operator involves selecting a random subset of individuals from the population, and then choosing the best individual from this subset to pass to the next generation.

The crossover operator can be implemented in various ways, such as one-point crossover, two-point crossover, and uniform crossover. To better suit the TCP problem, the PMX [2] operator is adopted in the TeCO framework.

In PMX, two crossover points are randomly selected on the parent chromosomes. The segment between these crossover points is swapped between the parents to generate offspring. To maintain consistency and avoid duplicate elements in the offspring, a mapping is created based on the swapped segments, allowing for the replacement of any conflicting elements with their corresponding mapped values. An example is shown as *Figure 5.3*.

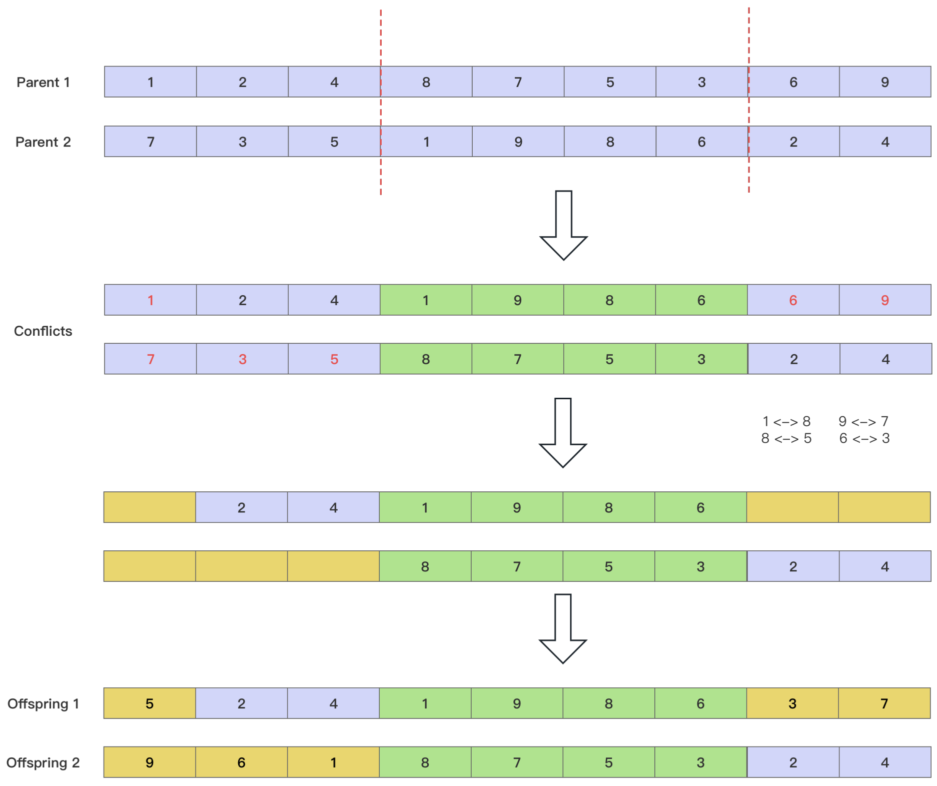


Figure 5.3: Example of Partially Mapped Crossover

[2]Partially Mapped Crossover

The mutation operator can be implemented in various ways, such as bit-flip mutation, swap mutation, and inversion mutation. In the TeCO framework, the mutation operator is the permutation swap operator, randomly swapping two chromosome positions.

For the remaining parameters within the GA, we set the population size to 50, the maximum iteration to 1000, the crossover rate to 0.9, and the mutation rate to $1/TestSize$.

These evolutionary operators and parameter settings are the default implementation within the TeCO framework. Due to this work not focusing on the best settings of the optimisation algorithms, we did not systematically compare the performance of different evolutionary operators and parameter settings. Users can modify or change these settings according to their preferences.

- *Particle Swarm Optimisation*

PSO by using positions and velocities to find the optimal solution, it is hard to represent the problem by permutation like GA. Intuitively, PSO is an optimisation algorithm to solve double problems. Then, how to map the double problem to the permutation problem became the critical point in solving the TCP problem.

In the TeCO framework, we solved this situation through the smallest value mapping. The search space of the TCP problem has n dimensions, where n equals the number of test cases. Then, each particle has n positions and velocities, which are *double* type. When mapping the double problem to the permutation problem, the smallest position value will be mapped to 1, and the largest position value will be mapped to n. *Figure 5.4* gives an example of this process. The ranking in double positions will be mapped to permutation numbers corresponding. In case there is a tie in positions, we give the first position a smaller permutation number.

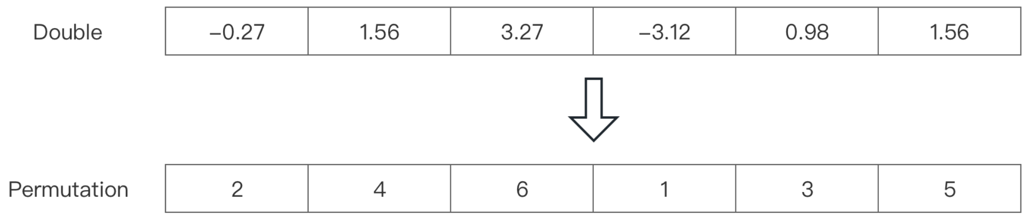


Figure 5.4: Mapping Process

In the TeCO problem, for solving the TCP problem, the original or modified APFD metric always serves as the fitness function.

When we adapted PSO to the TCP problem in our TeCO framework, the parameter settings were as followings: the initial positions are randomly in [0, 4], and the initial velocities are in [-4, 4]. The value of w is 0.9, $c1$ is 2, $c2$ is 2, and α is 0.975. The maximum iteration, same as GA, is 1000, while the size of the particles is 50. Again, all these settings are the default value within the TeCO framework and may not be optimal for all scenarios. Users can modify these values according to their own preferences.

- *Firefly Optimisation*

As same as the PSO algorithm, the problem is modelled as a double problem and by using smallest value mapping to covert to permutation problem.

The firefly algorithm updates the movement of each individual according to *Equation 3.6*. In the TeCO framework, we do not calculate every relationship between each firefly. Alternatively, we instead j with the global best solution, which means the firefly will always move toward the global best solution.

In the proposed TeCO framework, same as GA, we set the population size as 50, and the maximum iterations are 1000.

γ is 1, β_0 is 1 and α is 0.2.

- *Fish School Optimisation*

The fish school algorithm is a suitable method for solving double problems, using the smallest value mapping and (modified) APFD metric for mapping function and fitness evaluation. After randomly generating an initial population, the algorithm iteratively optimises to find the optimal solution.

Each fish in the school conducts a local search to explore promising regions in the search space through *Equation 3.7*. In our implementation, this value change from 1 to 0.001 as the iteration increase. The new position $x_i(t+1)$ will only be accepted when this movement increases the fitness value.

During the volitive movement within the fish school algorithm, Euclidean distance is used to calculate the distances between the fish and the barycentre. The $step_{vol}$ in volitive movement controls the distance of movement, which changes from 1 to 0.01 as the iteration increases.

In the proposed TeCO framework, same as GA, we set the population (school) size as 50, and the maximum iterations are 1000. In the default setting, the minimum weight of each fish is 1, and the maximum weight is 5000.

- *Cuckoo Search Optimisation*

During the cuckoo search optimisation, the fitness function is a measure of how well a particular solution performs with respect to the problem at hand, for here, it is still the original or modified APFD value.

The Lévy flight is a type of random walk used to generate new candidate solutions for the optimisation problem. The Levy flight is based on the Levy distribution, which is characterised by a heavy tail and a large variance, resulting in occasional large steps in the search space. In order to better implementation, in the TeCO framework, we calculate flight through Mantegna's algorithm by using *Equation 5.1*.

$$StepSize_i = u * |v|^{-1/\alpha} \tag{5.1}$$

In this equation, u and v are random numbers generated from a Gaussian distribution with mean 0 and variance 1, and α is a parameter that controls the step size, which we use 1.5 in implementation. The step size represents the amount by which the i^{th} dimension of the current solution should be modified to generate a new solution.

In the proposed TeCO framework, same as GA, we set the population (nest) size as 50, and the maximum iterations are 1000. And in the default setting, the percentage of abandoned nests randomly generate from the range [10, 25].

5.2 TCP Metrics

The TCP problem aims to find the best execution order of the test suite to benefit the testing process. When the test suite is prioritised, it is essential to assess the effectiveness of the proposed approach through evaluation metrics. An evaluation metric plays a crucial role in the TCP problem, which helps to quantify the performance of the proposed TCP approach and compare the different approaches. By utilising a well-defined evaluation metric, researchers can quantitatively compare different TCP approaches and identify the most effective one for a given testing scenario.

From the review conducted by Khatibsyarbini [130], the current widely used evaluation metrics within the TCP area are shown in *Figure 5.5*.

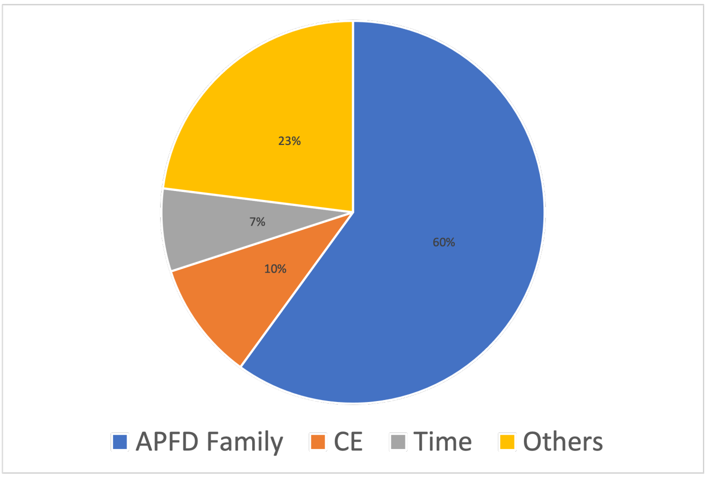


Figure 5.5: Evaluation Metrics for TCP Process

In the statistical result, around 60% of the research chose the APFD family, original APFD and $APFD_c$ as the evaluation metric. Recall from the related works that the APFD metric is utilised to determine the average percentage of fault detection rate for a given test suite and assess the early detection capabilities of the test suite for system faults. The APFD metric produces a value between 0 and 1, with a higher value indicating an increased ability to detect faults earlier. Essentially, this metric measures the speed at which the test suite identifies system faults. The result of the original APFD metric can be calculated by the *Equation 5.2*.

$$APFD = 1 - \frac{TF_1 + TF_2 + \cdots + TF_m}{nm} + \frac{1}{2n} \tag{5.2}$$

In the presented equation, TF_i represents which test case first detects the i^{th} fault, while n indicates the total number of test cases in the test suite, and m represents the total number of faults within the software system. In the context of MDE, because system faults can not be known in advance, the faults also can be instead of the artificially injected defects.

Nonetheless, the APFD metric assumes that all system faults have equal severity and each test case requires the same amount of testing effort. In reality, each test case has a distinct testing cost, and the severity of faults may vary in their impact during the testing process. Therefore, the $APFD_c$ metric has been introduced, which accounts for both testing costs and fault severity, enhancing the accuracy of the APFD. The metric for $APFD_c$ is calculated as *Equation 5.3*.

$$APFD_c = \frac{\sum_{i=1}^{m}(f_i * (\sum_{j=TF_i}^{n} t_j - \frac{1}{2} t_{TF_i}))}{\sum_{j=1}^{n} t_j * \sum_{i=1}^{m} f_i} \tag{5.3}$$

Similar to the APFD computation, the equation includes f_i, which represents the severity of the i^{th} fault. TF_i denotes the test case that detects the i^{th} fault, and t_j refers to the testing cost of the relevant test case. Additionally, n represents the total number of test cases in the test suite, while m signifies the number of faults.

The above definition of $APFD_c$ incorporates both the cost and fault severity of test cases. However, due to its ambiguous or system-specific estimation methods for fault severity and test case cost, the applicability of this metric is limited. Consequently, most researchers prefer using APFD to evaluate the effectiveness of their work. From the statistical result, 60% publications used the APFD family as the evaluation metric. In detail, 51% works used the APFD metric and 9% used $APFD_c$ metric. Despite the fact that the majority of research used this family of metrics to evaluate the effectiveness of the proposed approach, APFD has its own shortcomings and is not ideally suited for mutation testing and the MDE environment. We will analyse it in detail and introduce a modified version of the APFD metric in the next section.

The second widely used TCP evaluation metric is CE (coverage effectiveness) metric. In the TCP process, CE evaluates the effectiveness of the proposed approach by using a combination of techniques, such as branch coverage, statement coverage, and function coverage. These techniques are used to measure the extent to which a test suite covers the different aspects of the SUT and how quickly these aspects have been covered. CE values fall between zero and one, with higher values indicating greater effectiveness in terms of coverage.

The time execution metric is another primarily used evaluation metric of the TCP process, and is distributed 7% of the total used metric. The time execution metric is mainly used to verify the effectiveness of the proposed TCP approach by the time-related characters, such as the average time to find the defects and the time to find the first defects within the SUT.

Lastly, the remaining 23% of the overall distribution comprises several types of metrics, most of which have been adapted from the APFD metric or coverage-related metrics.

Overall, the evaluation metrics used in TCP are essential in assessing the effectiveness of the proposed approach and identifying the most effective one for a given testing scenario. These metrics allow researchers and practitioners to measure the effectiveness of a TCP approach in early fault detection ability during the testing process and in delivering higher-quality software systems.

5.3 Modified APFD Metric

We mentioned that the original APFD metric has its own shortcomings, especially under the mutation testing and MDE scenario. This section will analyse the corresponding information in detail and propose a modified version of the APFD metric to overcome these imperfections.

The range of APFD metric intends to range from 0 to 1, and a higher value indicates earlier fault detection ability. However, the actual maximum APFD value is $1 - \frac{1}{2n}$, when the first test case detects all defects. At the same time, the minimum APFD value is $\frac{1}{2n}$, when all defects are detected by the last test case.

The actual result cannot range from 0 to 1. *This is the first imperfection of the original APFD value.* Consider one extreme scenario when the test suite has only one test case that can detect all system faults. The APFD value will equal 0.5 (the highest value $1 - \frac{1}{2n}$), which is a relatively low result but is already the maximum value.

The second point is that the original APFD value assumes all defects are detectable. This assumption is unfeasible when performing mutation testing in an MDE environment. Mutation testing injects mutants into the system and then validates whether the test cases can kill the artificial mutants. Due to mutated specifications being generated based on the mutation rules, we cannot guarantee that the test suite can detect all system defects in this scenario.

One possible solution to undetectable fault is that we assume TF_i is the last position in the test suite. However, this cannot distinguish whether the test suite cannot detect the defects or the last test case detects the defect. Walcott proposed another possible solution, that when a fault is missed, the TF_i equals the number of test cases plus one [202]. This is a feasible solution to the second imperfection of the APFD metric, but in the extreme scenario that all faults can not be detected, the APFD value will be $-\frac{1}{2n}$, which does not range from 0 to 1 any more.

The third imperfection is that even when two test orders have the same APFD value, one may still be better than the other. Considering that there are three test cases and six faults, the fault detection information is shown as *Figure 5.6.*

	F1	F2	F3	F4	F5	F6
T1		✓	✓	✓	✓	
T2	✓	✓	✓			
T3				✓	✓	✓

Figure 5.6: An Example of Fault Detection

There are two test order, $\{1, 2, 3\}$ and $\{3, 2, 1\}$. These two orders have the same APFD value since the $TF_1 + TF_2 \cdots + TF_m$ is the same. For the first one is $2 + 1 + 1 + 1 + 1 + 3$, and for the second one is $2 + 2 + 2 + 1 + 1 + 1$, where all of them are 9. TCP aims to detect faults as early as possible, even if the testing process is prematurely halted. From the APFD results, these two test orders have the same effectiveness. However, if the testing process is halted after executing only one test case, the first order can detect four faults, while the second order can detect three faults. In this scenario, the original APFD value may not adequately reflect the effectiveness of the TCP approach.

To summarise, the original APFD metric suffers from the following three disadvantages. Firstly, its actual value range is not ranged between 0 and 1. Secondly, the metric may not provide an accurate assessment when faults are undetectable. Finally, even when two test orders have identical APFD values, one may still be superior to the other.

While some studies, such as [158], have attempted to address the issues identified with the original APFD metric, none have specifically tackled its application within the context of MDE. To address these shortcomings, we propose introducing a reward system that increases the APFD value when a fault is detected by the first test case. Simultaneously, we suggest including a penalty system that reduces the APFD value if the fault cannot be detected by any test case.

We proposed a modified APFD metric in this work. The modified APFD value can be calculated by *Equation 5.4* and *Equation 5.5*.

$$APFD_m = 1 - \tfrac{TF_1+TF_2+\cdots+TF_m}{nm} + \tfrac{1}{2n} + (\alpha - \beta) * \lambda \quad (5.4)$$

$$\lambda = \frac{1}{2nm} \quad (5.5)$$

In this metric, the first part is the same as the original APFD metric. We called this part a reward and penalty factor in the second part $(\alpha - \beta) * \lambda$. α is the number of faults that can be detected by the first test case, and β is the number of faults that cannot be detected by any test case. λ is the reward or penalty value, which equals to $\frac{1}{2nm}$.

The development of the modified APFD metric was driven by the need for a more suitable tool for evaluating test suites in the context of MDE and mutation testing and adjusting the scoring range of the original version. This modification aims to simplify the interpretation of evaluation results, focusing less on the specific characteristics of the test suite and system defects. For instance, when assessing two test suites, if their evaluation scores are close, it implies comparable effectiveness between them. This strategy could mitigate the issue highlighted earlier, where the outcome might appear relatively moderate yet already represents the maximum achievable value.

To fulfil our objective, the metric was adjusted to penalise undetectable defects, thereby enabling the lowest possible score within the modified APFD metric to be zero. We also intend to design the metric to reward early fault detection within a test suite more effectively. Currently, we extend rewards solely to the initial test case detected, allowing the metric to attain a maximum score of one. The exploration of a dynamic approach to reward early fault detection more comprehensively is an avenue we intend to investigate further, seeking to enhance the utility of the metric and its relevance in future applications.

Using the modified APFD value, this metric ranges from 0 (all defects are undetectable) to 1 (the first test case detects all defects). Through the improvements, this metric has the reward and penalty mechanism to deal with the aforementioned situations.

During the evaluation phase of this research, we will utilise both the original and the modified versions of the APFD metrics. These metrics will guide the experimental procedures and play a key role in evaluating the effectiveness of the optimised test suite. To gain a comprehensive understanding of our results, we will compare and investigate the data obtained using these two metrics.

Chapter 6

Test Case Minimisation

This chapter will focus on another test case optimisation technique, TCM, for the systems whose specifications are expressed in OCL. TCM, also known as test case reduction or test suite minimisation, is a technique used in the software testing process to reduce the number of test cases while maintaining the overall effectiveness of the test suite. The main goal of the TCM process is to reduce the efforts for the software testing process without compromising testing effectiveness.

Unlike the TCP problem, which can be modelled as a single-objective optimisation process, the TCM problem is consistently modelled as a multi-objective optimisation process. The TCM process aims to decrease the number of test cases while preserving the testing capabilities. Naturally, one objective corresponds to the test suite size, while the other objective(s) will relate to the testing ability.

In the multi-objective optimisation algorithm, the objectives are often conflicting, and there is no single optimal solution but instead a set of Pareto optimal solutions [203]. A solution is considered Pareto optimal if no other solution in the search space can dominate this solution. To better understand the Pareto optimal, the following concepts are necessary to discuss.

- Domination: Solution A dominates solution B when all objectives of A are better than or equal to those in B, and at least on objective in A is strictly better than that in B. In other words, solution A dominates solution B if there is no objective in B outperforms A and there is at least one objective in A outperforms that in B.

- Non-dominated solutions: Non-dominated solutions are not dominated by any other solution in the search space. These solutions represent the best trade-offs between the conflicting objectives in a multi-objective optimisation problem.

- Pareto optimal set: The Pareto optimal set, also known as the Pareto front, is the set of all non-dominated solutions in the search space. No solution in the Pareto optimal set can be improved in one objective without degrading at least one other objective.

The Pareto optimal set helps decision-makers understand the trade-offs between the different objectives and select the most appropriate solution based on their preferences or specific constraints. For the TCM problem, each solution within the Pareto optimal set can be used to describe the minimised test suite. But in the TeCO framework, the selection criteria for the minimised test suite are based on achieving the maximum fault detection ability with the smallest size.

In the following parts of this chapter, the five TCM algorithms implemented in the TeCO framework will be explained. And the corresponding evaluation metric to the TCM process will also be discussed.

6.1 TCM Process

In the TeCO framework, as demonstrated in related works, five optimisation algorithms are implemented to conduct the TCM process. These algorithms are Non-dominated Sorting Genetic Algorithm II (NSGA-II), Particle Swarm Optimisation (PSO), Multi-objective Evolutionary Algorithm based on Decomposition (MOEA/D), Strength Pareto Evolutionary Algorithm II (SPEA2) and Cuckoo Search algorithm.

The following are essential aspects of how we adapted this optimisation algorithm to systems whose specifications are expressed in OCL.

- *NSGA-II*

In the TeCO framework, the bit data structure is used to represent individuals, and one example is *Figure 6.1.*

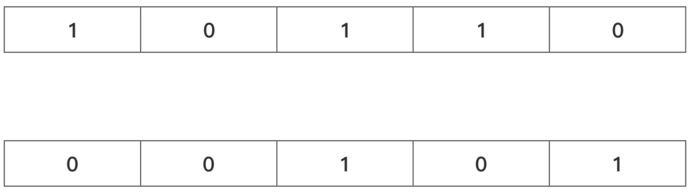


Figure 6.1: Individuals in NSGA-II

In this example, the test suite has five test cases, and in these two individuals, test cases {1, 3, 4} and {3, 5} will be included in the minimised test suite separately. In this TCM problem, the length of the chromosome equals the number of test cases within the original test suite, and each bit position is used to describe whether the corresponding test case is selected for the minimised test suite, where the value of 1 indicates that the test case has been selected and 0 indicates that it has not been selected.

The configurations for the three evolutionary operators, selection, crossover and mutation operators, are similar to the

GA. The selection operator is tournament selection, which randomly selects a subset of the population from the original one and then chooses the best one according to the fitness value. A larger tournament size increases the chance of selecting the best individual, but also increases the computational cost of the selection process, usually 2.

The crossover operator by default in the TeCO framework is two points crossover, which selects two crossover points in the parent solutions, and swaps the corresponding parts between these points to create new offspring solutions. *Figure 6.2*, and the crossover rate has been set to 0.9.

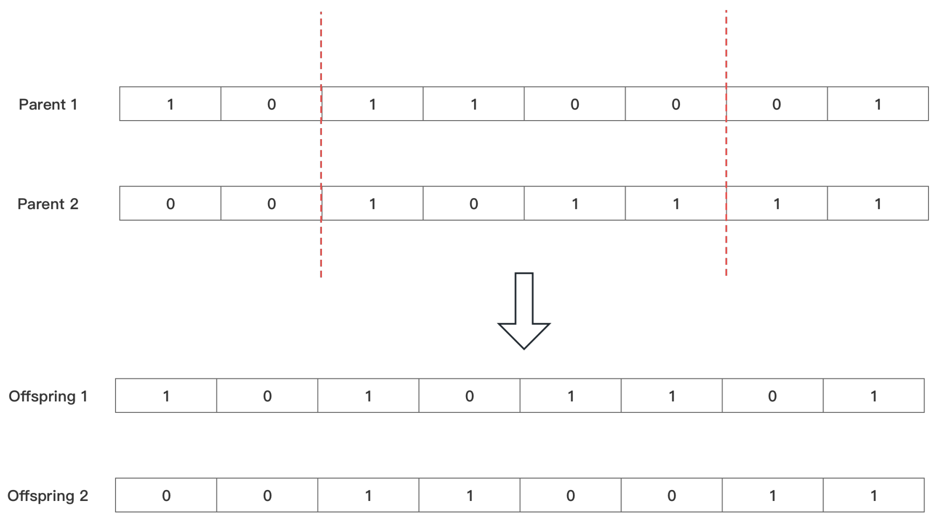


Figure 6.2: Two Points Crossover

The default mutation operator is bit flip mutation, which flips each bit position (0 to 1 and 1 to 0) according to the mutation probability, and this rate is set to $1/TestSize$. For the remaining parameters within the NSGA-II, we set the population size to 50, and the maximum iteration to 1000.

One of the most critical processes within the NSGA-II algorithm is fitness evaluation, which corresponds to the multi-objectives. The TCM process aims to save testing efforts by reducing the number of test cases, so the first optimisation objective is *Number of Test Cases.*

Also, the reduction should not compromise the fault detection ability or decrease as little as possible, and then the second optimisation objective is *Fault Detection Ability.* In case there is no actual fault detection information within the MDE or OCL context, this information will be instead by the mutant detection ability.

The TCM process only decides which test case should be selected into the minimised test suite and does not change the order of the test cases. Intuitively, this process will not be related to the APFD metric, but actually, this objective will help to choose which redundant test case should be removed. As an illustration, suppose that test case 2 and test case 5 both have the ability to identify identical faults. In this scenario, the optimisation process will decide to remove which test case according to the detailed detection ability of test cases 3 and 4 rather than randomly remove one arbitrarily. This preference is based on eliminating which one can benefit from the early fault detection ability. Then, the third objective is *Original or Modified APFD metric.*

In short, there are three objectives, the number of test cases, fault or mutant detection ability and the original or modified APFD metric.

- *Particle Swarm Optimisation (PSO)*

In the TeCO framework, the PSO algorithm used to solve the TCM problem follows the same general process as described in the previous chapter. However, the main difference is how to map the double problem to TCP or TCM problem. For TCP, the smallest value mapping is used to map each double value to the position of the test case. On the other hand, for TCM, a random larger value is used to solve the mapping problem.

The random larger value is used to map the double problem to the TCM problem. The length of each particle is N the same as the number of test cases, and then for each particle, there will be N double values during the optimisation process. *Figure 6.3* demonstrates the mapping process.

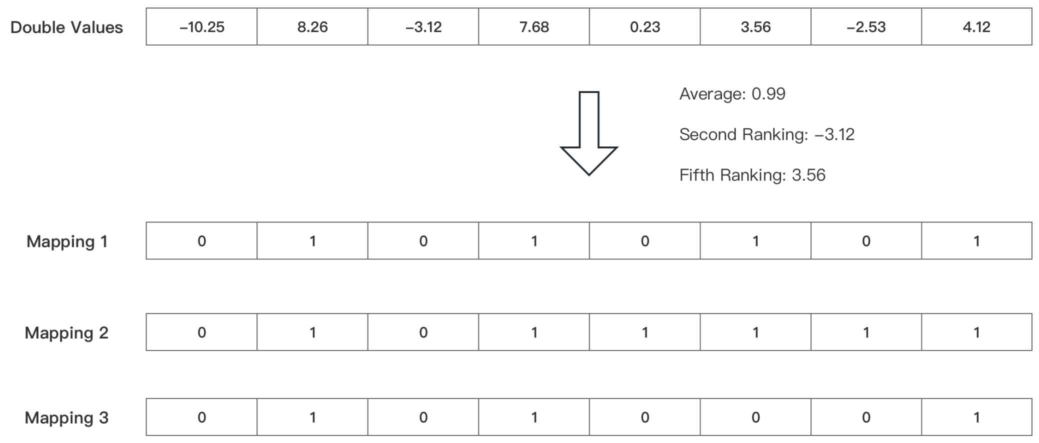


Figure 6.3: TCM Mapping Process

The mapping process first randomly chooses one double value within the individual or the customised value, like the average value. Then, each double value within the individual that is larger than the chosen value will be selected into the minimised test suite. In the provided value, mapping 1 chooses the average value, which lets test cases {2, 4, 6, 8} are included. Mapping 2 determined to use the second-ranking value, -3.12, which leads to the test cases {2, 4, 5, 6, 7, 8} have been included. Then mapping 3 consists of the test case {2, 4, 8} through the fifth-ranking value 3.56. The random value is used here, and through the stochastic searching ability of the optimisation algorithm to find the optimal solutions across the whole search space.

The median value is not recommended to be used during the mapping process due to this value will let only exact $\frac{N}{2}$ or $\frac{N}{2}-1$ [1] test cases will be included in the minimised test suite.

[1]Which depending on the boundary condition.

The average value has not been excluded due to this value is sensitive to the extreme value, which lets the average value may not be constrained to the fixed-size test suite during the minimisation process.

The objectives used in this algorithm are the same as NSGA-II, the number of test cases, fault or mutant detection ability and the original or modified APFD metric.

When we adapted PSO to the TCM problem in our TeCO framework, the parameter settings were as followings: The maximum iteration is 1000, the population size is 50, the range of each double value is [-1000, 1000], and the number of objectives is 3.

- *MOEA/D*

In the TeCO framework, for this algorithm, we applied the original MOEA/D algorithm that is supported by *JMetal* framework [204]. In order to keep the consistency across different TCM algorithms, we also set the population size to 50 and the maximum iterations to 1000. Same with the previous algorithms, the objectives are the number of test cases, fault or mutant detection ability and the original or modified APFD metric, and using random larger value to map double and TCM problems.

- *SPEA2*

In the SPEA2 algorithm, reproduction is a crucial phase that contributes to the generation of new offspring, which potentially possess better fitness values compared to their parents. This process involves selecting individuals based on their fitness and using genetic operators such as crossover and mutation to create a new population.

In the TeCO framework, the default maximum generation is 1000, population size is 50, the selection operator is tournament selection, the crossover operator is two points crossover, the crossover possibility is 0.9, the mutation operator is the bit flip operator, and the mutation rate is $1/TestSize$. The optimisation objectives of the SPEA2 are the number of test cases, fault or mutant detection ability and the original or modified APFD metric. As with the other adapted algorithms, the population size is set to 50.

- *Cuckoo Search*

Different from the TCP problem, the double problem will be mapped to a bit problem instead of a permutation problem in the TCM process. For here, we still use the random larger value to perform the mapping process. And in the fitness evaluation process, the aim will be transferred from the specific evaluation metric to the Pareto optimal set.

In the proposed TeCO framework, same as other TCM algorithms, we set the population (nest) size as 50, and the maximum iterations are 1000. The optimisation objectives of this algorithm are the number of test cases, fault or mutant detection ability and the original or modified APFD metric.

6.2 TCM Metrics

The TCM problem aims to benefit the testing process by executing fewer test cases to achieve the same testing effectiveness. When the test suite is minimised, it is essential to assess the effectiveness of the proposed approach through evaluation metrics. An evaluation metric plays a crucial role in the TCM problem, which helps quantify the performance of the proposed approach.

One of the most widely used evaluation metrics is code coverage, which measures the proportion of code that the minimised test suite has covered. According to the granularity, the code coverage includes statement coverage, statement coverage, path coverage, etc. These different specific coverage-based metrics correspond to the proportion of executable statements, decision points and possible execution paths that have been examined by the minimised test suite. These metrics are useful for evaluating the effectiveness of the proposed TCM approach, and ensuring the removed test cases will not decrease the code coverage ability.

Test execution time is also an important metric to evaluate the effectiveness of the TCM process. This type of metric includes total and average test case execution time. Since one of the objectives of the TCM process is to speed up the testing process, time is an apparent measurement to evaluate the effectiveness of the proposed approach.

Fault detection metrics measure the performance of the minimised test suit in terms of fault detection ability. The TCM process benefits the testing process by removing redundant test cases. Still, the fault detection ability is necessary to guarantee not to be compromised or lost any little as possible. This type of metric examines the fault detection ability of the minimised test suite against the original one to measure the effectiveness of the proposed TCM approach. This metric is one of the evaluation metrics we used in this thesis.

Size metric measures the number of test cases or the reduction rate of the minimised test suite. The reduction ratio measures the percentage of test cases that have been removed during the optimisation process. This metric is a significant evaluation metric since the main aim of the TCM process is to reduce the number of test cases. This is the other evaluation metric that we used in the evaluation process.

In this thesis and the proposed TeCO framework, the *Equation 6.1* and *Equation 6.2* will be used to evaluate the effectiveness of the TCM process.

$$(1 - \frac{number\ of\ test\ cases\ in\ the\ reduced\ test\ suite}{number\ of\ test\ cases\ in\ the\ original\ test\ suite}) * 100\% \quad (6.1)$$

$$(1 - \frac{number\ of\ faults\ detected\ by\ the\ reduced\ test\ suite}{number\ of\ faults\ detected\ by\ the\ original\ test\ suite}) * 100\% \quad (6.2)$$

The TCM optimisation process aims to decrease the number of test cases within the test suite while maintaining the fault detection ability. These two equations correspond to the purposes of the TCM process. One relates to the reduction rate of the minimised test suite, and the other corresponds to fault detection ability. The aims are to reduce as many test cases as possible while losing as little fault detection ability as possible. Under the MDE or OCL environment, real system faults can not be used directly in the evaluation process. The system defects will be simulated by mutants mentioned in previous sections. The original or modified APFD metric will also be used in this thesis to analyse how the TCM process will affect these evaluation metrics.

Chapter 7

Evaluation

7.1 Research Questions

In order to evaluate this research work systematically, the following five main research questions (RQs) were proposed in Chapter 1.

• **RQ 1: *Effectiveness.* What is the effectiveness of the TCO processes within the context of OCL?**

RQ 1.1: Effectiveness for the TCM process.

RQ 1.1.1: What is the test suite reduction rate achieved by the adapted algorithms during the TCM process?

RQ 1.1.2: What is the fault detection capability of the adapted algorithms during the TCM process?

RQ 1.2: Effectiveness for the TCP process.

RQ 1.2.1: What are the performances of the adapted TCP algorithms under the (modified) APFD metric?

RQ 1.2.2: How does the performance of the adapted algorithms during the TCP process compare with a random approach?

RQ 1.3: Does the sequence of TCP and TCM processes affect the overall optimisation process?

- **RQ 2: *Scalability.* How scalable are the proposed algorithms in handling real-world OCL specifications with varying complexities?**
- **RQ 3: *Comparison.* Which of the adapted TCO algorithms performs optimally during the optimisation processes?**

RQ 3.1: During the TCM process, which algorithm demonstrates superior performance?

RQ 3.2: During the TCP process, which algorithm performs the best?

RQ 3.3: Which type of OCL specification is most compatible with the proposed algorithms?

- **RQ 4: *Metric.* How to evaluate the modified APFD metric?**

RQ 4.1: What are the differences between the original and the modified APFD metric?

RQ 4.2: How does the TCP process perform when evaluated under the original and modified metrics?

- **RQ 5: *Efficiency.* What is the time overhead when the TCO processes are applied to the systems whose specifications are expressed in OCL?**

RQ 5.1: What is the overhead for each proposed algorithm during the TCM process?

RQ 5.2: What is the overhead for each proposed algorithm during the TCP process?

RQ 5.3: What is the overhead during the pre-process phase preceding the optimisation processes?

This study primarily investigates the practicability of implementing TCO strategies within systems whose specifications are expressed by OCL. Thus, RQ 1 is proposed to evaluate the effectiveness of this application.

The TCM process aims to reduce the size of test cases while maintaining its fault detection ability. Accordingly, the effectiveness of minimisation algorithms is frequently evaluated via the test suite reduction rate and the fault detection capability compared to the original one. RQ 1.1 is introduced to examine the effectiveness of the TCM process.

The TCP process is designed to optimise the testing procedure by rearranging the order of the test sequence. Thus, RQ 1.2 is presented to investigate the effectiveness of the TCP process. Furthermore, RQ 1.3 determines whether the sequence of the TCM and TCP processes has any relevance to the optimisation results. While, RQ 2 measures how the scalability of the proposed algorithms can be applied to the OCL specifications with different complexities.

As we have adapted five distinct optimisation algorithms for the TCM and TCP processes separately, it becomes crucial to compare their relative effectiveness. Beyond the intrinsic nature of the algorithms, which type of OCL specifications are better compatible with the proposed algorithms is also necessary to be discussed. This subject is examined by RQ 3.

The APFD metric has been modified to accommodate the context of MBT better. RQ 4 has been proposed to evaluate the differences between the original one and the performance of the TCP process under the adjusted metric.

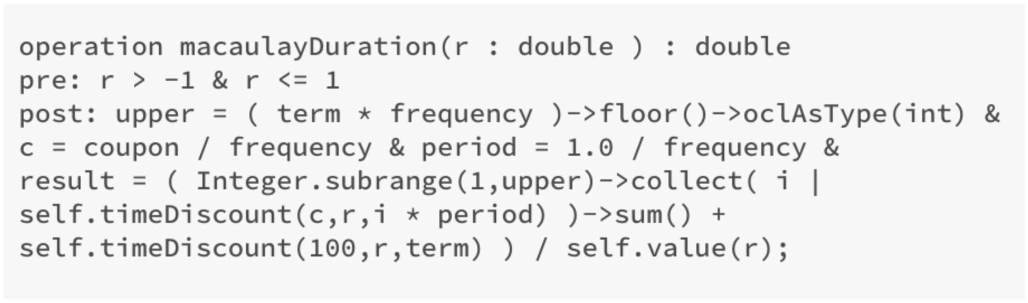
While effectiveness is a key consideration, the efficiency of the proposed algorithms is equally crucial. RQ 5 is designed to measure the computational overhead of these algorithms, and efficiency is assessed by considering time consumption.

7.2 Evaluation Process

To answer the research question and assess our proposed method, we utilise four distinct OCL specifications collected from real-world studies: Bond, Interest Rate, MathLib and UML2PY. These studies vary in their complexity. The specifics of each case study, including their size (as measured by lines of code) and the number of operations, are demonstrated in *Table 7.1*. *Figure 7.1* provides an example OCL specification for a single function macaulayDuration within in the Bond case study, which including the function name, pre-conditions, and post-conditions.

Table 7.1: Case Study Details

Study Name	Size	Operation
Bond	93	5
Interest Rate	41	2
MathLib	212	15
UML2PY	1053	18



```
operation macaulayDuration(r : double ) : double
pre: r > -1 & r <= 1
post: upper = ( term * frequency )->floor()->oclAsType(int) &
c = coupon / frequency & period = 1.0 / frequency &
result = ( Integer.subrange(1,upper)->collect( i |
self.timeDiscount(c,r,i * period) )->sum() +
self.timeDiscount(100,r,term) ) / self.value(r);
```

Figure 7.1: OCL Example

During the experiment process, the AgileUML tool will generate corresponding test cases (forming the original test suite), mutated specifications, and executable JAVA files when given OCL specifications as input. For a particular function, the number of generated test cases (at a functional level) for the four case studies ranges from 2 to 243.

The mutated OCL specifications are created in accordance with the corresponding OCL expressions and invariant variables, using the detailed transition rules shown in *Chapter 4*. The test cases at the functional level are constructed using the boundary value of each parameter and the various combinations of these parameters. The boundary value analysis in our context involves both lower and upper bounds of numerical ranges (also the values around and within the boundary) as critical points for test case generation. The empty, null, and random values are used for the String type to perform boundary testing. Also, the test values can be directly configured by the end user.

The selection of boundary value analysis over more advanced methods, as discussed in the systematic literature review, warrants justification. The primary rationale behind this choice is twofold. First, boundary value analysis offers a straightforward method for identifying critical faults with minimal computational resources. Its simplicity and directness in pinpointing potential error-prone areas make it an attractive option for initial testing phases. Second, while advanced testing techniques provide broader coverage and deeper insights into potential system vulnerabilities, they often require more sophisticated setup and significantly greater computational effort.

However, it is important to acknowledge the limitations of relying exclusively on boundary value analysis. Advanced testing methodologies can supplement boundary value analysis by providing a more comprehensive examination. The current version of the AgileUML tool solely supports boundary testing, while future enhancements could involve integrating these sophisticated approaches to enrich the depth and breadth of testing strategies.

Algorithm 1 shows the general process for the TCM activ-

Algorithm 1 General Test Case Minimisation Process

```
Input: OCL Specification
Output: Selected Test Cases
 1: AgileUML ← OCL Specification (mutated specifications, test suite, executable
    Java program are generated)
 2: information ← agileUMLAnalysis(testSuite, mutants)
 3: initialPopulation(testSuite)
 4: until Stop Criteria do
 5:    Evolutionary Operator(s)
 6:    setObjective(0, selectedSize)
 7:    setObjective(1, faultDetectionRate)
 8:    setObjective(2, APFDMetric)
 9:    fitnessEvaluation(population)
10: end until
11: solutions ← getSolutions()
12: result ← solutionAnaysis(solutions)
```

ity, where the input is OCL specification and the output is the selected (minimised) test cases. The first two lines are the preparation works for the algorithm. When the input specification for the AgileUML tool set is OCL, the mutated OCL specification, test suite and corresponding executable JAVA programs can be generated. The generated test suite is treated as the original one, waiting to be minimised.

In *line 2*, AgileUML will analyse the original test suite and mutants to extract the necessary information, mainly focusing on which test case can detect which mutants. Despite various attempts to make OCL directly executable [205, 206, 207], when system specifications are expressed in OCL, the test suite cannot run directly against these specifications. To tackle this obstacle, we initially produce corresponding JAVA specifications via model-to-text transformation, which are then used to test the suite. It is worth noting that this is just one viable solution we have opted for and is not the exclusive answer.

line 3 - *line 10* detail the key procedures of the TCM algorithms. Beginning with a randomly generated population, then repetitively using evolutionary operators to solve the associated problem. As previously described in *Chapter 6*, in this study, we define three objectives for the TCM process. These objectives are the number of test cases, the ability to detect mutants, and the measure of the original or modified APFD metric. The intent of using the APFD metric as an objective is to optimise the testing ability when multiple test cases share the same detection capability. In this study, we exclusively used the modified APFD metric in this context.

The multi-objective optimisation algorithm will generate a solution set known as the Pareto set. This set contains a group of solutions where none can outperform the others based on all objectives simultaneously. However, we need to select one final solution as the minimisation result. The selection criteria for this solution are based on achieving the maximum fault detection ability with the smallest size. This choice aligns with the objective of the TCM process, which aims to minimise the number of test cases while ensuring a high fault detection rate.

We always give the highest priority to the fault capabilities over the size of the test suite, as the goal of the minimisation process is to enhance the testing procedure without reducing its effectiveness. Consider a scenario in the Pareto optimal set: one test case retains 40% of the original test cases but offers 90% of fault detection ability compared to the original test suite, while another maintains 50% of the test cases with unchanged fault detection capability. In such a scenario, the second test case would be chosen as the minimisation result.

Algorithm 2 demonstrates the general process for the TCP action. The input can either come from the minimised test cases from the TCM process or from the original OCL specification when conducting the TCP process exclusively. The

Algorithm 2 General Test Case Prioritisation Process

```
Input: Selected Test Cases or OCL Specification
Output: Prioritised Test Cases
1: initialPopulation(selectedTestSuite)
2: until stop criteria do
3:     Evolutionary Operator(s)
4:     setObjective(APFDMetric)
5:     fitnessEvaluation(population)
6: end until
7: result ← getSolution()
```

first approach involves merging TCM and TCP processes, whereas the second approach runs the TCP process in isolation. In instances where the input is the OCL specification, some preparation work is needed, similar to the first two lines within *Algorithm 1*. In contrast to TCM, TCP has a single objective, the original or modified APFD metric.

The following details the configurations for each optimisation algorithms. To limit the influence of parameter configurations, the same settings were applied across all four case studies.

The TCP process has five optimisation algorithms, GA, PSO, Firefly, Fish School and Cuckoo Search algorithm. For the GA optimisation algorithm, the crossover rate was also 0.9, and the Partially Mapped Crossover (PMX) was the crossover operator of choice. The mutation rate was defined as$1/TestSize$, and a permutation swap mutation operator was used. When we adapted PSO to the TCP problem in our TeCO framework, the parameter settings were as followings: the initial positions are randomly in [0, 4], and the initial velocities are in [-4, 4]. The value of w is 0.9, c1 is 2, c2 is 2, and α is 0.975. As for the firefly algorithm, we set γ as 1, β_0 as 1 and α as 0.2. The initial light intensity randomly ranged from -5 to 5. Within the fish school algorithm, the minimum weight of each fish is 1, and the maximum weight is 5000. Within the cuckoo search algorithm, we set the number of abandoned nests randomly generated from the range [10, 25].

The TCM process also has five optimisation algorithms, NSGA-II, PSO, Cuckoo Search, MOEA/D and SPEA2. For the NSGA-II algorithm (used for TCM), we adopted a crossover rate of 0.9 with a two-point crossover operator. The mutation rate was set to $1/TestSize$, employing a bit-flip mutation operator. Binary tournament selection was the chosen selection operator. For the PSO algorithm, different from the TCP process, the range of each double value is set as [-1000, 1000]. The cuckoo search algorithm uses the same setting as the TCP process. For the MOEA/D algorithm, we applied the original MOEA/D algorithm that is supported by *JMetal* framework. The default setting for the SPEA2 algorithm is the selection operator is tournament selection, the crossover operator is two points crossover, the crossover possibility is 0.9, the mutation operator is bit flip operator, and the mutation rate is $1/TestSize$.

In our TCO approach, which encompasses both TCM and TCP processes, we do not aim to identify the best combination of these optimisation algorithms. During our evaluation process, we chose to use the NSGA-II algorithm for the TCM process and the GA algorithm for the TCP process due to the solid overall performance delivered by these two algorithms.

All these settings are defaulted within the TeCO framework and may not be optimal for all scenarios. The users can modify these settings according to their own preferences.

During the preliminary phase of this study, an informal combinatorial experiment was conducted in which the population interval was set at 10 and the iteration interval at 100. This led us to the final configuration of 50 for the population size and 1000 for the iterations across all optimisation algorithms. For our chosen case studies, these settings delivered effective outcomes in a reasonable overhead.

However, it is critical to note that this study is not primarily focused on identifying the most optimal configurations for the optimisation algorithm. Hence, the parameters we chose to work with may not be the best for all circumstances. In real-world applications, these variables and the evolutionary operators can be adjusted based on specific needs and requirements.

For the original APFD metric, undetectable defects are not taken into account. In our experimental approach, when we encounter a mutant that cannot be detected, we hypothesise the last test case will identify the mutant, thus making the TF_i equal to the total number of test cases in the test suite.

Given the inherent randomness in both optimisation algorithms, we carried out each experiment 50 times to decrease the deviations. To maintain a consistent experiment environment, all experiments were conducted on the same machine with MacOS 12.3.1, a 2GHz Quad-Core Intel Core i5 processor, and 16GB 3733 MHz LPDDR4X RAM. The algorithms were implemented in the JAVA within IntelliJ Idea IDE.

In the subsequent section, we will use a straightforward system, String Process, as our running example to explain the entire procedure involved in executing the TCO processes. This particular system was chosen for its simplicity and the ease with which it allows us to showcase the various steps and strategies involved in our optimisation processes.

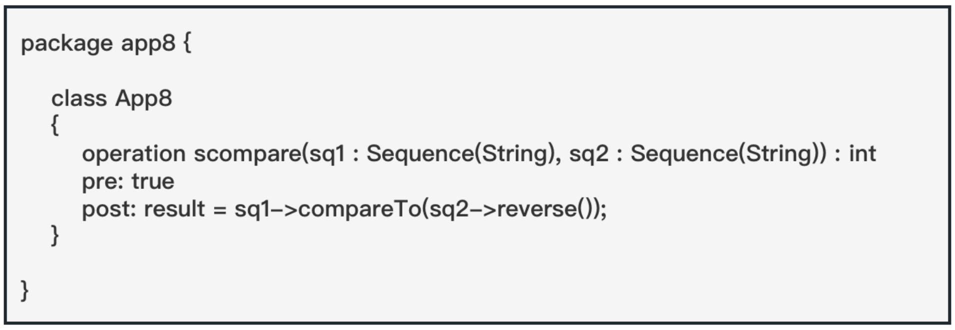
We will demonstrate how the AgileUML has participated in the optimisation process, the form in which tests are initially generated, how the TCM and TCP processes optimise these tests, etc.

Following the running example, the next four sections will present the results and corresponding analysis of the four case studies. These results contain details about the original test suite, the time required for each algorithm, and the overall overhead for the entire optimisation process. In the context of the TCM process, the selection rate and the rate of fault detection loss will be provided. The selection rate measures the percentage of test cases included in the minimised test suite relative to the original, calculated by comparing the count of test cases in both the minimised and the original suites. Meanwhile, fault detection loss evaluates the reduction in the ability to detect faults when comparing the minimised test suite to the original. As for the TCP process, the prioritised test suite will be compared against a randomised approach under the original and modified APFD metrics.

7.3 Running Example: String Process

This section will showcase an illustrative running example of the OCL-based test case optimisation process using a straightforward system named String Process. This system will serve as a practical example to better understand the various steps involved in the optimisation process. The specification for this system is shown in *Figure 7.2*.

This system offers a singular function that conducts a lexicographical comparison between two sequences of *sq1* and reversed *sq2*. If all strings in *sq1* precisely match the corresponding strings in *sq2*, the function will return a value of 0. If, however, a mismatch is found, the function will return either -1 or 1. Specifically, if the string in *sq1* is lexicographically lesser than the corresponding string in *sq2*, the result will be -1. If the contrary is true, the result will be 1.



```
package app8 {

    class App8
    {
        operation scompare(sq1 : Sequence(String), sq2 : Sequence(String)) : int
        pre: true
        post: result = sq1->compareTo(sq2->reverse());
    }

}
```

Figure 7.2: System Specification for String Process

Once the system is loaded within the AgileUML tool, the corresponding UML diagram is presented as *Figure 7.3*.

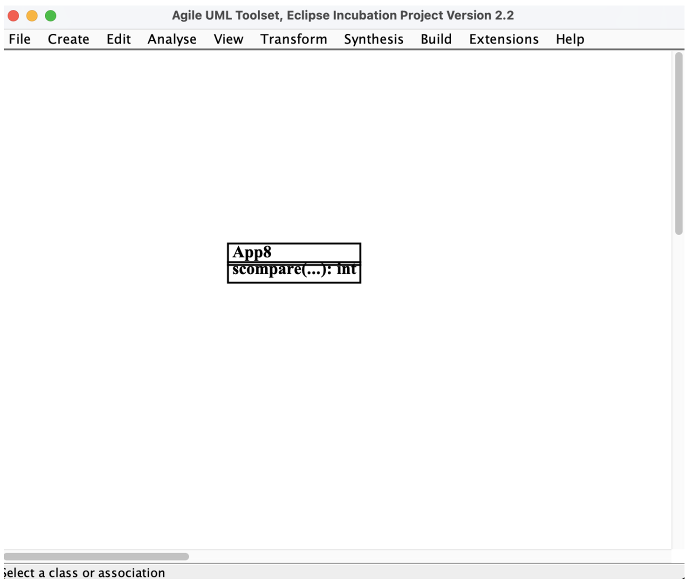

Agile UML Toolset, Eclipse Incubation Project Version 2.2
File
Create
Edit
Analyse
View
Transform
Synthesis
Build
Extensions
Help
App8
scompare(...): int
select a class or association

Figure 7.3: Loaded System

The generated JAVA operation to the original specification is shown as *Figure 7.4*.

Utilising the function provided as *Figure 7.5*, we can generate both mutants and corresponding test cases.

Upon completion of this operation, it is noticeable from the UML diagram, *Figure 7.6*, that two mutants have been created. The original specification *result = sq1->compareTo(sq2)->reverse();*, has been altered to *result = sq1->compareTo(sq2->copy());* and *result = sq2->reverse()-*

```
  public int scompare(List sq1,List sq2)
{   int result = 0;

result = Set.sequenceCompare(sq1,Set.reverse(sq2));
  return result;
}
```

Figure 7.4: JAVA Code to Original Specification

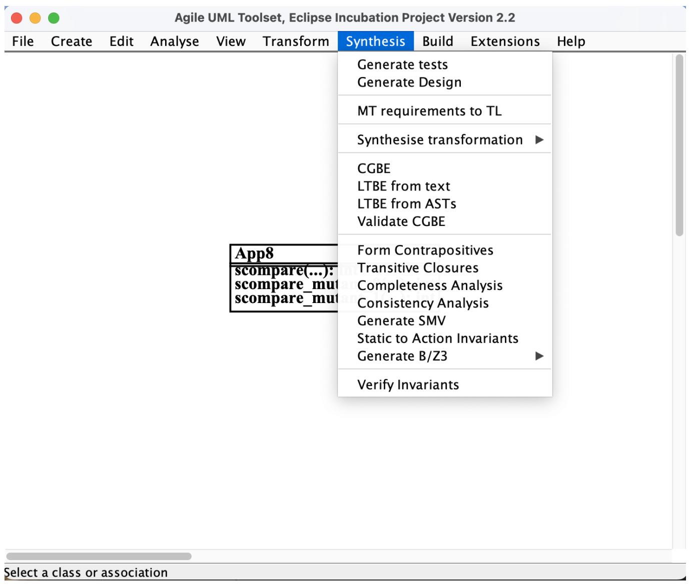


Figure 7.5: AgileUML Function

>compareTo(sq1);.

The corresponding generated JAVA operations to these two mutants are demonstrated in *Figure 7.7*.

Initially, the test suite is run against the original specification to collect the expected behaviour of the operation. Subsequently, it is executed on the mutated specification(s) to obtain the respective outcome(s) and determine if the mutant(s) have been identified.

In the context of OCL, which is based on an object-oriented system, different instances of a class can yield varied results for a specific function, even when the input parameters are the same. Given that the String Process system contains only one class without any attributes, a single class instance, app8x_0, is generated in the input file, as can be seen in *Figure 7.8*.

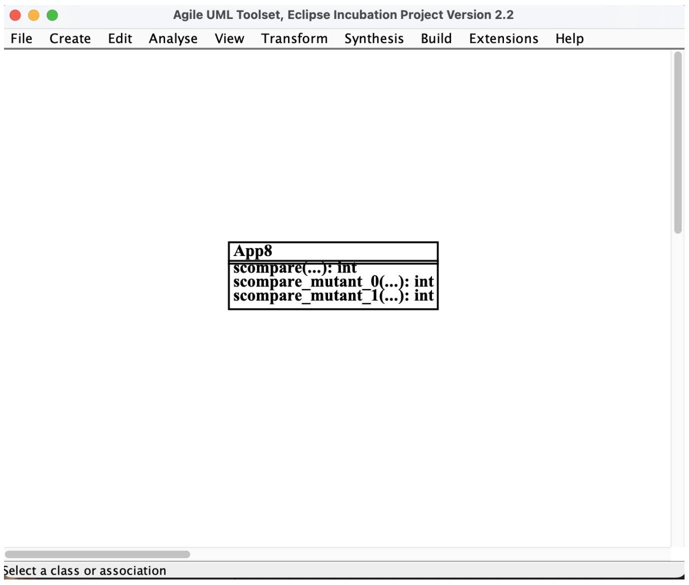

Agile UML Toolset, Eclipse Incubation Project Version 2.2
File Create Edit Analyse View Transform Synthesis Build Extensions Help
App8
scompare(...): int
scompare_mutant_0(...): int
scompare_mutant_1(...): int
Select a class or association

Figure 7.6: Updated UML Diagram

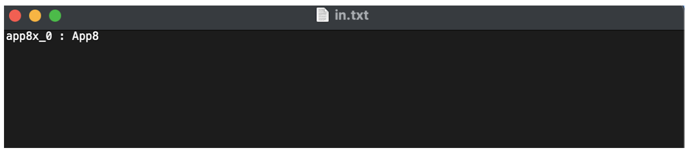
```
  public int scompare_mutant_0(List sq1,List sq2)
{   int result = 0;

result = Set.sequenceCompare(sq1,sq2);
  return result;
}


  public int scompare_mutant_1(List sq1,List sq2)
{   int result = 0;

result = Set.sequenceCompare(Set.reverse(sq2),sq1);
  return result;
}
```

Figure 7.7: JAVA Code to Mutated Specifications



```
app8x_0 : App8
```

Figure 7.8: Class Instance

In discussing the correct or expected system outputs, in some cases, there is an objective correct result to compare against. In particular, there is a known correct result with mathematical functions such as factorial or combinatorial. For code generators such as UML2PY, one check on correctness is that the target code compiles. Otherwise, the output from the original specification may serve as the correct result.

The test cases at the functional level are constructed using the boundary value of each parameter and the various combinations of these parameters. In this example, there are 25 test cases have been constructed. When we have the input file, we can use the *Tests GUI*, *Figure 7.9*, generated by AgileUML to perform mutation testing and collect the corresponding result.

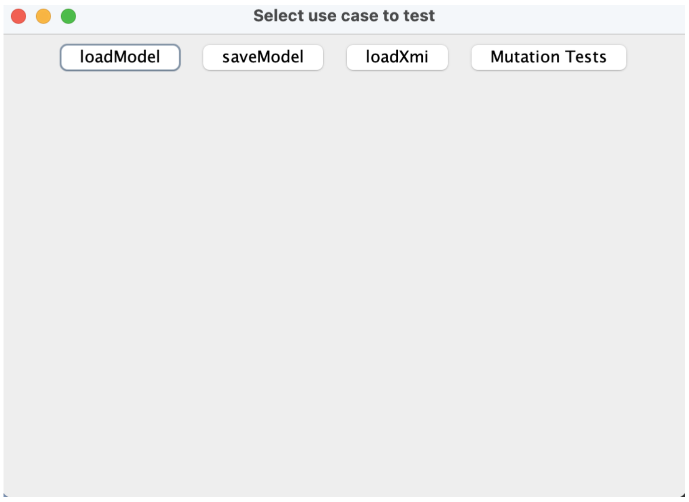
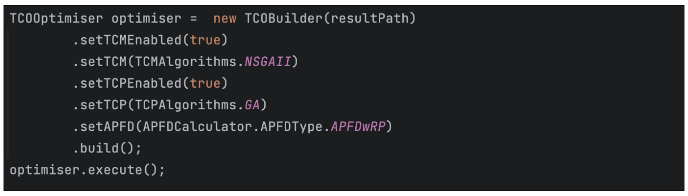


Figure 7.9: Tests GUI

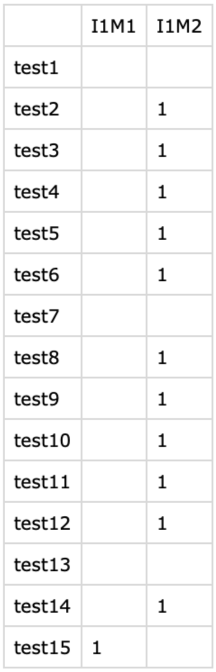
Once we have gathered the mutation testing results, they can be fed into the TeCO framework to execute the TCO process. The configuration part of the framework, as presented in *Figure 7.10*, only requires the location of the result file. It allows users the flexibility to customise the optimisation procedures, allowing them to choose between TCM, TCP, or a combination of both processes, as well as their preferred algorithms. In this instance, as illustrated in the figure, both the TCM and TCP processes are being utilised, with the NSGA-II algorithm assigned to the TCM process and the GA algorithm given to the TCP process. The evaluation metric for this configuration is the modified APFD metric.



```
TCOOptimiser optimiser =  new TCOBuilder(resultPath)
        .setTCMEnabled(true)
        .setTCM(TCMAlgorithms.NSGAII)
        .setTCPEnabled(true)
        .setTCP(TCPAlgorithms.GA)
        .setAPFD(APFDCalculator.APFDType.APFDwRP)
        .build();
optimiser.execute();
```

Figure 7.10: TeCO Configuration

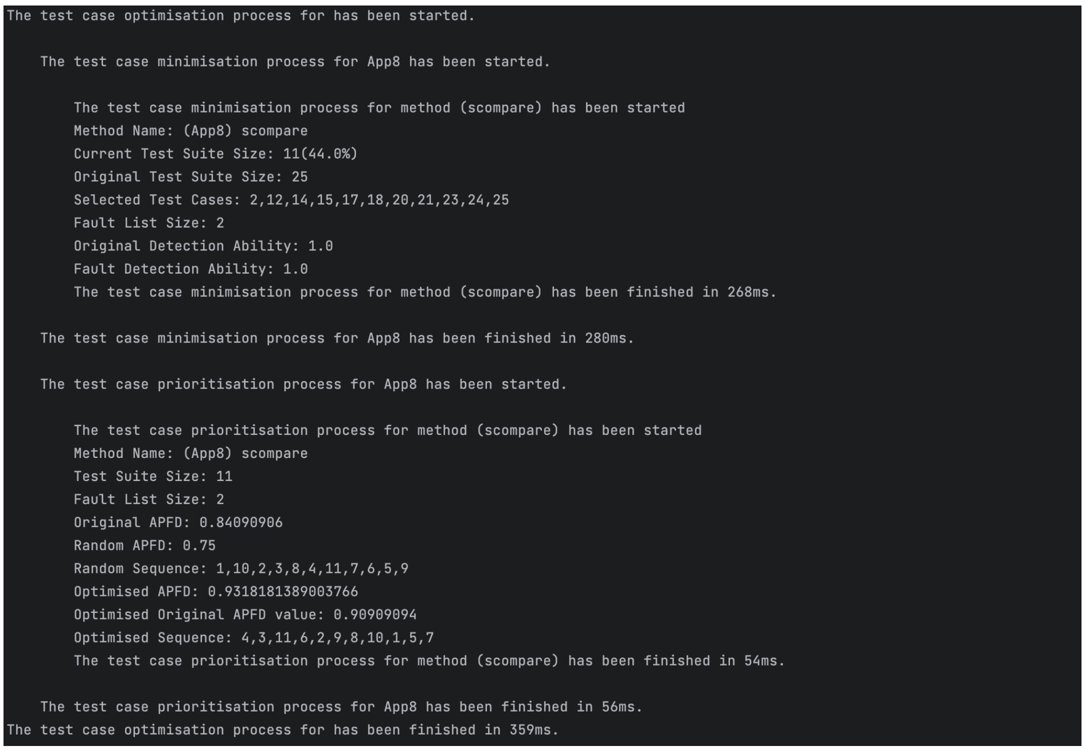
Before the commencement of the optimisation process, a pre-processing step is carried out on the result file. This stage primarily focuses on separating the result file and extracting essential information, such as the correlation between test cases and the mutants they can identify. This data is then documented and stored in the form of CSV file(s), as illustrated in *Figure 7.11* (a fragment).



	I1M1	I1M2
test1		
test2		1
test3		1
test4		1
test5		1
test6		1
test7		
test8		1
test9		1
test10		1
test11		1
test12		1
test13		
test14		1
test15	1	

Figure 7.11: CSV File

Once the pre-processing phase is finished, the TeCO framework will carry out the TCO processes based on the user configuration. The resulting log is displayed as seen in *Figure 7.12*. Alongside the information displayed on the console, relevant data is also archived in accordance with the corresponding result file path.



```
The test case optimisation process for has been started.

    The test case minimisation process for App8 has been started.

        The test case minimisation process for method (scompare) has been started
        Method Name: (App8) scompare
        Current Test Suite Size: 11(44.0%)
        Original Test Suite Size: 25
        Selected Test Cases: 2,12,14,15,17,18,20,21,23,24,25
        Fault List Size: 2
        Original Detection Ability: 1.0
        Fault Detection Ability: 1.0
        The test case minimisation process for method (scompare) has been finished in 268ms.

    The test case minimisation process for App8 has been finished in 280ms.

    The test case prioritisation process for App8 has been started.

        The test case prioritisation process for method (scompare) has been started
        Method Name: (App8) scompare
        Test Suite Size: 11
        Fault List Size: 2
        Original APFD: 0.84090906
        Random APFD: 0.75
        Random Sequence: 1,10,2,3,8,4,11,7,6,5,9
        Optimised APFD: 0.9318181389003766
        Optimised Original APFD value: 0.90909094
        Optimised Sequence: 4,3,11,6,2,9,8,10,1,5,7
        The test case prioritisation process for method (scompare) has been finished in 54ms.

    The test case prioritisation process for App8 has been finished in 56ms.
The test case optimisation process for has been finished in 359ms.
```

Figure 7.12: Test Case Optimisation Result

In regards to the TCM process, the most critical information displayed on the console is the rate of selection along with the corresponding selected test cases. The fault (mutation) detection rate for both the original and minimised test suites will also be demonstrated.

Simultaneously, for the TCP process, the initial APFD value according to the assessment metric is displayed. Details of the prioritised test suite and its corresponding APFD value will be compared to a randomised approach. To compare the differences between the original and modified metrics, we will also present the original APFD value for the prioritised test suite in this instance.

In the following four sections, the experiment results for the corresponding case study will be demonstrated and analysed.

7.4 Case Study 1: Bond

This section provides a comprehensive overview of the experimental results of the first case study, Bond, available in *Appendix B*. Subsequent case studies will follow a similar structure in presentation. *Table 7.2* details the Bond system, including the names of the operations, the size of the faults (mutants), the size of the original test suite (directly generated by AgileUML), the corresponding original and modified APFD values and the fault detection rate. Within the Bond study, these operations are symbolised by codes *A1 - A5* for simplicity.

The concept of faults is tied to the object-oriented nature of OCL, where varying class instances can lead to different outcomes, even with identical input parameters. Therefore, the fault size is determined as a combination of the mutants generated for each operation and the specific instances related to the system specification. For instance, if a specific class has m instances and there are n mutated specifications for a given operation, then the fault size can be determined by multiplying m by n, yielding $m \times n$. For the Bond study, 625 unique instances were constructed, considering the class attributes, then resulting in the fault size is always the multiplication of 625 (like m in the example). Due to the limitation of space, all data will be retained to a maximum of four decimal places. The fault detection rate is presented as a ratio, not in percentage form.

Table 7.2: Bond Case Study Details

	Name	**Fault**	**Test**	**APFD**	$\boldsymbol{APFD_m}$	**Detection**
A1	bisection	1250	64	0.9399	0.9399	1
A2	discount	1250	80	0.9187	0.9187	1
A3	macaulayDuration	1875	4	0.6376	0.6836	0.6853
A4	timeDiscount	1250	80	0.9187	0.9187	1
A5	value	1875	4	0.7389	0.8186	0.8186

Table 7.3: TCM Results for Bond

Name	**A1**	**A2**	**A3**	**A4**	**A5**
NSGA-II					
Selection Rate	43.5937	43.225	50	43.25	50
Detection Loss	0	0	0	0	0
Time (ms)	3600.38	5901.3	6251.02	5786.12	6304.36
PSO					
Selection Rate	41.8437	42.1	50	42.025	50
Detection Loss	0	0	0	0	0
Time (ms)	389064.74	620476.86	558595.36	601616.8	567772.34
Cuckoo Search					
Selection Rate	43.9062	43.65	50	44.3	50
Detection Loss	0	0	0	0	0
Time (ms)	411514.58	664105.18	570286.5	656759.94	604004.28
SPEA2					
Selection Rate	41.7812	41.95	34	42.025	34
Detection Loss	0	0	0	0	0
Time (ms)	189250.94	304152.68	283929	296608.92	238486.88
MOEA/D					
Selection Rate	41.375	41.75	50	41.8	50
Detection Loss	0	0	0	0	0
Time (ms)	4037.62	6348.64	5758.34	6042.38	6061.78

Table 7.3 displays the average outcomes when executing the TCM process exclusively. Within the table, details are provided for each algorithm, including the selection rate, fault detection loss rate, and time consumed in milliseconds. The selection rate and fault detection loss rate are presented in percentages.

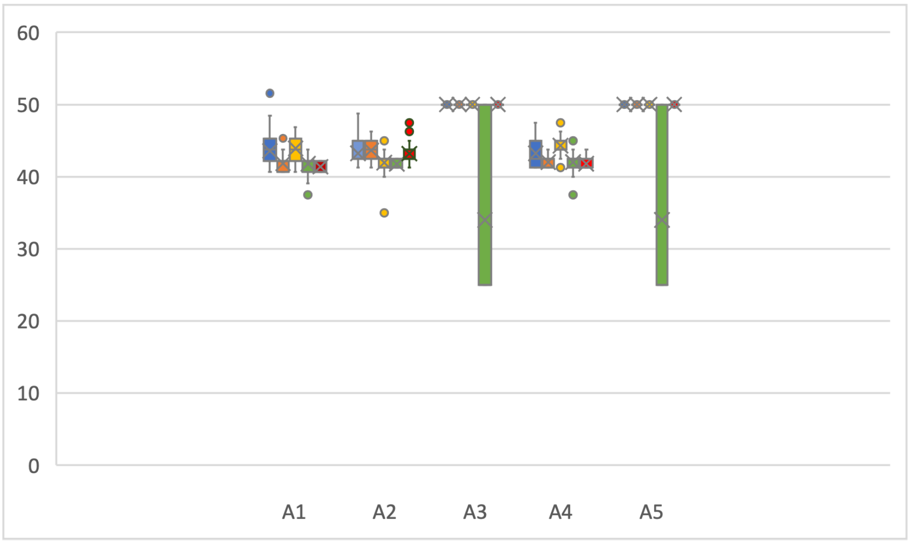


Figure 7.13: Bond: TCM Distribution

It is observable that NSGA-II and MOEA/D are the most compatible algorithms for the TCM process under evaluation for time usage, PSO and Cuckoo Search are the slowest ones, and SPEA2 is in the middle of these groups. *Figure 7.13* displays the distribution of the selection rate for the minimised test suite for each minimisation algorithm. In the chart presented, each group of plots corresponds to a distinct operation within the system specification. For a specific group, five colours can be observed, with each one representing a different minimisation algorithm. These algorithms are sequenced as NSGA-II (blue), PSO (orange), Cuckoo Search (yellow), SPEA2 (green), and MOEA/D (red) for the TCM context. In the remaining TCM distribution figures, the colour scheme remains consistent with what we have used here. However, if the deviation is slight, the colour of the plot might blend into the grey border. Observing this chart, it is noticeable that, except for operations *A3* and *A5* for the SPEA2 algorithm, the minimisation process consistently exhibits a stable performance with only slight deviations across 50 executions.

Table 7.4 details the results of the TCP process for the Bond case study. The table is segmented into five distinct parts, each representing a unique prioritisation algorithm. Within each part, details are provided for both the original and modified APFD metrics. These details include the prioritised results, the corresponding random results for the same metric, and the time consumption for the prioritisation process.

From the prioritisation results table, it can be informally but clearly observed that in the Bond case study, GA consistently performs the best in terms of time usage. The PSO, Firefly, and Cuckoo Search algorithms share a similar level of time consumption, while the Fish School algorithm always takes the longest.

For the specific operation *A5*, the analysis reveals that the optimised test suite has the same evaluation outcome under both the original and modified APFD metrics. This scenario arises when the original test suite is already arranged in an optimal testing sequence, thus leaving no possibility for enhancement. However, such an optimal condition is not always achievable with every generated test suite. Therefore, we conducted further comparisons between the prioritised test suite and a randomly generated approach, which will be demonstrated in the following.

Similarly to the previous figure, *Figure 7.14* and *Figure 7.15* display the distribution of APFD value for the prioritised test suite under the original and modified metric. Much like the earlier distribution chart for the TCM process, each

Table 7.4: TCP Results for Bond

Name	**A1**	**A2**	**A3**	**A4**	**A5**
GA					
$APFD$	0.9886	0.9935	0.6383	0.9937	0.7389
Random	0.8844	0.9491	0.5863	0.9376	0.7050
Time (ms)	2194.48	2140.52	7097.42	2099.76	6136.56
$APFD_m$	0.9947	0.9996	0.6843	0.9996	0.8186
Random	0.8980	0.9479	0.6078	0.9444	0.7587
Time (ms)	2188.06	2169.42	7119.42	2103.9	6158.28
PSO					
$APFD$	0.9886	0.9932	0.6383	0.9937	0.7389
Random	0.9002	0.9393	0.5907	0.9437	0.6742
Time (ms)	139222.88	162822.88	416150.68	159067.42	377595.78
$APFD_m$	0.9946	0.9994	0.6843	0.9996	0.8186
Random	0.8979	0.9480	0.6138	0.9393	0.7207
Time (ms)	220400.08	279079.52	487119.32	261134.08	433921.2
Firefly					
$APFD$	0.9881	0.9930	0.6383	0.9937	0.7389
Random	0.8934	0.9460	0.6043	0.9621	0.7022
Time (ms)	161535.12	175901.16	371644.08	168485.08	333263.42
$APFD_m$	0.9944	0.9996	0.6843	0.9994	0.8186
Random	0.8973	0.9463	0.6279	0.9437	0.7531
Time (ms)	159063.8	179948.38	372078.62	167928.7	324446.2
Fish School					
$APFD$	0.9886	0.9930	0.6383	0.9937	0.7389
Random	0.8864	0.9360	0.5968	0.9365	0.6869
Time (ms)	1130607.32	1052639.54	2738217.84	1043040.8	2337481.4
$APFD_m$	0.9947	0.9994	0.6843	0.9996	0.8186
Random	0.8919	0.9453	0.6121	0.9402	0.7151
Time (ms)	1272115.84	1247468.78	3443070.7	1237350.02	2901368.06
Cuckoo Search					
$APFD$	0.9886	0.9932	0.6383	0.9937	0.7389
Random	0.9044	0.9373	0.5938	0.9555	0.6835
Time (ms)	219088.6	271409.22	476489.88	255396.94	420470.66
$APFD_m$	0.9946	0.9996	0.6843	0.9994	0.8186
Random	0.8979	0.9480	0.6138	0.9393	0.7207
Time (ms)	220398.2	279083.04	487121.74	261137.52	433923.96

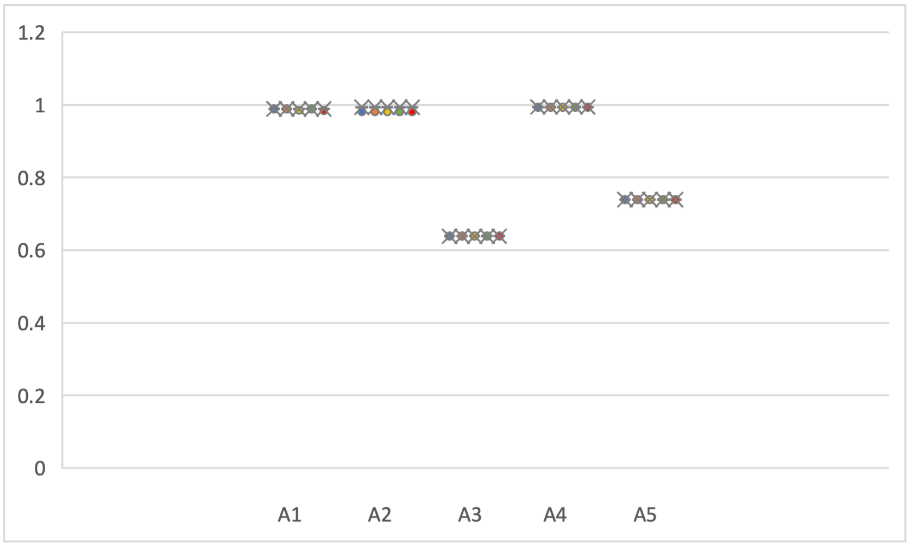


Figure 7.14: Bond: TCP Distribution under Original APFD Metric

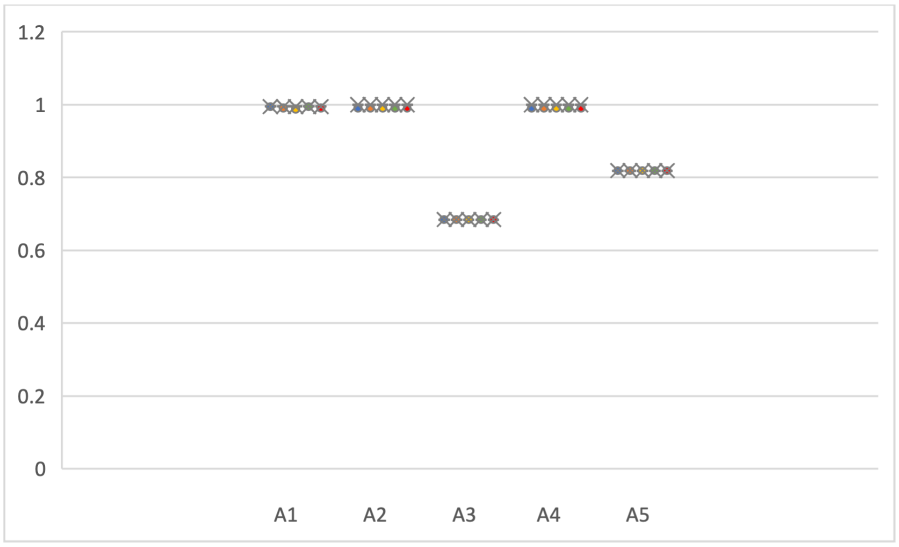


Figure 7.15: Bond: TCP Distribution under Modified APFD Metric

Table 7.5: Bond: Comparison between Random Approach (p-value)

Name	A1	A2	A3	A4	A5
GA					
$APFD$	2.7277e-19	1.5166e-16	6.8667e-16	7.2112e-17	1.4235e-08
$APFD_m$	1.3585e-19	7.9893e-17	6.3002e-17	9.7413e-14	1.1706e-20
PSO					
$APFD$	3.3081e-20	1.1621e-16	6.5652e-15	2.3051e-16	2.5726e-09
$APFD_m$	2.3433e-19	3.0667e-17	6.0886e-17	2.1413e-16	2.0421e-20
Firefly					
$APFD$	1.1888e-18	6.5921e-15	1.9553e-14	2.8077e-14	1.4137e-08
$APFD_m$	6.0254e-19	1.0714e-18	5.9916e-17	4.3793e-16	1.1823e-20
Fish School					
$APFD$	3.3071e-20	3.6648e-17	2.0076e-14	1.9882e-17	1.6966e-07
$APFD_m$	3.3101e-20	2.1194e-18	2.0361e-14	6.2622e-17	2.5216e-11
Cuckoo Search					
$APFD$	4.4468e-19	3.2102e-17	6.6073e-15	4.7601e-14	8.0996e-07
$APFD_m$	2.3433e-19	1.8155e-17	6.0886e-17	3.6391e-16	2.0421e-20

group of plots corresponds to a specific operation, and each colour represents a distinct prioritisation algorithm. However, the sequence of these algorithms within each group has shifted to GA (blue), PSO (orange), Firefly (yellow), Fish School (green), and Cuckoo Search (red). In the remaining TCP distribution figures, the same as the TCM figures, the colour scheme remains consistent with what we have used here. If the deviation is minor, the colour of each plot might be blurred by the grey border. From the figures, the results demonstrate all algorithms are stable during the prioritisation process under the corresponding APFD metric.

We performed the non-parametric Wilcoxon test on the original and modified APFD metrics to verify a significance level of 5%, with the null hypothesis that the observed differences between the prioritised test suite and random approach are not statistically significant. The results are shown in *Table 7.5*. The results show that the proposed prioritisation algorithms always perform better than the random approach.

Table 7.6 combines the TCM (NSGA-II) and TCP (GA)

Table 7.6: TCO Results for Bond

Name	**A1**	**A2**	**A3**	**A4**	**A5**
Selection Rate	43.5937	43.2	50	42.975	50
Detection Loss	0	0	0	0	0
$APFD$	0.9645	0.9843	0.5913	0.9836	0.6593
$APFD_m$	0.9781	0.9986	0.6833	0.9979	0.8186
Random	0.8034	0.9016	0.6714	0.9042	0.6807
Time (ms)	2037.2	1940.76	5045.76	1916.94	4506.9
Overhead	42270.96 ms				

processes together and includes the average selection rate, fault detection loss rate, the original APFD value, modified APFD value for the optimised test suite and the modified APFD value for the random approach. For the TCO process, we initially applied the TCM process on the original test suite, followed by the TCP process. We collected the time expense for the TCP process on the minimised test suite to validate whether the TCM process could benefit the TCP process in terms of time usage. The *Overhead* line captures the average total overhead, encompassing time allocations for generating test cases and mutants, processing fault detection information, and the operational time for the NSGA-II and GA for all operations within the corresponding case study.

In order to check whether the TCM process could benefit the TCP process in terms of time usage, we also conduct the non-parametric Wilcoxon test on the time usage for the TCP process, and the results are shown in *Table 7.7.* The results illustrate that within this case study, the prioritisation time for minimised test suites is always less than the original one.

It is reasonable to deliver these results since the TCM process will decrease the search space of the TCP process, enhancing overall efficiency.

We utilised five different algorithms for both the TCM and TCP processes with the intent of identifying which one

Table 7.7: Bond: Comparison of TCP Time after Minimisation (p-value)

Name	**p-value**
A1	1.3894e-13
A2	2.6998e-15
A3	7.0581e-18
A4	2.4533e-13
A5	7.0644e-18

Table 7.8: Bond: Rank for TCM Algorithms

Name	**A1**	**A2**	**A3**	**A4**	**A5**
NSGA-II	b	b	b	b	b
PSO	a	a	b	a	b
Cuckoo Search	b	b	b	c	b
SPEA2	a	a	a	a	a
MOEA/D	a	a	b	a	b

performed best in terms of effectiveness. To determine this, we conducted pairwise comparisons following Kruskal-Wallis and Dunn's test. The results are shown in *Table 7.8 - 7.10*, with significant differences between the approaches denoted by different letters within the tables. Specifically, *Table 7.8* represents the results for the TCM process, *Table 7.9* displays the results for the TCP process under the original APFD metric, and *Table 7.10* illustrates the results for the TCP process employing the modified APFD metric.

In the comparative analysis of optimisation algorithms across four case studies, the letters found in the result tables categorise the algorithms by their effectiveness. Distinct letters between two methods signify a significant difference in their effectiveness (with a p-value < 0.05). Conversely, when two methods are assigned the same letter(s), it indicates that the difference in their performances is not statistically significant. Algorithms associated with letters closer to the beginning of the alphabet exhibit superior performance.

From the information in *Table 7.8*, for this case study, the

Table 7.9: Bond: Rank for TCP Algorithms under Original APFD

Name	**A1**	**A2**	**A3**	**A4**	**A5**
GA	a	a	a	a	a
PSO	a	a	a	a	a
Firefly	b	a	a	a	a
Fish School	a	a	a	a	a
Cuckoo Search	a	a	a	a	a

PSO, SPEA2, and MOEA/D algorithms outperform both NSGA-II and Cuckoo Search. When considering time consumption as a factor, both NSGA-II and MOEA/D are more efficient than the other algorithms. Consequently, in the context of this case study, MOEA/D appears to be the most fitting algorithm for the minimisation process.

Table 7.10: Bond: Rank for TCP Algorithms under Modified APFD

Name	**A1**	**A2**	**A3**	**A4**	**A5**
GA	a	a	a	a	a
PSO	a	a	a	a	a
Firefly	a	a	a	a	a
Fish School	b	a	a	a	a
Cuckoo Search	a	a	a	a	a

From *Table 7.9* and *Table 7.10*, we observe that in most cases, these prioritisation algorithms do not exhibit statistically significant differences under both original and modified APFD metrics. However, examining the previous table reveals that there are noticeable variations in time, with GA being identified as the fastest algorithm.

7.5 Case Study 2: Interest Rate

Our second case study is called Interest Rate, and the detailed system specification is in *Appendix C*. *Table 7.11* details this system. Within the Interest Rate study, these operations are symbolised by codes *B1* - *B2* for simplicity. For

Table 7.11: Interest Rate Case Study Details

	Name	**Fault**	**Test**	**APFD**	$\boldsymbol{APFD_m}$	**Detection**
B1	nelsonseigal	9	243	0.9727	0.9727	1
B2	ns	3	243	0.4190	0.4183	0.6666

Table 7.12: TCM Results for Interest Rate

Name	**B1**	**B2**
NSGA-II		
Selection Rate	43.7860	42.4279
Detection Loss	0	0
Time (ms)	146.44	141.52
PSO		
Selection Rate	40.3292	40.8312
Detection Loss	0	0
Time (ms)	5656.34	5839.9
Cuckoo Search		
Selection Rate	40.5185	42.9053
Detection Loss	0	0
Time (ms)	19261.5	18769.02
SPEA2		
Selection Rate	44.5102	36.8971
Detection Loss	0	0
Time (ms)	5670.82	4044.08
MOEA/D		
Selection Rate	43.5720	43.4074
Detection Loss	0	0
Time (ms)	150.2	140.9

this case study, there only one class instance is constructed.

Similarly, *Table 7.12* displays the average outcomes when only executing the TCM process for this case study. The selection rate and fault detection loss rate are presented in percentages, and the time consumption is measured in milliseconds.

From the examination of the result table, the situation is akin to what was observed in the Bond case study. NSGA-II and MOEA/D appear as the most time-efficient algorithms for the TCM process under evaluation. In contrast, the Cuckoo Search algorithm became the slowest in this context.

Meanwhile, PSO and SPEA2 shared a similar level of time consumption.

Figure 7.16 displays the distribution of the selection rate for the minimised test suite for each minimisation algorithm across 50 executions. The distribution chart has two groups of plots because this case study encompasses only two operations. For the TCM process across all four case studies, each colour within the respective group corresponds to a distinct minimisation algorithm, ordered as NSGA-II, PSO, Cuckoo Search, SPEA2, and MOEA/D. Generally, the minimisation process shows a steady performance, marked by only minor fluctuations.

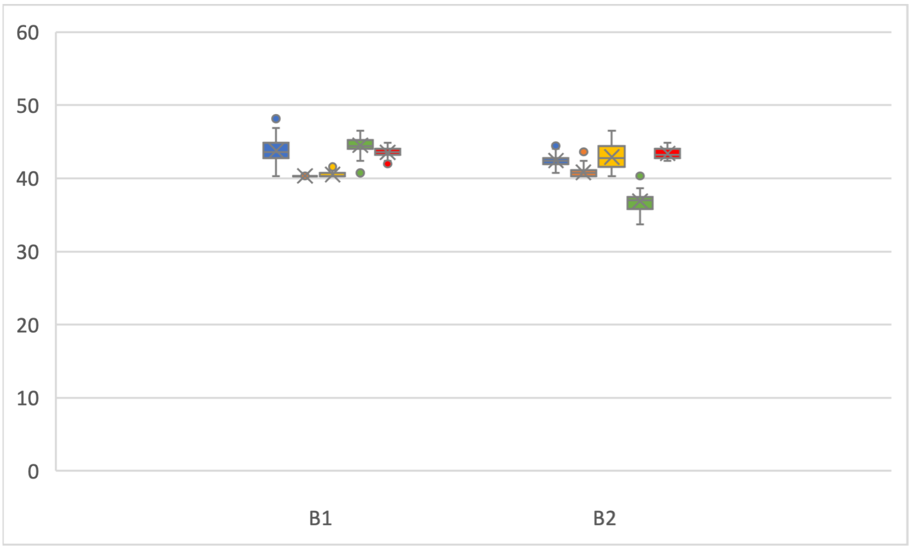


Figure 7.16: Interest Rate: TCM Distribution

Table 7.13 details the results of the TCP process for the Interest Rate case study.

In examining the prioritisation results from the Interest Rate case study, the GA algorithm clearly leads the group, consistently delivering the fastest performance in terms of time usage. The Fish School algorithm always takes the longest time to complete the tasks. Positioned between these two extremes, the Firefly and Cuckoo Search algorithms demonstrate similar time consumption levels, while the PSO

Table 7.13: TCP Results for Interest Rate

Name	**B1**	**B2**
GA		
$APFD$	0.9974	0.6659
Random	0.9794	0.6061
Time (ms)	77.44	169.3
$APFD_m$	0.9993	0.6666
Random	0.9820	0.6111
Time (ms)	66.98	81.94
PSO		
$APFD$	0.9974	0.6659
Random	0.9795	0.6065
Time (ms)	14648.9	14872.48
$APFD_m$	0.9993	0.6666
Random	0.9828	0.6076
Time (ms)	14527.96	14788.58
Firefly		
$APFD$	0.9974	0.6659
Random	0.9775	0.6007
Time (ms)	179196.32	179719.54
$APFD_m$	0.9993	0.6666
Random	0.9818	0.6070
Time (ms)	176726.8	177014.84
Fish School		
$APFD$	0.9974	0.6659
Random	0.9810	0.6052
Time (ms)	1828569.78	1740017.34
$APFD_m$	0.9993	0.6666
Random	0.9807	0.6028
Time (ms)	1926360.58	1814902
Cuckoo Search		
$APFD$	0.9974	0.6659
Random	0.9812	0.5939
Time (ms)	230349.88	231581.74
$APFD_m$	0.9993	0.6666
Random	0.9850	0.6066
Time (ms)	232736.06	233680.7

operates at a faster pace.

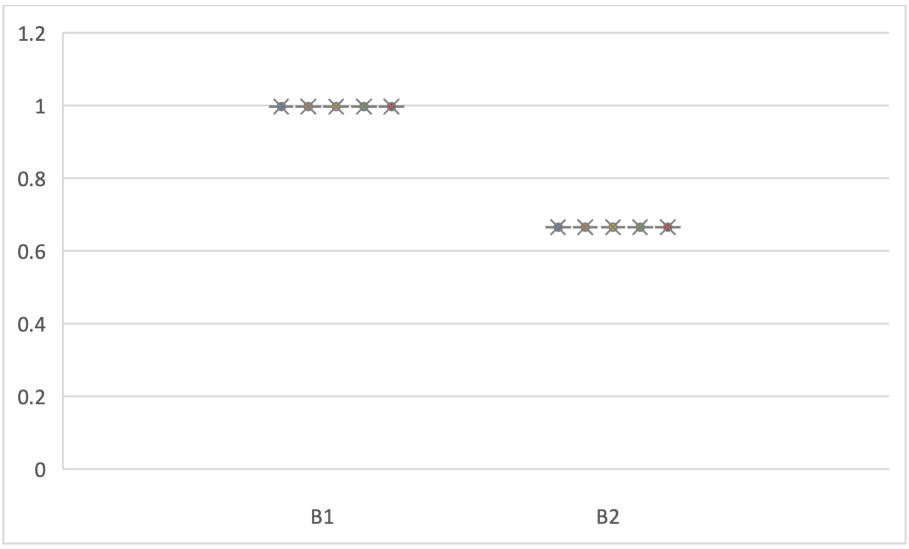


Figure 7.17: Interest Rate: TCP Distribution under Original APFD Metric

Figure 7.17 and *Figure 7.18* display the distribution of APFD value for the prioritised test suite under the original and modified metric. For the TCP process in all case studies except the fourth, each colour within the designated group represents a specific prioritisation algorithm sequenced as GA, PSO, Firefly, Fish School, and Cuckoo Search. Based on the figures, the results reveal consistent stability across all algorithms during the prioritisation process, as evaluated under the corresponding APFD metric.

We also performed the non-parametric Wilcoxon test on the original and modified APFD metrics to verify a significance level of 5%, with the null hypothesis that the observed differences between the prioritised test suite and random approach are not statistically significant for this case study. The results are shown in *Table 7.14*. The results show that the proposed prioritisation algorithms always perform better than the random approach.

Table 7.15 shows the collected results when we combine the TCM and TCP process jointly.

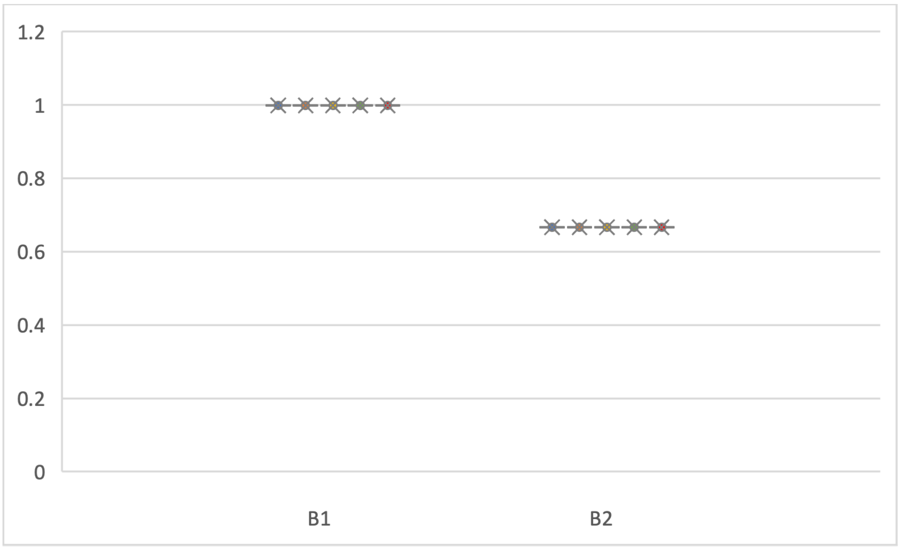


Figure 7.18: Interest Rate: TCP Distribution under Modified APFD Metric

Table 7.14: Interest Rate: Comparison between Random Approach (p-value)

Name	**B1**	**B2**
GA		
$APFD$	6.6082e-20	5.9347e-18
$APFD_m$	3.3003e-20	1.6842e-18
PSO		
$APFD$	1.2546e-19	6.9008e-17
$APFD_m$	3.3023e-20	1.6759e-18
Firefly		
$APFD$	1.2571e-19	3.2877e-20
$APFD_m$	3.3052e-20	1.6819e-18
Fish School		
$APFD$	1.2535e-19	4.6465e-19
$APFD_m$	3.2945e-20	1.6800e-18
Cuckoo Search		
$APFD$	6.0962e-20	4.6517e-19
$APFD_m$	3.3033e-20	4.6231e-19

Table 7.15: TCO Results for Interest Rate

Name	**B1**	**B2**
Selection Rate	44.0823	42.6666
Detection Loss	0	0
$APFD$	0.9942	0.6650
$APFD_m$	0.9984	0.6666
Random	0.9566	0.5304
Time (ms)	26.64	28.26
Overhead	327.88 ms	

Table 7.16: Interest Rate: Comparison of TCP Time after Minimisation (p-value)

Name	**p-value**
B1	5.5686e-16
B2	1.6016e-16

Table 7.17: Interest Rate: Rank for TCM Algorithms

Name	**B1**	**B2**
NSGA-II	b	c
PSO	a	b
Cuckoo Search	a	cd
SPEA2	b	a
MOEA/D	b	d

To investigate if the TCM process could lead to time-saving benefits for the TCP process, we conducted a non-parametric Wilcoxon test on the time usage for the TCP process. The results, displayed in *Table 7.16*, show that in this case study, the prioritisation time for minimised test suites is consistently less than that for the original ones.

In order to compare or rank these TCM and TCP algorithms, we conducted pairwise comparisons following Kruskal-Wallis and Dunn's tests. The results are shown in *Table 7.17 - 7.19*, with significant differences between the approaches denoted by different letters within the tables.

From *Table 7.17*, it becomes apparent that definitively determining the best algorithm for the TCM process is a complex task. For instance, while the Cuckoo Search algorithm exhibits outstanding performance in *B1*, its efficacy reduces in *B2*, leading to a less favourable outcome. An approach can have more than one letter. As an example, looking at the results for *B2* operation, we can tell that Cuckoo Search is not different from MOEA/D, and it is also not different from NSGA-II, even though NSGA-II is different from MOEA/D.

Upon reviewing *Table 7.18* and *Table 7.19*, and combining

Table 7.18: Interest Rate: Rank for TCP Algorithms under Original APFD

Name	B1	B2
GA	a	a
PSO	a	a
Firefly	a	a
Fish School	a	a
Cuckoo Search	a	a

Table 7.19: Interest Rate: Rank for TCP Algorithms under Modified APFD

Name	B1	B2
GA	a	a
PSO	a	a
Firefly	a	a
Fish School	a	a
Cuckoo Search	a	a

the findings with the results from the Bond case study, we observed that in most scenarios, the five prioritisation algorithms do not show statistically significant differences under both the original and modified APFD evaluation metrics. As a result, in the subsequent two case studies, we will refrain from reporting the rankings between these algorithms unless a significant difference is noted for the specific operation.

7.6 Case Study 3: MathLib

Our third case study is called MathLib, and the detailed system specification is in *Appendix D. Table 7.20* details this system. Within the Interest Rate study, these operations are symbolised by codes *C1 - C15* for simplicity. For this case study, there only one class instance is constructed. More details about the system are available in [14].

Similarly, *Table 7.21 - 7.23* demonstrates the average results for this case study when only the TCM process is implemented. The metrics displayed include the selection rate and the fault detection loss rate, both represented as percent-

Table 7.20: MathLib Case Study Details

	Name	**Fault**	**Test**	**APFD**	$APFD_m$	**Detection**
C1	acosh	3	3	0.6111	0.7222	1
C2	asinh	4	5	0.9	1	1
C3	atanh	4	5	0.6	0.6	0.75
C4	bitwiseAnd	8	25	0.32	0.315	0.75
C5	bitwiseNot	3	5	0.6333	0.6333	1
C6	bytes2integer	6	7	0.2142	0.1785	0.5
C7	combinatorial	5	20	0.155	0.14	0.4
C8	decimal2binary	2	5	0.5	0.5	1
C9	decimal2bits	2	5	0.8	0.85	1
C10	decimal2hex	2	5	0.9	1	1
C11	decimal2oct	2	5	0.8	0.85	1
C12	decimal2octal	2	5	0.9	1	1
C13	integer2bytes	3	5	0.4333	0.4666	1
C14	modInverse	5	20	0.345	0.34	0.8
C15	modPow	4	100	0.795	0.795	1

ages, along with the time consumption, which is measured in milliseconds.

Analysing the results table shows similarities to observations made in earlier case studies. The NSGA-II and MOEA/D algorithms consistently stand out as the most time-effective options for the TCM process. The Cuckoo Search algorithm tends to be the slowest, with the exception of the *C5* operation. A particularly interesting finding in this study is the significant variation in time usage by the SPEA2 algorithm across different operations.

Figure 7.19 - 7.21 display the distribution of the selection rate for the minimised test suite for each minimisation algorithm across 50 executions.

From the presented figures, it can be observed that the performances of the PSO and MOEA/D algorithms are consistently stable across all 15 operations. The Cuckoo Search algorithm exhibits occasional slight instability for the operations *C4* and *C7*. NSGA-II generally maintains stability but exhibits exceptions in *C4*, *C6*, *C7*, and *C14*. Interestingly, the

Table 7.21: TCM Results for MathLib (1)

Name	C1	C2	C3	C4	C5
NSGA-II					
Selection Rate	33.3333	60	60	34.8	20
Detection Loss	0	0	0	0	0
Time (ms)	17.42	13.98	15.1	21.82	14.9
PSO					
Selection Rate	66.6666	60	60	44	60
Detection Loss	0	0	0	0	0
Time (ms)	287.48	320.78	306.86	722.54	297.64
Cuckoo Search					
Selection Rate	66.6666	60	60	45.76	60
Detection Loss	0	0	0	0	0
Time (ms)	431.7	598.58	592.9	2238.58	568.26
SPEA2					
Selection Rate	33.3333	33.6	35.6	25.52	20
Detection Loss	0	0	0	0	0
Time (ms)	23.24	6.76	15.12	1668.8	1945.58
MOEA/D					
Selection Rate	66.6666	60	60	44	20
Detection Loss	0	0	0	0	0
Time (ms)	23.4	21.5	19.44	31.24	20.16

Table 7.22: TCM Results for MathLib (2)

Name	C6	C7	C8	C9	C10
NSGA-II					
Selection Rate	33.7142	33	60	60	60
Detection Loss	0	0	0	0	0
Time (ms)	14.72	19.28	14.3	13.18	13.86
PSO					
Selection Rate	42.8571	45	60	60	60
Detection Loss	0	0	0	0	0
Time (ms)	348.6	566.16	250.7	266.24	263.42
Cuckoo Search					
Selection Rate	43.4285	47	60	60	60
Detection Loss	0	0	0	0	0
Time (ms)	721.24	1692.16	536.34	539.54	535.94
SPEA2					
Selection Rate	14.2857	24.8	31.6	34	34
Detection Loss	0	0	0	0	0
Time (ms)	1297	1466.94	408.58	9.7	7.12
MOEA/D					
Selection Rate	42.8571	45	60	60	60
Detection Loss	0	0	0	0	0
Time (ms)	20.88	27.7	18.64	20.34	19.42

Table 7.23: TCM Results for MathLib(3)

Name	C11	C12	C13	C14	C15
NSGA-II					
Selection Rate	60	60	60	31.7	41.84
Detection Loss	0	0	0	0	0
Time (ms)	13.52	14.38	13.9	18.62	42.4
PSO					
Selection Rate	60	60	60	45	41
Detection Loss	0	0	0	0	0
Time (ms)	264.92	270.04	269.08	558.78	2114.44
Cuckoo Search					
Selection Rate	60	60	60	45	41.54
Detection Loss	0	0	0	0	0
Time (ms)	533.62	529.14	534.22	1718.08	7938.42
SPEA2					
Selection Rate	32.4	36.8	32.4	18.2	42.66
Detection Loss	0	0	0	0	0
Time (ms)	8.96	6.14	20.76	1383.72	2714.42
MOEA/D					
Selection Rate	60	60	60	45	42.1
Detection Loss	0	0	0	0	0
Time (ms)	19.22	20.72	19.16	27.18	62.26

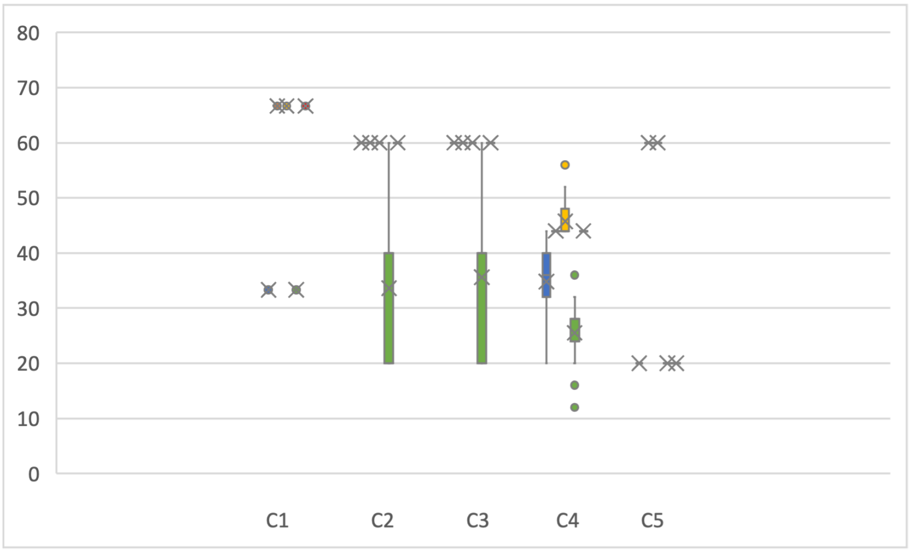


Figure 7.19: MathLib: TCM Distribution (1)

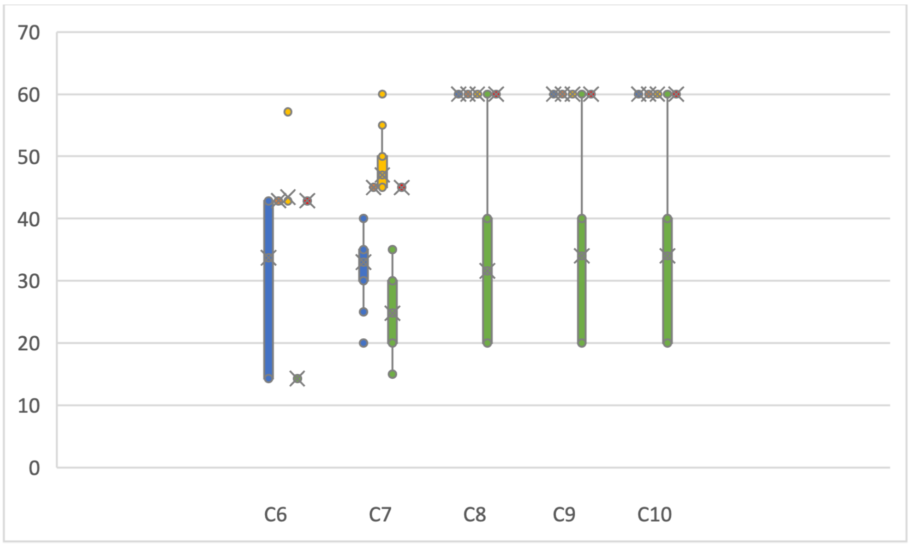


Figure 7.20: MathLib: TCM Distribution (2)

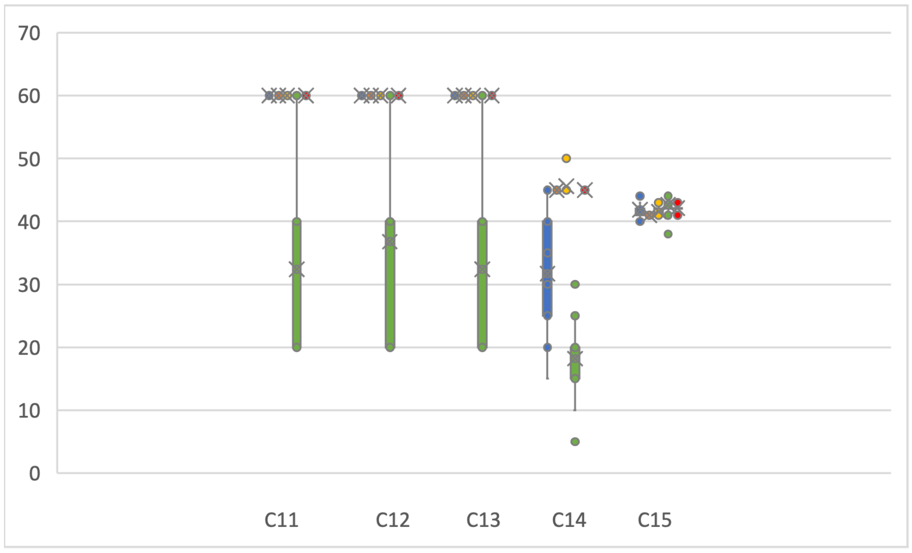


Figure 7.21: MathLib: TCM Distribution (3)

SPEA2 algorithm is consistently unstable throughout this case study. Despite efforts to identify the cause, the underlying reason for this phenomenon remains unattainable. We remain hopeful that future research may shed light on this phenomenon, leaving an open question that may be interesting for future research.

Table 7.24 - 7.26 detail the results of the TCP process for the MathLib case study.

In the MathLib case study, as reflected in the results tables and consistent with earlier observations, the GA algorithm consistently outperforms all other algorithms in terms of time usage. Following GA, the ranking in time usage becomes apparent, with PSO coming next, then Firefly, followed by Cuckoo Search. The Fish School algorithm is once again found to be the slowest, requiring the most time to complete the prioritisation task.

In the data presented within the tables, we observe that certain prioritised test suites achieve a modified APFD value of 1, meaning that the initial test case identifies all of the defects or mutants within the system. This occurrence is not unusual in this specific case study, given that the system being modelled is largely centred on mathematical computations. When mutants are constructed for these pure mathematical operators, like altering addition to subtraction, such mutants tend to be easily killed.

Similar to *A5*, both *C10* and *C12* already have an optimal sequence within the original test suite, meaning that the prioritised test suites do not show any improvement under the evaluation metric. However, achieving such an optimal condition within the original test suite is not always possible. Consequently, we conducted further comparisons between the prioritised test suite and a random approach, and the com-

Table 7.24: TCP Results for MathLib(1)

Name	C1	C2	C3	C4	C5
GA					
$APFD$	0.8333	0.9	0.6990	0.7389	0.9
Random	0.66	0.9	0.6040	0.4745	0.8213
Time (ms)	5.12	3.48	3.82	7.84	3.52
$APFD_m$	1	1	0.7470	0.7497	1
Random	0.8277	1	0.6185	0.4882	0.8806
Time (ms)	5.32	3.78	4.24	10.56	4.4
PSO					
$APFD$	0.8333	0.9	0.6980	0.7388	0.9
Random	0.6488	0.9	0.5950	0.475	0.8280
Time (ms)	41.76	61.58	65.34	527.06	59.2
$APFD_m$	1	1	0.7470	0.7494	1
Random	0.7588	1	0.6245	0.4643	0.9026
Time (ms)	41.36	60.02	64.22	531.04	58.98
Firefly					
$APFD$	0.8333	0.9	0.6970	0.738	0.9
Random	0.6577	0.9	0.6060	0.4742	0.8453
Time (ms)	52.06	79.56	82.6	719.78	69.56
$APFD_m$	1	1	0.7455	0.7491	1
Random	0.7544	1	0.6050	0.4307	0.9206
Time (ms)	50.94	80.3	84.1	740.74	69.64
Fish School					
$APFD$	0.8333	0.9	0.6980	0.7388	0.9
Random	0.6377	0.9	0.6030	0.4712	0.8146
Time (ms)	283.68	400.4	415.14	3166.18	310.88
$APFD_m$	1	1	0.7440	0.7497	1
Random	0.7833	1	0.6135	0.442	0.8886
Time (ms)	283.84	399.72	411	3123.2	311.74
Cuckoo Search					
$APFD$	0.8333	0.9	0.6990	0.7394	0.9
Random	0.6311	0.9	0.6060	0.4538	0.8080
Time (ms)	162.9	247.62	260.32	1643.04	237.82
$APFD_m$	1	1	0.7455	0.7497	1
Random	0.7877	1	0.6255	0.4838	0.8973
Time (ms)	166.42	267.76	274.98	1772.16	265.08

Table 7.25: TCP Results for MathLib(2)

Name	**C6**	**C7**	**C8**	**C9**	**C10**
GA					
$APFD$	0.4985	0.393	0.9	0.9	0.9
Random	0.4085	0.2254	0.724	0.882	0.476
Time (ms)	4	6.56	3.26	3.26	3.38
$APFD_m$	0.4978	0.3835	1	1	1
Random	0.4042	0.1901	0.836	0.97	0.442
Time (ms)	5.22	8.56	3.98	4.1	4.04
PSO					
$APFD$	0.4971	0.395	0.9	0.9	0.9
Random	0.4271	0.2190	0.716	0.876	0.524
Time (ms)	96.54	385.86	59.98	57.12	56.16
$APFD_m$	0.4957	0.385	1	1	1
Random	0.4307	0.2145	0.814	0.97	0.512
Time (ms)	99.08	383.98	57.66	57.16	58.98
Firefly					
$APFD$	0.4999	0.3838	0.9	0.9	0.9
Random	0.4271	0.2092	0.762	0.878	0.464
Time (ms)	110.32	481.36	69.14	66.98	67.78
$APFD_m$	0.4957	0.3725	1	1	1
Random	0.4021	0.2132	0.8340	0.964	0.528
Time (ms)	112.84	497.1	70.96	68.14	69.78
Fish School					
$APFD$	0.4957	0.395	0.9	0.9	0.9
Random	0.4214	0.2102	0.73	0.878	0.476
Time (ms)	502.44	2030.54	317.88	318.34	317.82
$APFD_m$	0.4957	0.385	1	1	1
Random	0.4149	0.2039	0.775	0.973	0.49
Time (ms)	511.58	1969.44	314.64	318.52	310
Cuckoo Search					
$APFD$	0.4985	0.395	0.9	0.9	0.9
Random	0.4242	0.2048	0.738	0.878	0.504
Time (ms)	345.38	1135.16	234.44	226.56	233.14
$APFD_m$	0.4978	0.385	1	1	1
Random	0.4042	0.1979	0.778	0.97	0.582
Time (ms)	392.9	1329.3	269.22	265.16	263.8

Table 7.26: TCP Results for MathLib(3)

Name	**C11**	**C12**	**C13**	**C14**	**C15**
GA					
$APFD$	0.9	0.9	0.7	0.7817	0.9924
Random	0.884	0.838	0.5906	0.4252	0.9104
Time (ms)	3.04	3.1	3.38	6.4	20.86
$APFD_m$	1	1	0.7333	0.7987	0.9960
Random	0.973	0.886	0.6386	0.4245	0.9283
Time (ms)	3.78	3.82	4.1	7.9	26.42
PSO					
$APFD$	0.9	0.9	0.7	0.7785	0.9925
Random	0.882	0.822	0.6026	0.4627	0.9099
Time (ms)	56.78	61.04	66.38	375.6	3154.16
$APFD_m$	1	1	0.7333	0.7987	0.9962
Random	0.97	0.923	0.6346	0.4462	0.9153
Time (ms)	57.54	62.1	66.96	383.06	3210.2
Firefly					
$APFD$	0.9	0.9	0.7	0.7713	0.9920
Random	0.886	0.832	0.6106	0.4216	0.9015
Time (ms)	67.88	70.28	71.46	478.42	13577.78
$APFD_m$	1	1	0.7333	0.7975	0.9957
Random	0.982	0.899	0.6253	0.3855	0.9055
Time (ms)	69.26	70.96	71.86	492.22	14224.48
Fish School					
$APFD$	0.9	0.9	0.7	0.7753	0.9925
Random	0.888	0.818	0.6026	0.4263	0.9132
Time (ms)	314.98	304.42	320.04	1919.96	78026.38
$APFD_m$	1	1	0.7333	0.7975	0.9962
Random	0.949	0.884	0.652	0.405	0.9083
Time (ms)	315.78	306.58	330.26	1895.3	77912.18
Cuckoo Search					
$APFD$	0.9	0.9	0.7	0.7817	0.9925
Random	0.872	0.856	0.5719	0.4476	0.9098
Time (ms)	227.66	230.86	233.1	1131.64	15586.84
$APFD_m$	1	1	0.7333	0.7987	0.9962
Random	0.979	0.913	0.616	0.4038	0.9046
Time (ms)	260.26	259.82	260.6	1290.6	17303.24

parison will be demonstrated later.

Figure 7.22 - 7.24 and *Figure 7.25 - 7.27* display the distribution of APFD value for the prioritised test suite under the original and modified metric. The figures indicate that all the algorithms used in the prioritisation process maintain consistent stability, with only occasional fluctuations observed.

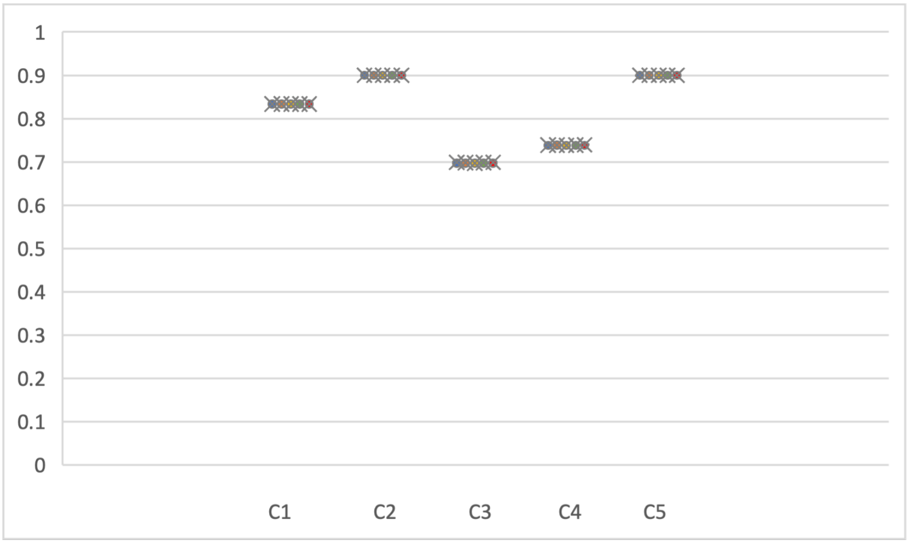


Figure 7.22: MathLib: TCP Distribution under Original APFD Metric (1)

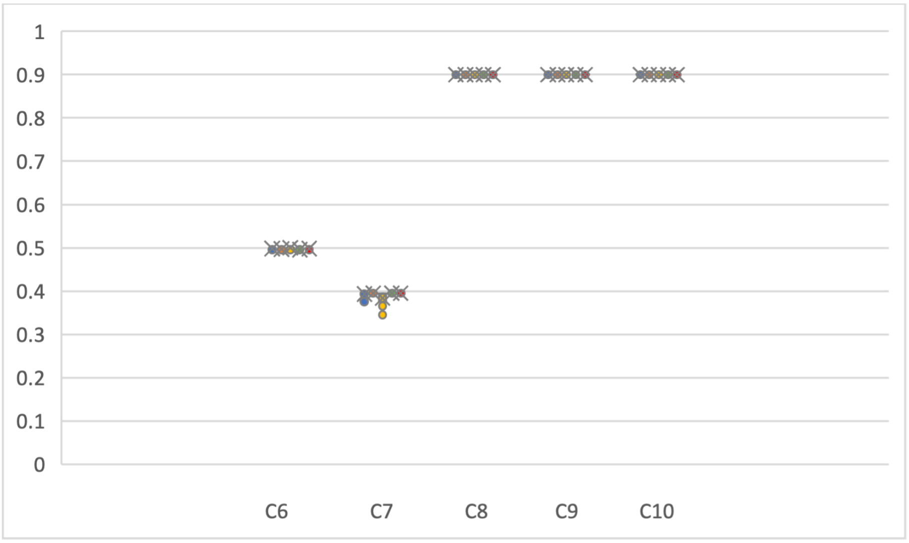


Figure 7.23: MathLib: TCP Distribution under Original APFD Metric (2)

We also performed the non-parametric Wilcoxon test on the original and modified APFD metrics to verify a significance level of 5%, with the null hypothesis that the observed

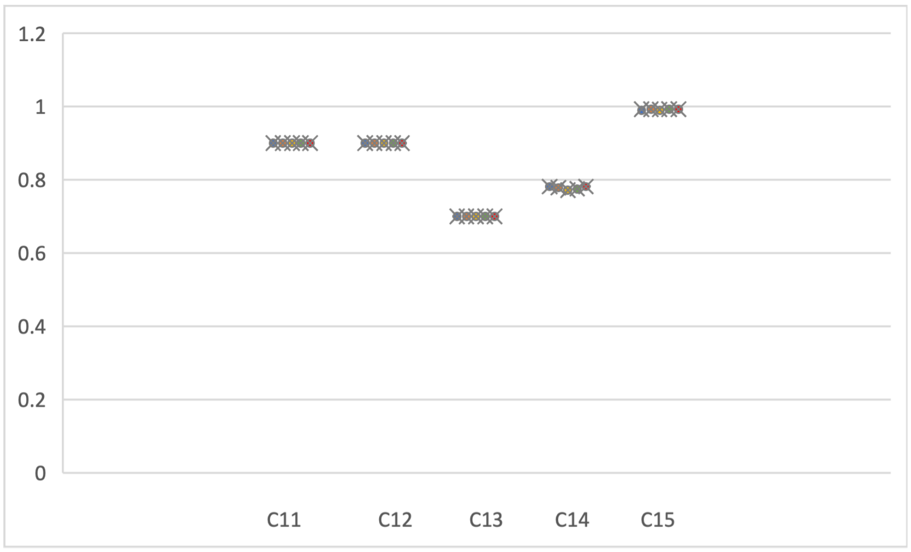


Figure 7.24: MathLib: TCP Distribution under Original APFD Metric (3)

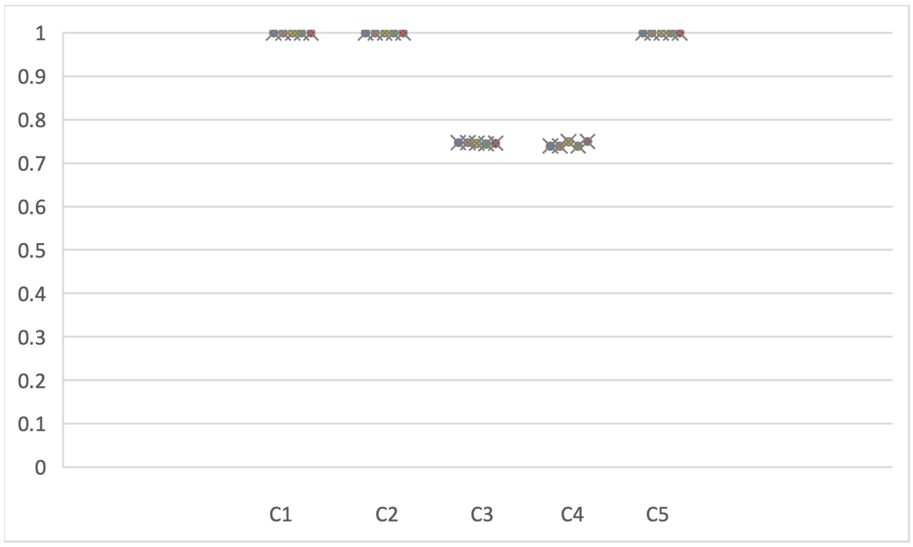


Figure 7.25: MathLib: TCP Distribution under Modified APFD Metric (1)

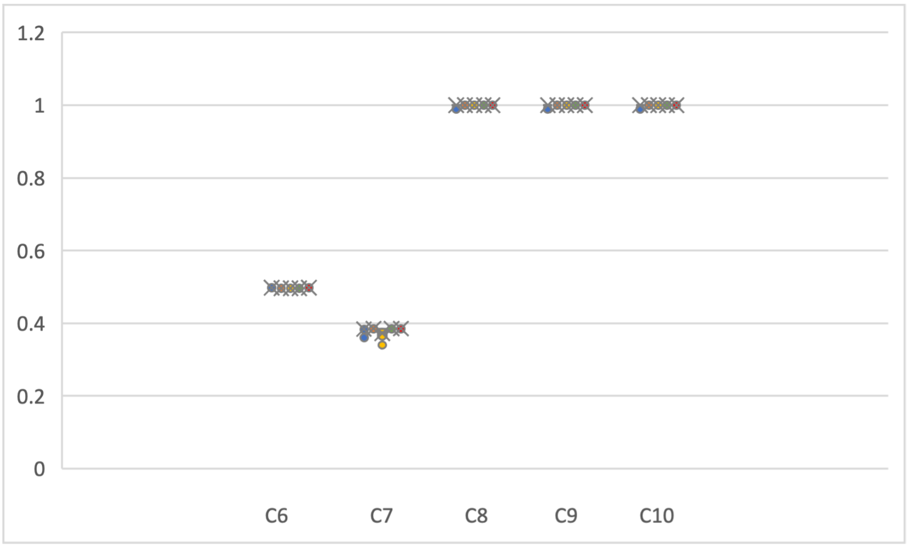


Figure 7.26: MathLib: TCP Distribution under Modified APFD Metric (2)

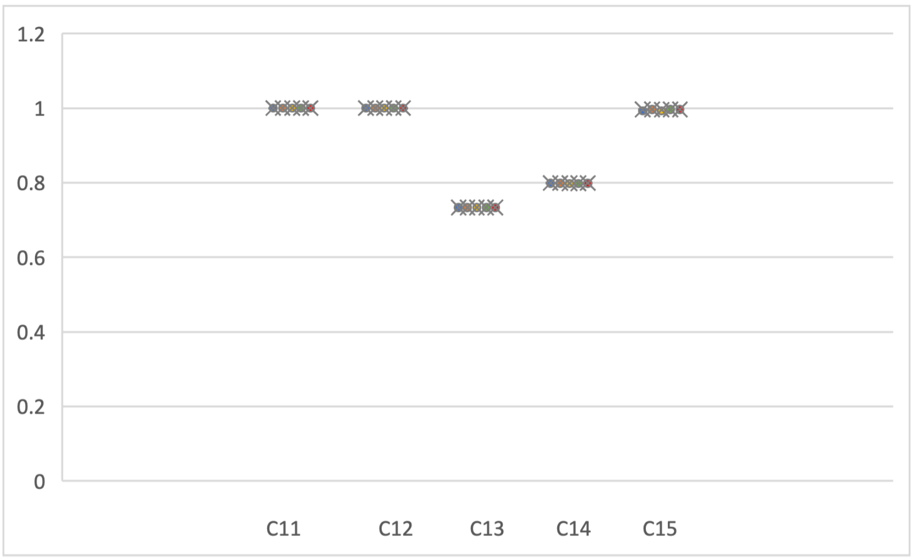


Figure 7.27: MathLib: TCM Distribution under Modified APFD Metric (3)

Table 7.27: MathLib: Comparison between Random Approach (p-value) (1)

Name	**C1**	**C2**	**C3**	**C4**	**C5**
GA					
$APFD$	3.5577e-12	1	1.6850e-15	2.8645e-19	1.0375e-09
$APFD_m$	1.4457e-08	1	1.4205e-13	7.3740e-19	4.3253e-10
PSO					
$APFD$	9.8785e-12	1	2.7943e-17	3.0815e-19	3.9898e-10
$APFD_m$	6.7074e-11	1	9.3113e-11	4.2020e-18	6.0835e-09
Firefly					
$APFD$	4.3051e-10	1	4.6714e-14	2.8188e-18	7.3957e-08
$APFD_m$	6.7798e-11	1	3.0942e-13	3.8581e-19	7.3823e-08
Fish School					
$APFD$	1.2851e-12	1	4.7617e-14	1.3074e-18	2.3531e-11
$APFD_m$	4.3569e-10	1	2.0696e-12	7.1526e-19	8.358e-12
Cuckoo Search					
$APFD$	4.6598e-13	1	1.3963e-15	7.0003e-19	1.6685e-10
$APFD_m$	6.7284e-11	1	1.2587e-11	4.7288e-20	1.5584e-10

differences between the prioritised test suite and random approach are not statistically significant for this case study. The results are shown in *Table 7.27 - 7.29*. The results show that the proposed TCP processes always perform better than the random approach except *C2* and some parts of *C6*.

When we delve into the details of *C2*, we discover that all the generated test cases within the original test suite can detect the same mutants exactly. This leads to a situation where, no matter the sequence of execution, the result will be absolutely the same. Therefore, the evaluation metrics reveal no improvement in this instance, and even when compared to a random approach, the situation remains unchanged with no enhancement. The situation with *C6* arises because multiple orders of test cases can reach an optimal state. In some instances, there may be no significant differences when compared to a random approach.

Table 7.30 - 7.32 shows the collected results when we combine the TCM and TCP process concurrently. Within the tables, certain cells are denoted by "-", indicating that fol-

Table 7.28: MathLib: Comparison between Random Approach (p-value) (2)

Name	**C6**	**C7**	**C8**	**C9**	**C10**
GA					
$APFD$	6.5763e-05	4.2157e-19	1.2073e-12	1.7954e-03	5.8785e-15
$APFD_m$	2.8514e-01	1.6603e-19	1.7046e-10	9.3299e-04	1.7981e-16
PSO					
$APFD$	1.4524e-02	1.2342e-19	3.3858e-12	2.4435e-04	5.5215e-14
$APFD_m$	5.513e-01	3.257e-20	4.285e-10	9.3299e-04	5.9774e-15
Firefly					
$APFD$	6.3786e-02	4.431e-18	4.0094e-10	4.8008e-04	5.3726e-14
$APFD_m$	3.7932e-02	3.1473e-15	6.135e-09	2.4435e-04	5.9264e-15
Fish School					
$APFD$	5.4412e-11	.236e-20	1.3505e-13	4.8008e-04	5.9917e-15
$APFD_m$	2.0518e-07	1.2378e-19	7.0643e-11	1.7954e-03	1.8407e-16
Cuckoo Search					
$APFD$	7.1895e-02	1.2353e-19	2.453e-11	4.8008e-04	6.1069e-15
$APFD_m$	7.3593e-02	3.1709e-20	1.2169e-12	9.3299e-04	1.6095e-13

Table 7.29: MathLib: Comparison between Random Approach (p-value) (3)

Name	**C11**	**C12**	**C13**	**C14**	**C15**
GA					
$APFD$	3.4255e-03	1.7108e-06	4.7475e-14	9.2924e-19	4.7016e-20
$APFD_m$	9.3852e-04	7.353e-08	2.5343e-11	7.9089e-19	1.1096e-19
PSO					
$APFD$	1.7954e-03	3.144e-08	4.3617e-13	3.2716e-18	3.2801e-20
$APFD_m$	9.3299e-04	7.5629e-06	1.4399e-13	3.8972e-17	3.2936e-20
Firefly					
$APFD$	6.4895e-03	7.5941e-06	2.5101e-11	1.143e-17	4.2278e-19
$APFD_m$	1.2231e-02	8.1249e-07	2.6035e-11	7.0318e-20	3.4608e-19
Fish School					
$APFD$	1.2231e-02	3.2195e-08	1.2366e-12	2.7159e-17	3.2907e-20
$APFD_m$	6.8688e-06	7.4166e-08	1.1447e-12	3.7612e-19	3.281e-20
Cuckoo Search					
$APFD$	6.1035e-05	6.4229e-05	4.5655e-16	3.4939e-19	3.2685e-20
$APFD_m$	6.4895e-03	1.7276e-06	5.2238e-14	3.0784e-18	3.2907e-20

Table 7.30: TCO Results for MathLib (1)

Name	**C1**	**C2**	**C3**	**C4**	**C5**
Selection Rate	33.3333	60	60	35.92	20
Detection Loss	0	0	0	0	0
$APFD$	-	0.8333	0.6650	0.7201	-
$APFD_m$	-	1	0.7475	0.7484	-
Random	-	1	0.5900	0.4550	-
Time (ms)	0	3.46	3	4.12	0
Overhead	336.14 ms				

Table 7.31: TCO Results for MathLib (2)

Name	**C6**	**C7**	**C8**	**C9**	**C10**
Selection Rate	35.4285	33.8000	60	60	60
Detection Loss	0	0	0	0	0
$APFD$	0.37	0.3849	0.8333	0.8333	0.8333
$APFD_m$	0.37	0.3547	1	1	1
Random	0.37	0.1959	0.935	1	0.5066
Time (ms)	2.8108	3.64	3.16	2.88	3.18
Overhead	336.14 ms				

lowing the TCM process, only a single test case remains. Consequently, the TCP process becomes redundant and unnecessary in such instances.

In the results of the TCO, we found that certain cases, such as *C9* and *C11*, resemble the previously mentioned *C2*. In these instances, the test cases in the prioritised suite all have the same ability to detect mutants, meaning that the minimisation process did not select the fewest necessary test cases. Consequently, the prioritisation of the minimised suite

Table 7.32: TCO Results for MathLib (3)

Name	**C11**	**C12**	**C13**	**C14**	**C15**
Selection Rate	60	60	60	31.9	41.78
Detection Loss	0	0	0	0	0
$APFD$	0.8333	0.8333	0.5	0.7451	0.9815
$APFD_m$	1	1	0.5555	0.7974	0.9902
Random	1	0.8483	0.5555	0.4431	0.8535
Time (ms)	2.66	2.92	2.68	3.38	8.92
Overhead	336.14 ms				

Table 7.33: MathLib: Comparison of TCP Time after Minimisation (p-value)

Name	**p-value**	**Name**	**p-value**	**Name**	**p-value**
C1	-	**C6**	2.0754e-12	**C11**	6.9555e-05
C2	3.4163e-02	**C7**	1.5432e-16	**C12**	2.9457e-04
C3	7.3789e-05	**C8**	4.3427e-03	**C13**	4.3644e-06
C4	3.0111e-16	**C9**	9.6766e-07	**C14**	7.6147e-16
C5	-	**C10**	6.7426e-07	**C15**	9.5572e-18

became redundant. This occurrence may be attributed to the fact that the minimisation process is a multi-objective optimisation process, with the (modified) APFD metric chosen as one of its goals. Including the number of test cases within the equation for this metric may generate conflicting objectives during the minimisation process, and then leading to this situation. An interesting insight from this observation is the potential for future research, focusing on improving the multi-objective optimisation process.

To explore whether the TCM process might result in time-saving advantages for the TCP process, we performed a non-parametric Wilcoxon test on the time consumed during the TCP process. The findings, illustrated in *Table 7.33*, reveal that in this particular case study, the time required for prioritising the minimised test suites is consistently shorter than that for the original ones, except for the cases where prioritisation process was deemed unnecessary.

In order to compare or rank these TCM algorithms, we conducted pairwise comparisons following Kruskal-Wallis and Dunn's tests. The results are shown in *Table 7.34 and 7.35* , with significant differences between the approaches denoted by different letters within the tables.

From the result tables, we observe that SPEA2 consistently leads the selection rate for the TCM process in the majority of cases, with the exception of *C15*. The PSO,

Table 7.34: MathLib: Rank for TCM Algorithms (1)

Name	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
NSGA-II	a	a	b	b	a	b	a	b	b	b
PSO	b	a	b	c	b	c	b	b	b	b
Cuckoo Search	b	a	b	c	b	c	b	b	b	b
SPEA2	a	a	a	a	a	a	a	a	a	a
MOEA/D	b	a	b	c	a	c	b	b	b	b

Table 7.35: MathLib: Rank for TCM Algorithms (2)

Name	C11	C12	C13	C14	C15
NSGA-II	b	b	b	b	cd
PSO	b	b	b	c	a
Cuckoo Search	b	b	b	c	bc
SPEA2	a	a	a	a	e
MOEA/D	b	b	b	c	de

Cuckoo Search, and MOEA/D algorithms are often grouped together, except in instances *C5* and *C15*. This grouping typically demonstrates the lowest effectiveness in terms of selection rate. NSGA-II exhibits instability in its rankings, aside from *C15*, it ranks first four times, three times between SPEA2 and the aforementioned group, and seven times within that group. *C15* shows a break from the usual pattern, with SPEA2 performing the worst and PSO performing the best. Despite SPEA2 usually performing the best in comparison, the dramatic instability in both effectiveness and efficiency may make users cautious about choosing it for the minimisation process.

Table 7.36 and *Table 7.37* illustrate the comparisons that reveal significant differences between the proposed prioritisation algorithms.

In this case study, under the original and modified APFD metric, the prioritisation algorithms for the three mentioned operations have significant differences, and it seems the Firefly performs slightly worse than others in these instances.

Table 7.36: MathLib: Rank for TCP Algorithms under Original APFD

Name	C6	C7	C15
GA	a	a	a
PSO	a	a	a
Firefly	a	b	b
Fish School	b	a	a
Cuckoo Search	a	a	a

Table 7.37: MathLib: Rank for TCP Algorithms under Modified APFD

Name	C6	C7	C15
GA	a	a	a
PSO	b	a	a
Firefly	b	b	b
Fish School	b	a	a
Cuckoo Search	a	a	a

However, for the overall TCP process for this study, all these prioritisation algorithms have similar effectiveness under the (modified) APFD metric. When also taking time usage into account, GA appears to exhibit the best overall performance.

7.7 Case Study 4: UML2PY

Our final case study is UML2PY, and the detailed system specification is in *Appendix E*. *Table 7.38* details this system. Within the UML2PY study, these operations are symbolised by codes *D1 - D18* for simplicity. For this case study, 216 class instances are constructed for BasicExpression and one for Property.

Similarly, *Table 7.39 - 7.41* present the average outcomes from this case study when only executing the TCM process. The tables display metrics like the selection rate and the fault detection loss rate, both shown in percentages, and time consumption, measured in milliseconds.

Examining the results table reveals patterns consistent

Table 7.38: UML2PY Case Study Details

	Name	Fault	Test	APFD	$APFD_m$	Detection
D1	instancesOps	2	2	0.25	0.125	0.5
D2	contextAndObject	648	5	0.3666	0.3666	0.6666
D3	isOclExceptionCreation	2808	5	0.8384	0.9230	0.9230
D4	mapAttributeExpression	864	125	0.5949	0.5934	0.625
D5	mapBasicExpression	1944	125	0.3213	0.3186	0.3333
D6	mapErrorCall	432	25	0.5533	0.5499	0.8333
D7	mapFunctionExpression	6696	125	0.7118	0.7114	0.9032
D8	mapInsertAtFunctionExpression	432	125	0.764	0.764	1
D9	mapIntegerFunctionExpression	1080	125	0.156	0.1528	0.2
D10	mapOperationExpression	864	125	0.708	0.707	0.75
D11	mapReferencedAttributeExpression	648	125	0.3213	0.3186	0.3333
D12	mapSetAtFunctionExpression	432	125	0.764	0.764	1
D13	mapStaticAttributeExpression	216	125	0.9560	0.9560	1
D14	mapStaticOperationExpression	432	125	0.916	0.916	1
D15	mapTypeExpression	4752	5	0.7787	0.8530	0.8939
D16	mapValueExpression	1728	5	0.7	0.7625	0.875
D17	mapVariableExpression	2592	125	0.5413	0.5396	0.5833
D18	noContextnoObject	648	5	0.1	0.0333	0.3333

with earlier case studies. Both NSGA-II and MOEA/D algorithms appear as the most time-efficient for the TCM process. In this case study, the PSO algorithm often takes the longest, with Cuckoo Search being slightly quicker, though they operate at roughly the same time efficiency level. The SPEA2 algorithm typically falls between these two groups. One observation unique to this study is that some operations demand significantly more time compared to previous case studies, attributed to the presence of large-scale mutants and test cases. Notably, for the operation *D7*, the most time-consuming algorithm, PSO, takes an average of approximately 27.4 hours to finalise the prioritisation process, whereas NSGA-II completes it in roughly 20 minutes.

Figure 7.28 - 7.30 illustrate the distribution of the selection rate for each minimisation algorithm based on 50 runs.

Based on the displayed figures, most algorithms exhibit consistent performance across scenarios. However, as observed in earlier case studies, SPEA2 occasionally shows vari-

Table 7.39: TCM Results for UML2PY (1)

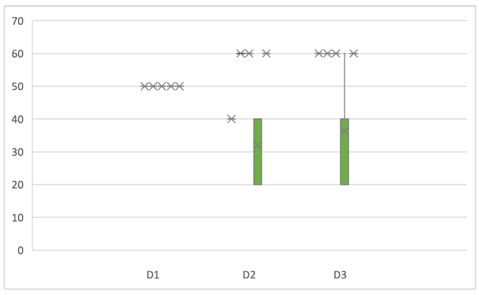
Name	D1	D2	D3	D4	D5	D6
NSGA-II						
Selection Rate	50	40	60	41.776	41.776	43.68
Detection Loss	0	0	0	0	0	0
Time (ms)	12.22	593.12	20205.2	23694.46	127896.86	492.86
PSO						
Selection Rate	50	60	60	42.0159	41.968	43.68
Detection Loss	0	0	0	0	0	0
Time (ms)	563.14	60998.52	2062836.24	2478647.52	13888421.92	45358.52
Cuckoo Search						
Selection Rate	50	60	60	45.232	46.688	46
Detection Loss	0	0	0	0	0	0
Time (ms)	482.06	61555.5	1684309	1967762.96	10113338.46	41394.46
SPEA2						
Selection Rate	50	32	36.4	41.5680	41.568	37.76
Detection Loss	0	0	0	0	0	0
Time (ms)	1658.34	9932.02	189385.04	990830.24	4881267.74	23234.98
MOEA/D						
Selection Rate	50	60	60	42.2079	42.0479	43.68
Detection Loss	0	0	0	0	0	0
Time (ms)	24.24	666.2	17663.84	19419.98	103200.42	428.7

Table 7.40: TCM Results for UML2PY (2)

Name	D7	D8	D9	D10	D11	D12
NSGA-II						
Selection Rate	42.096	42.208	42.1279	41.888	41.712	42.096
Detection Loss	0	0	0	0	0	0
Time (ms)	1048380.34	748.02	5535.28	25250.92	14143.82	739.28
PSO						
Selection Rate	42.544	42.688	44.368	41.904	41.968	42.464
Detection Loss	0	0	0	0	0	0
Time (ms)	98869108.7	73691.26	572120.88	2564276.3	1477092.04	73061.34
Cuckoo Search						
Selection Rate	45.68	45.648	49.184	44.368	47.04	45.68
Detection Loss	0	0	0	0	0	0
Time (ms)	93198424.92	77653.56	419637.1	2041206.14	1164418.82	77923.72
SPEA2						
Selection Rate	41.056	41.024	40.672	41.264	41.776	41.024
Detection Loss	0	0	0	0	0	0
Time (ms)	49173291.32	36393.4	219861.18	995533.2	566077	37613.18
MOEA/D						
Selection Rate	42.3199	42.1759	42.3519	42.3199	42.3519	42.128
Detection Loss	0	0	0	0	0	0
Time (ms)	823357.66	728.02	4186.18	20819.68	11254.42	732.96

Table 7.41: TCM Results for UML2PY (3)

Name	D13	D14	D15	D16	D17	D18
NSGA-II						
Selection Rate	41.76	42.352	40	20	42.384	20
Detection Loss	0	0	0	0	0	0
Time (ms)	679.84	972.8	58970.94	7264.12	60109.9	165.28
PSO						
Selection Rate	41.76	41.872	60	60	42.634	60
Detection Loss	0	0	0	0	0	0
Time (ms)	68293.36	92934.7	6336982.94	765060.48	6975869.46	15361.48
Cuckoo Search						
Selection Rate	44.224	44.3679	60	60	47.2479	60
Detection Loss	0	0	0	0	0	0
Time (ms)	68780.6	98861.48	4634164.6	611897.24	5582338.56	14714.72
SPEA2						
Selection Rate	41.84	41.888	32	20	41.936	20
Detection Loss	0	0	0	0	0	0
Time (ms)	32576.06	48980.24	648155.7	294334.14	2364379.64	9738.62
MOEA/D						
Selection Rate	42.0959	41.9679	40	60	42.2879	60
Detection Loss	0	0	0	0	0	0
Time (ms)	629.62	934.3	49361.1	6708.26	54250.44	257.02



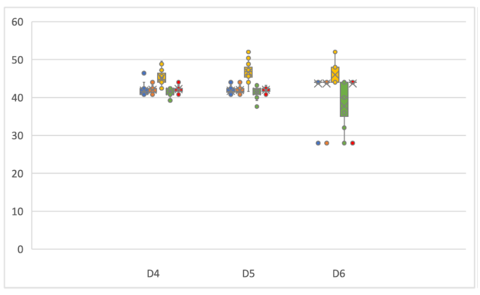
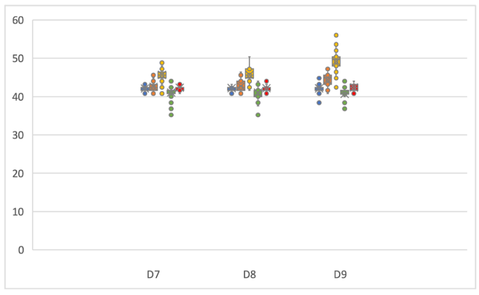


Figure 7.28: UML2PY: TCM Distribution (1)



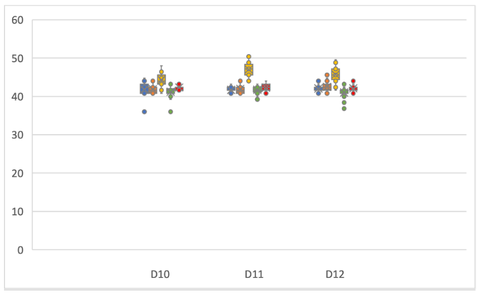
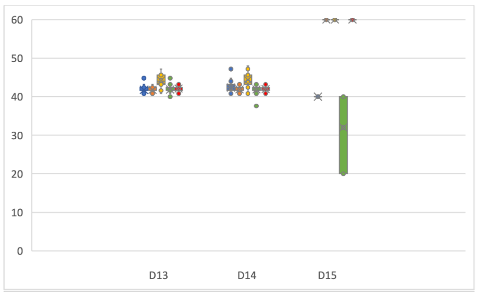


Figure 7.29: UML2PY: TCM Distribution (2)



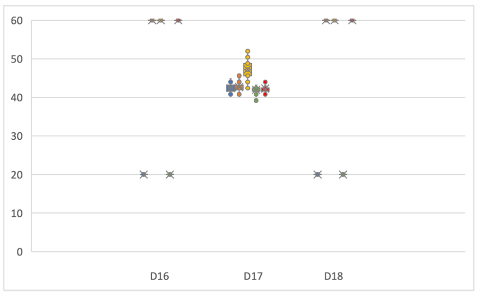


Figure 7.30: UML2PY: TCM Distribution (3)

ability, notably in operations such as *D2* and *D6*.

Table 7.42 - 7.44 detail the results of the TCP process for the UML2PY case study. In this study, when using the Fish School algorithm for prioritisation, the process was not completed even after two days. Consequently, we deemed this algorithm unsuitable for this case study and omitted its results.

In the UML2PY case study, the results tables reveal a trend similar to previous findings that the GA algorithm consistently takes the lead in time efficiency compared to other algorithms. After GA, the time usage hierarchy is evident with PSO, Firefly, and then Cuckoo Search. Remarkably, the Fish School algorithm was deemed unsuitable for this context. The infeasibility of this specific algorithm was due to the inability to provide the prioritisation results within a practical period (two days), which may be attributed to the complexity and extensive scope of operations encountered during the optimisation process. Specifically, for operation *D7*, the most time-consuming in the TCM process, the TCP process demanded roughly 15 hours for PSO, Firefly and Cuckoo Search algorithm. In contrast, GA accomplished the same task in about 15 minutes.

Figure 7.31 - 7.33 and *Figure 7.34 - 7.36* display the distribution of APFD value for the prioritised test suite under

Table 7.42: TCP Results for UML2PY (1)

Name	D1	D2	D3	D4	D5	D6
GA						
$APFD$	0.5	0.5666	0.8384	0.6237	0.3346	0.8193
Random	0.395	0.4919	0.8384	0.6217	0.3338	0.7113
Time (ms)	4.1	857.68	16469.72	42589.78	261379.36	536.06
$APFD_m$	0.5	0.5666	0.9230	0.6248	0.3332	0.8323
Random	0.305	0.4799	0.9230	0.6212	0.3326	0.7223
Time (ms)	2.68	814.32	15830	41444.58	245046	496.24
PSO						
$APFD$	0.5	0.5666	0.8384	0.6235	0.3346	0.8186
Random	0.36	0.5066	0.8384	0.6217	0.3341	0.6999
Time (ms)	76.78	45832.4	858736.96	2259018.64	13455823.62	27727.04
$APFD_m$	0.5	0.5666	0.9230	0.6246	0.3332	0.8323
Random	0.3275	0.4906	0.9230	0.6194	0.3327	0.7219
Time (ms)	107.76	46813.14	862121.54	2253994.9	13832833.56	27334.82
Firefly						
$APFD$	0.5	0.5666	0.8384	0.6230	0.3346	0.818
Random	0.355	0.5013	0.8384	0.6213	0.3340	0.7139
Time (ms)	80.4	52213.96	887041.62	2296554.22	13776911.8	28891.14
$APFD_m$	0.5	0.5666	0.9230	0.6245	0.3332	0.8313
Random	0.305	0.4733	0.9230	0.6207	0.3320	0.7143
Time (ms)	106.66	52337.18	886176	2281375.68	13603419.02	29127.74
Cuckoo Search						
$APFD$	0.5	0.5666	0.8384	0.6235	0.3345	0.8186
Random	0.425	0.4626	0.8384	0.6222	0.3341	0.7073
Time (ms)	143.56	62795.66	967612.74	2538895.52	14863963.4	32780.54
$APFD_m$	0.5	0.5666	0.9230	0.6247	0.3332	0.8313
Random	0.2675	0.4853	0.9230	0.6210	0.3326	0.7213
Time (ms)	173.66	60245.52	944511.4	2502393.66	14934077.9	32597.1

Table 7.43: TCP Results for UML2PY (2)

Name	**D7**	**D8**	**D9**	**D10**	**D11**	**D12**
GA						
$APFD$	0.8997	0.9956	0.2022	0.7479	0.3346	0.9958
Random	0.8850	0.9537	0.1938	0.7461	0.3338	0.9387
Time (ms)	1040076.5	229.88	12145.1	39253.58	30050.82	230.02
$APFD_m$	0.9029	0.9995	0.1998	0.7498	0.3332	0.9995
Random	0.8821	0.9357	0.1892	0.7452	0.3316	0.9250
Time (ms)	862544.42	213.6	11878.32	38329.26	29532.8	211.1
PSO						
$APFD$	0.8995	0.9956	0.2023	0.7478	0.3346	0.9956
Random	0.8834	0.9537	0.1914	0.7459	0.3338	0.9320
Time (ms)	48785943.08	16599.44	639865.78	2088140.46	1595350.4	16844.14
$APFD_m$	0.9027	0.9995	0.1999	0.7496	0.3332	0.9992
Random	0.8870	0.9508	0.1861	0.7474	0.3323	0.9373
Time (ms)	55057559.78	16842.86	645427.7	2066891.94	1578389.76	16815.5
Firefly						
$APFD$	0.8995	0.9950	0.2022	0.7477	0.3345	0.9958
Random	0.8840	0.9438	0.1901	0.7449	0.3339	0.9382
Time (ms)	57198281.86	38360.04	672176.44	2106987.04	1614246.84	38369.52
$APFD_m$	0.9027	0.9992	0.1997	0.7498	0.3330	0.9990
Random	0.8856	0.9401	0.1875	0.7480	0.3326	0.9261
Time (ms)	56424854.34	38264.08	668027.34	2105416.82	1605824.26	38221.56
Cuckoo Search						
$APFD$	0.8997	0.9956	0.2022	0.7479	0.3346	0.9958
Random	0.8832	0.9396	0.1910	0.7456	0.3342	0.9460
Time (ms)	53906267.84	46976.9	742452.6	2341433.1	1794201.98	47049.76
$APFD_m$	0.9029	0.9997	0.1998	0.7498	0.3330	0.9995
Random	0.8867	0.9363	0.1885	0.7470	0.3322	0.9296
Time (ms)	57305775.3	46567.48	742623.92	2330855.4	1773364.76	46324.86

Table 7.44: TCP Results for UML2PY (3)

Name	**D13**	**D14**	**D15**	**D16**	**D17**	**D18**
GA						
$APFD$	0.9958	0.9958	0.8060	0.8	0.5781	0.3666
Random	0.9941	0.9649	0.7951	0.7455	0.5164	0.2253
Time (ms)	78.94	228.38	54273.12	7152.04	116588.42	256.4
$APFD_m$	0.9995	0.9997	0.8803	0.875	0.5771	0.3333
Random	0.9962	0.9716	0.8732	0.8155	0.5262	0.1526
Time (ms)	74.8	210.4	52146.72	6763.22	112976.4	232.82
PSO						
$APFD$	0.9958	0.9958	0.8060	0.8	0.5793	0.3666
Random	0.9938	0.9678	0.7922	0.75	0.5258	0.2293
Time (ms)	8322.14	16352.32	2884341	390749.7	6141649.2	13199.5
$APFD_m$	0.9992	0.9995	0.8803	0.875	0.5786	0.3333
Random	0.9965	0.9733	0.8717	0.8025	0.5153	0.1546
Time (ms)	8314.9	16370.84	2888595.3	389998.76	6191992.22	13154.84
Firefly						
$APFD$	0.9956	0.9956	0.8060	0.8	0.5697	0.3666
Random	0.9931	0.9641	0.7962	0.74	0.5178	0.2333
Time (ms)	29064.62	37665.78	3038223.3	438775.28	6732420.08	13766.24
$APFD_m$	0.9992	0.9990	0.8803	0.875	0.5693	0.3333
Random	0.9963	0.9645	0.8706	0.822	0.5254	0.1613
Time (ms)	28875.32	37246.18	3023643.3	435634.14	6650265.7	13516.42
Cuckoo Search						
$APFD$	0.9956	0.9958	0.8060	0.8	0.5709	0.3666
Random	0.9939	0.9664	0.7929	0.7465	0.5244	0.2386
Time (ms)	35780.54	45453.42	3353145.94	505206.06	7434245.54	15310.02
$APFD_m$	0.9990	0.9992	0.8803	0.875	0.5710	0.3333
Random	0.9982	0.9657	0.8704	0.81025	0.5198	0.1726
Time (ms)	35207.52	44817.48	3388085.74	502585.88	7440845.72	14999.12

the original and modified metric. In this case study, the Fish School algorithm was not applicable. Thus, the distribution figures for the TCP process display groups with four distinct colours. Each colour represents a specific prioritisation algorithm, arranged in the order of GA, PSO, Firefly, and Cuckoo Search. The figures highlight the consistent stability across the algorithms during the prioritisation process, with only infrequent fluctuations observed.

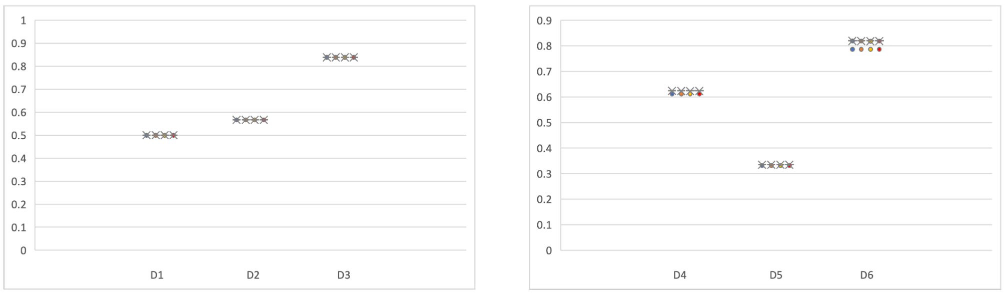


Figure 7.31: UML2PY: TCP Distribution under Original APFD Metric (1)

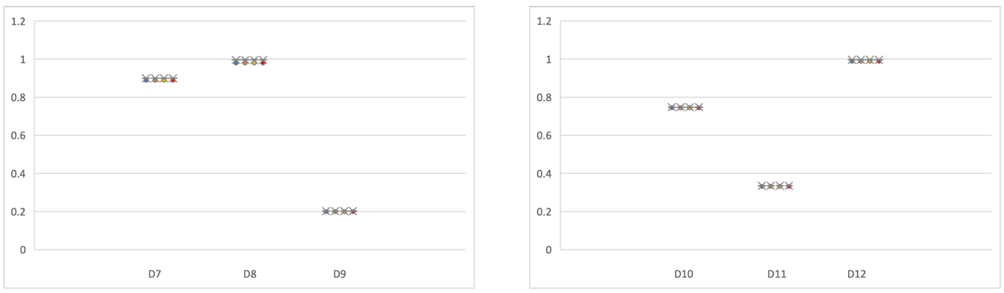


Figure 7.32: UML2PY: TCP Distribution under Original APFD Metric (2)

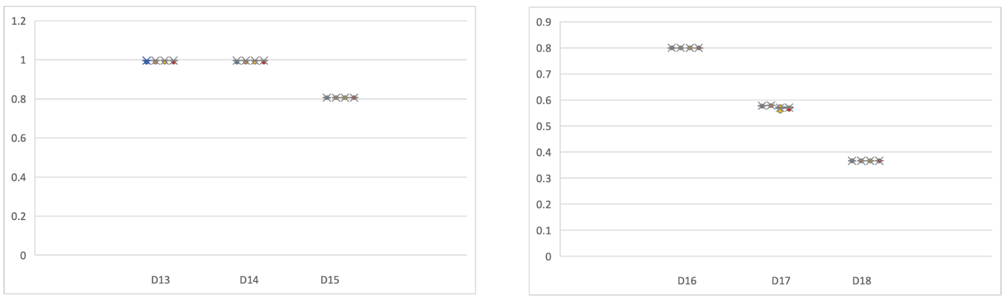


Figure 7.33: UML2PY: TCP Distribution under Original APFD Metric (3)

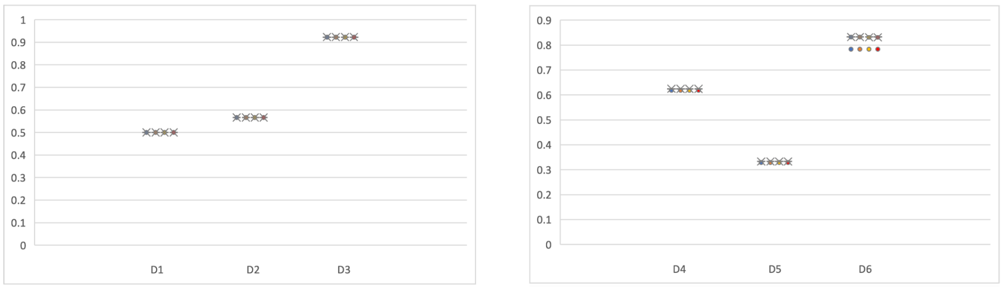


Figure 7.34: UML2PY: TCP Distribution under Modified APFD Metric (1)

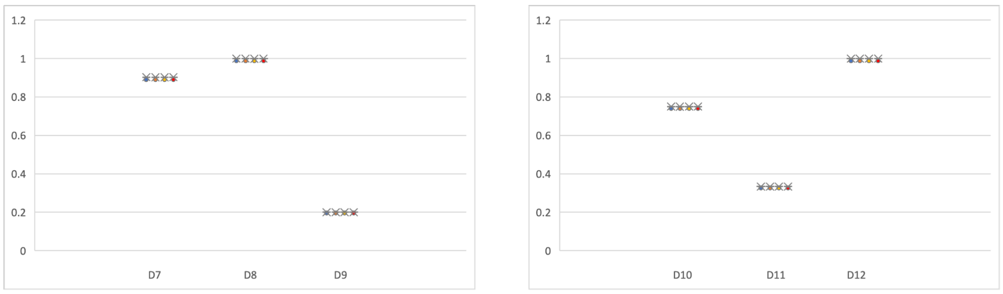


Figure 7.35: UML2PY: TCP Distribution under Modified APFD Metric (2)

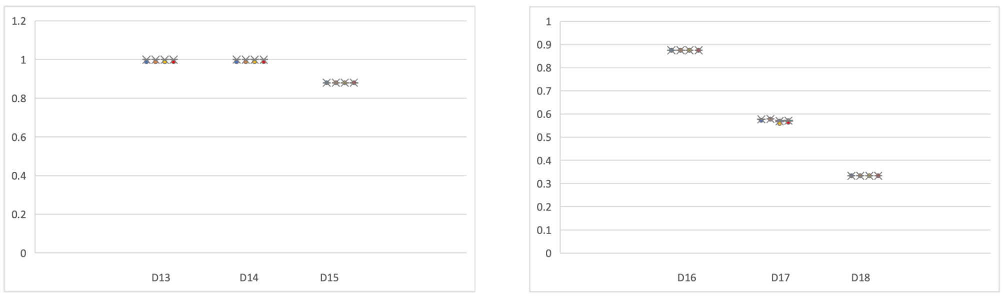


Figure 7.36: UML2PY: TCM Distribution under Modified APFD Metric (3)

Table 7.45: UML2PY: Comparison between Random Approach (p-value) (1)

Name	**D1**	**D2**	**D3**	**D4**	**D5**	**D6**
GA						
$APFD$	2.9743e-07	1.0287e-09	1	2.0601e-04	2.1455e-03	1.6213e-13
$APFD_m$	3.7869e-09	2.4871e-11	1	6.5772e-06	1.4988e-02	5.0707e-13
PSO						
$APFD$	5.6318e-10	1.4024e-08	1	1.5261e-04	1.4735e-02	1.2496e-15
$APFD_m$	5.5342e-08	6.4274e-11	1	1.4579e-08	2.8292e-02	1.95e-14
Firefly						
$APFD$	2.0898e-10	1.4199e-08	1	6.1823e-04	7.8706e-03	1.1941e-13
$APFD_m$	3.7869e-09	1.2078e-12	1	7.6233e-05	1.0742e-03	1.2417e-14
Cuckoo Search						
$APFD$	2.9902e-05	1.4193e-13	1	1.8928e-03	2.5674e-02	1.2751e-15
$APFD_m$	2.6485e-11	2.4251e-11	1	5.2278e-06	8.1289e-03	5.4178e-13

Table 7.46: UML2PY: Comparison between Random Approach (p-value) (2)

Name	**D7**	**D8**	**D9**	**D10**	**D11**	**D12**
GA						
$APFD$	1.1572e-17	1.239e-19	1.0928e-16	1.8434e-06	1.1115e-03	7.7392e-20
$APFD_m$	2.5618e-16	1.0107e-15	5.6305e-15	3.3971e-08	4.7624e-05	6.7245e-18
PSO						
$APFD$	1.2472e-14	1.3831e-19	1.7622e-17	1.6553e-05	5.7755e-04	1.1494e-19
$APFD_m$	5.8263e-13	1.8745e-18	6.1491e-17	4.6044e-05	1.4756e-03	2.1169e-16
Firefly						
$APFD$	4.1214e-16	1.6763e-18	1.9878e-17	2.0306e-06	2.2163e-03	8.6179e-20
$APFD_m$	3.485e-15	7.8144e-15	4.6132e-17	6.5663e-04	9.4138e-02	1.3838e-17
Cuckoo Search						
$APFD$	3.6818e-17	1.0605e-19	2.4099e-17	1.6983e-05	1.5233e-02	9.6899e-20
$APFD_m$	1.1405e-15	1.1378e-17	2.1633e-18	2.3372e-06	3.504e-03	2.916e-15

We utilised the non-parametric Wilcoxon test on both the original and modified APFD metrics, setting a significance threshold at 5%. The underlying null hypothesis was that the differences observed between the prioritised test suite and the random approach were not statistically significant in this study. As detailed in *Table 7.45 - 7.47*, the evidence suggests that our proposed TCP methods generally outperform the random strategy, apart from in the *D3* instance.

Upon closely examining *D3*, we find a similarity with the earlier mentioned *C2*. In this situation, every test case generated within the original suite can precisely identify the same mutants. Consequently, irrespective of the test sequence, the outcomes remain identical. Thus, no improvement is

Table 7.47: UML2PY: Comparison between Random Approach (p-value) (3)

Name	**D13**	**D14**	**D15**	**D16**	**D17**	**D18**
GA						
$APFD$	6.4108e-20	1.186e-19	4.2023e-10	1.7658e-16	5.9389e-18	5.4067e-15
$APFD_m$	1.9747e-03	1.589e-14	8.072e-07	1.5751e-06	6.3766e-18	1.8221e-16
PSO						
$APFD$	9.6665e-20	8.9637e-20	5.0696e-14	1.8262e-14	4.7193e-20	6.0764e-15
$APFD_m$	3.5749e-03	1.5191e-13	7.4968e-08	2.0925e-12	6.6195e-20	5.3744e-17
Firefly						
$APFD$	2.5909e-18	3.83e-19	4.1211e-10	4.7257e-14	1.1163e-17	6.0088e-16
$APFD_m$	3.3804e-03	4.504e-13	1.0312e-09	8.833e-04	5.8993e-16	6.0735e-16
Cuckoo Search						
$APFD$	1.484e-18	9.1537e-20	6.1603e-11	1.8516e-16	6.1851e-17	4.5794e-13
$APFD_m$	2.3344e-01	8.8409e-15	2.5228e-09	3.414e-09	7.8126e-18	1.8148e-15

Table 7.48: TCO Results for UML2PY (1)

Name	**D1**	**D2**	**D3**	**D4**	**D5**	**D6**
Selection Rate	50	40	60	41.744	41.584	44
Detection Loss	0	0	0	0	0	0
$APFD$	-	0.4166	0.7820	0.6223	0.3362	0.8015
$APFD_m$	-	0.4166	0.9230	0.6246	0.3330	0.8310
Random	-	0.4166	0.9230	0.6175	0.3306	0.6787
Time (ms)	0	413.34	11093.66	14645.76	82150.2	348.76
Overhead	1958134.06 ms					

indicated by the evaluation metrics, and this is maintained even when compared to a random approach, with no evident progress in results.

For the case of *D13*, when evaluated using the modified APFD metric, the Cuckoo Search algorithm did not show a significant difference from the random approach. As this was an isolated incident, we theorised it might be due to random variation. To confirm this, we executed an additional 20 runs for this specific scenario, and the results delivered a p-value below 0.05.

Table 7.48 - 7.50 presents the outcomes from concurrently running the TCM and TCP processes. In these tables, cells marked with "-" denote situations where, after the TCM process, only one test case is left. As a result, proceeding with the TCP process in these cases is redundant.

In this study, for operations *D1*, *D16* and *D18*, the results

Table 7.49: TCO Results for UML2PY (2)

Name	**D7**	**D8**	**D9**	**D10**	**D11**	**D12**
Selection Rate	42.144	42.2559	42.16	42.144	42.064	42.144
Detection Loss	0	0	0	0	0	0
$APFD$	0.8952	0.9893	0.2055	0.7451	0.3362	0.9901
$APFD_m$	0.9027	0.9986	0.1998	0.7498	0.3330	0.9994
Random	0.8705	0.8910	0.1776	0.7445	0.3312	0.8751
Time (ms)	499110.5	219.52	3896.86	13545.98	9918.06	225.18
Overhead	1958134.06 ms					

Table 7.50: TCO Results for UML2PY (3)

Name	**D13**	**D14**	**D15**	**D16**	**D17**	**D18**
Selection Rate	41.776	42.4	40	20	42.416	20
Detection Loss	0	0	0	0	0	0
$APFD$	0.9902	0.9901	0.6742	-	0.5686	-
$APFD_m$	0.9998	0.9994	0.8598	-	0.5670	-
Random	0.9966	0.9359	0.8598	-	0.4686	-
Time (ms)	66.34	222.08	29368.18	0	38629.96	0
Overhead	1958134.06 ms					

post-minimisation made the subsequent prioritisation redundant. Like certain cases within the MathLib case study, we found in operation *D3*, the test cases in the prioritised suite all have the same ability to detect mutants, meaning that the minimisation process did not select the fewest necessary test cases. The reason for this could be tied to the multi-objective optimisation for the minimisation process, which may demand deeper exploration in future studies.

To assess if the TCM process offers time efficiency benefits for the TCP process, we conducted a non-parametric Wilcoxon test on the time spent during the TCP process for the original and minimised test suite. The results are presented in *Table 7.51*.

From our analysis, in most instances of this study, the time taken to prioritise minimised test suites is consistently shorter than the original ones, except for cases where prioritisation was considered unnecessary. However, there were

Table 7.51: UML2PY: Comparison of TCP Time after Minimisation (p-value)

Name	p-value	Name	p-value	Name	p-value
D1	-	**D7**	7.0661e-18	**D13**	2.9601e-18
D2	6.9178e-18	**D8**	1.5718e-08	**D14**	2.5375e-15
D3	7.0581e-18	**D9**	7.0597e-18	**D15**	7.0613e-18
D4	7.0613e-18	**D10**	7.0645e-18	**D16**	-
D5	1.3493e-16	**D11**	7.0645e-18	**D17**	7.0661e-18
D6	1.1607e-16	**D12**	2.1596e-12	**D18**	-

Table 7.52: UML2PY: Rank for TCM Algorithms (1)

Name	D1	D2	D3	D4	D5	D6	D7	D8	D9	D10
NSGA-II	a	a	b	ab	a	b	b	b	b	ab
PSO	a	b	b	ab	a	b	b	b	c	ab
Cuckoo Search	a	b	b	c	b	c	c	c	d	c
SPEA2	a	a	a	a	a	a	a	a	a	a
MOEA/D	a	b	b	b	a	b	b	b	b	b

some instances, specifically in operations *D8*, *D12* and *D14*. Although the comparison results are significantly different, the prioritisation time for the original test suite is slightly less than the minimised one. Given that these processes were completed within one second, this may be due to the influence of system scheduling. These rare occurrences could be a point of in-depth study in future research.

To evaluate and rank the TCM algorithms, we employed Kruskal-Wallis and Dunn's tests for pairwise comparisons. The outcomes can be found in *Table 7.52* and *Table 7.53*. Distinct letters in the tables highlight significant variances among the methods.

Table 7.53: UML2PY: Rank for TCM Algorithms (2)

Name	D11	D12	D13	D14	D15	D16	D17	D18
NSGA-II	a	b	a	a	a	a	ab	a
PSO	ab	b	a	a	b	b	b	b
Cuckoo Search	c	c	b	b	b	b	c	b
SPEA2	ab	a	a	a	a	a	a	a
MOEA/D	b	b	a	a	b	b	ab	b

Table 7.54: UML2PY: Rank for TCP Algorithms

	Original: D17	**Modified: D17**
GA	b	b
PSO	a	a
Firefly	c	c
Cuckoo Search	c	c

Based on the results from the tables, SPEA2 consistently tops the selection rate in the TCM process across all scenarios. Contrary to the previous MathLib case study, NSGA-II here presents a notable challenge to SPEA2, often falling within the same group. PSO and MOEA/D consistently show comparable effectiveness in terms of selection rate. In contrast, Cuckoo Search always performs the worst if there is any significant difference existing in this study. When factoring in the time metric, NSGA-II seems more favourable for the minimisation process.

Table 7.54 presents the comparison results for TCP algorithms. Particularly, only in operation *D17* do we find a notable difference between these algorithms when considering both the original and modified APFD metrics. Therefore, we have combined these two parts into a single table.

In these two instances, PSO stands out as the top performer, closely followed by GA. Cuckoo Search and Firefly report similar levels of effectiveness. Upon examining the detailed average results (both original and modified), it is clear that, although statistically distinct, the differences between them are less than 0.01. Summarising this case study, all prioritisation algorithms exhibit similar effectiveness when evaluated using the original and modified APFD metrics. However, GA takes the lead when we account for time efficiency, and the Fish School algorithm is found unsuitable for this setting.

7.8 Results & Discussions

At the beginning of this chapter, we proposed five main research questions, and then the results for four real-world case studies have been detailed in the previous sections. In this section, based on the experiment results, we will clarify the answers to those research questions alongside the discussions based on the results.

• **RQ 1: *Effectiveness.* What is the effectiveness of the TCO processes within the context of OCL?**

RQ 1.1: Effectiveness for the TCM process.

RQ 1.1.1: What is the test suite reduction rate achieved by the adapted algorithms during the TCM process?

Answer: An analysis of *Table 7.3, 7.12, 7.21 - 7.23, and 7.39 - 7.39* indicates that the execution of the TCM process on the original test suite results in a reduction of size between 33.3% and 81.8%, without any loss in fault detection capacity. Generally, the minimised test suite contains around 40% of the original test cases. *These results confirm the efficacy and safety of applying the TCM process to systems whose specifications are expressed in OCL.*

Figure 7.13, 7.16, 7.19 - 7.19 and 7.28 - 7.30 display the distribution of the selection rate for the minimised test suite for each case study. Each plot in these figures corresponds to a particular operation and minimisation algorithm within the respective case study. Observing these charts, it is noticeable that, except for the SPEA2 algorithm, *the minimisation process consistently exhibits a stable performance with only slight deviations in most scenarios.*

RQ 1.1.2: What is the fault detection capability of the adapted algorithms during the TCM process?

Answer: Drawing from the data in *Table 7.3, 7.12, 7.21 - 7.23, and 7.39 - 7.39*, we noticed that across all four case studies, the rates of fault detection loss all stand at zero. This suggests that post-minimisation, the resulting test suites retain the fault detection capability exhibited by their original ones. Thus, by inference, applying TCM algorithms on systems whose specifications are expressed in OCL does not diminish their fault detection capability, *considering these methodologies as both effective and safe.*

RQ 1.2: Effectiveness for the TCP process.

RQ 1.2.1: What are the performances of the adapted TCP algorithms under the (modified) APFD metric?

Answer: We examined the effectiveness of the TCP process using the original and modified APFD metrics without the involvement of the TCM process. From the *Table 7.4, 7.13, 7.24 - 7.26, and 7.42 - 7.44*, we can find regardless of which metric is used for evaluation, *there is an observable increase in the APFD value following the prioritisation process in most cases.* While *A5, C2, C10, C12*, and *D3* show no improvements, we further analysed these specific instances. The absence of enhancement can be traced to the optimal sequencing already present in the original test suite, which results in the prioritisation process not offering any additional benefits to the testing process.

From *Figure 7.14 - 7.15, 7.17 - 7.18, 7.22 - 7.27 and 7.31 - 7.36* display the distribution of the evaluation results for the prioritised test suites under the original and modified APFD metrics. Each plot in these figures corresponds to a particular operation and prioritisation algorithm under the corresponding APFD metric. Regarding these charts, it is noticeable that, for all adapted algorithms, *the prioritisation process consistently exhibits a stable performance with only occasional fluctuations observed.*

RQ 1.2.2: How does the performance of the adapted algorithms during the TCP process compare with a random approach?

Answer: We observed that, with the exceptions of operations *A5, C2, C10, C12*, and *D3*, enhancements were seen in all instances under both the original and modified APFD metrics. These particular operations already maintained the optimal sequences within their original test suites. To explore deeper into the effectiveness of our algorithms, we performed the non-parametric Wilcoxon test on the original and modified APFD metrics to verify a significance level of 5%, with the null hypothesis that the observed differences between the prioritised test suite and random approach are not statistically significant for these case studies. Referencing the results in *Table 7.5, 7.14, 7.27 - 7.29 and 7.45 - 7.47*, it is evident that most operations display a noticeable variance from the random approach, except for operations *C2*, *D3*, and specific comparisons of *C6*. The latter, *C6*, is unique as numerous test case orders can achieve an optimal state, occasionally resulting in no significant difference from a random approach.

Diving deeper into the characteristics of the two exceptional operations, *C2* and *D3*, it becomes clear that every generated test case in the original test suite can identically detect the same mutants. This equivalence leads to a scenario where, irrespective of the execution order, outcomes remain consistent. Thus, our evaluation metrics do not report any significant improvement in such situations, and this consistency persists even when compared with a random strategy. Based on these findings, we can infer that *the prioritised test suites always exhibit significant deviations from random approaches.*

RQ 1.3: Does the sequence of TCP and TCM processes affect the overall optimisation process?

Answer: In our integrated TCO approach, combining the TCM and TCP phases, our goal was not to identify the ultimate combination of optimisation algorithms. Instead, we applied the NSGA-II for the TCM and the GA for TCP during the evaluation based on their consistent and reliable performances. The TCO strategy began with applying TCM to the original test suite and subsequently moved to the TCP. The collective results can be reviewed in *Table 7.6, 7.15, 7.30 - 7.32, and 7.48 - 7.50.*

The presence of "-" in certain table cells signifies that post-minimisation, only a singular test case was left in the test suite. This causes the subsequent TCP process redundant. Observing beyond these particular cases, it becomes evident that *the TCO processes invariably promote the effectiveness of the testing process, both in terms of selection rate and the (modified) APFD metrics.*

In some instances, like *C2* and *C9*, and *D3*, the minimisation process did not yield the smallest test suite. This might be due to the multi-objective nature of the minimisation process, where the (modified) APFD metric, including the number of test cases within the equations, can introduce conflicting goals. The purpose of this objective is to favour earlier testing ability when multiple test cases exhibit similar fault detection capabilities. A preliminary experiment we conducted on the running example showed that this metric indeed selected the corresponding test cases and achieved its purpose. Although this objective sometimes prevents the minimisation process from achieving the optimal minimisation result and needs further investigation, we still suggest that *the (modified) APFD metric should remain a part of the objectives during the TCM process.*

We were also keen to explore whether the TCM and TCP sequence influenced the final optimisation outcomes. While the effectiveness remains largely uninfluenced by the sequence, efficiency witnesses a notable impact where the TCM process trims the search space for the TCP process. A non-parametric comparison of the time usages between the TCP process applied to both original and minimised test suites is illustrated in *Table 7.7, 7.16, 7.33, and 7.51.* These tables substantiate that *the TCM process consistently increases the efficiency of the TCP process, except for cases where prioritisation is unnecessary.*

In summary to RQ 1, the results validate that applying the TCP process, the TCM process, or a combination of both, generally brings advantages to the testing process for systems whose specifications are expressed in OCL. This observation is consistent regardless of whether the original or the modified version of APFD is used as the evaluation metric.

- **RQ 2: *Scalability.* How scalable are the proposed algorithms in handling real-world OCL specifications with varying complexities?**

To thoroughly evaluate the scalability and adaptability of our adapted algorithms and proposed TeCO framework, we conduct four distinct real-world case studies. These studies are chosen based on their variability in terms of complexity and size. Specifically, when assessed based on lines of specifications, the sizes ranged from a brief 41 lines to a more extensive 1053 lines. Regarding operations, the number was from a minimal 2 to a more comprehensive 18. When considering the number of clauses of OCL expressions for a specific operation, the number is from 1 to 29. A critical metric, the number of faults, which is represented by the mutants generated across various instances, had a wide range, fluctuating from a mere 2 to a substantial 6696. Furthermore, when we measure the scalability based on the number of produced test cases, the figures range from 2 to 243.

Considering the time factor, the overall time overhead for the TCO process, utilising both the NSGA-II and GA algorithms, was commendably efficient. It ranged from an almost immediate duration of a mere second to about 30 minutes, depending on the complexities, typically shorter than the time one might spend on a lunch break.

Considering this broad range of data and the consistent performance across diverse scenarios, it is evident that *our adapted optimisation algorithms and TeCO framework are well-suited for the systems whose specifications are expressed in OCL, irrespective of their size or complexity.*

• **RQ 3: *Comparison.* Which of the adapted TCO algorithms performs optimally during the optimisation processes?**

RQ 3.1: During the TCM process, which algorithm demonstrates superior performance?

For the TCM process, we adapted five algorithms, NSGA-II, PSO, Cuckoo Search, SPEA2 and MOEA/D, in this work. Examining their effectiveness using the selection rate metric, data from *Table 7.8, 7.17, 7.34 - 7.35 and 7.52 - 7.53* reveals that definitively determining the best algorithm for the TCM process is a complex task.

For the Bond study, SPEA2, MOEA/D, and PSO took the top rank, with NSGA-II running behind, then the Cuckoo Search. In the Interest Rate study, there was no definitive leader. In the MathLib study, SPEA2 consistently outperformed others, with NSGA-II following closely and the rest under-performing. For the UML2PY study, while SPEA2 took the lead, NSGA-II was a close competitor. PSO and MOEA/D were mid-tier performers, with Cuckoo Search performing the worst.

Considering execution time, NSGA-II and MOEA/D were the fastest. Like the selection rate, the execution time for SPEA2 is still unstable and inconsistent across operations. Due to the unstable and unpredictable performance of SPEA2, *the users should be cautious when choosing SPEA2 as the minimisation algorithm.* Given the consistent performance in both effectiveness and efficiency, we suggest for *NSGA-II as the primary choice for the TCM process.*

RQ 3.2: During the TCP process, which algorithm performs the best?

We selected and adapted five prioritisation algorithms, GA, PSO, Firefly, Fish School and Cuckoo Search, in this research work. We intend to compare their effectiveness when measured under both the original and modified APFD metrics. Surprisingly, *there was minimal variation in the performance across these algorithms.* However, certain instances, namely *A1*, *C5*, *C7*, *C15*, and *D7*, demonstrated the minor differences between these algorithms. Exploring deeper into the details of these outcomes, we noticed that while these results were statistically significant, their actual differences were minor, indicating that the practical differences in real-world applications might be negligible.

But when considering the time factor inside, *the GA consistently exhibited superior speed.* In contrast, the Fish School algorithm was notably the slowest and even cannot produce the result in a reasonable time within the UML2PY case study.

RQ 3.3: Which type of OCL specification is most compatible with the proposed algorithms?

During our assessment, we conducted experiments on four real-world case studies, each varying in size and complexity. Two primary facets appeared to affect the general optimisation processes.

The Impact of Original Test Suite: The intrinsic character of the original test suite sets the upper limitation for the optimisation process. While the inherent fault detection capability remains unaltered post-optimisation, the restructured suite might detect defects earlier or require fewer test cases instead of more defects. Both the TCM and TCP processes alter the original test suite, and our evaluation metrics consistently compare the optimised suite with the original and random ones. This raises a question: Are these optimisations still meaningful if the initially generated test suite is considered sufficient, the conditions we have already noted in some instances of our case studies? Indeed, it is always desirable to optimise the test case generation process. However, it is crucial to note that a perfect generation process cannot always be guaranteed. Furthermore, numerous pre-existing test suites still need to be optimised, illustrating the continuing applicability of these optimisation techniques.

The Influence of System Specifications: System specifications predominantly impact optimisation efficiency. Key factors here include specification length, number of operations, and expression types. Logically, if specifications feature larger numbers of operations or lengths (or more clauses in the post-condition), the optimisation process will naturally demand more time. While size is often discussed, the nature of the expressions should not be overlooked. Specifications only composed of straightforward mathematical operations tend to be processed more swiftly than those involving String or Collection-based operations.

In summary, the original test suite and system specifications jointly influence the optimisation process, with their co-relationship determining both the outcome and efficiency of the TCO processes.

To summarise RQ 3, for the TCM process, while SPEA2 often yields the smallest test cases, its unpredictability in both effectiveness and efficiency makes NSGA-II a more reliable choice. In the TCP process, the adapted algorithms display similar performance when evaluated under the original and modified APFD metrics. However, considering execution speed, GA emerges as the better option. Determining which specification type best suits the proposed optimisation algorithms is complex. Factors like specification length, operation numbers, and expression types influence the overall performance of the adapted algorithms. However, employing NSGA-II and GA ensures that one consistently derives optimised test suites effectively and efficiently in most scenarios.

- **RQ 4: *Metric.* How to evaluate the modified APFD metric?**

RQ 4.1: What are the differences between the original and the modified APFD metric?

This research question aims to uncover the correlation and variances between the original and modified versions of the APFD metric. The modified variant enhances the original by introducing a reward and penalty system for the test case and undetectable defects. Throughout our experimental procedure, we could not determine a direct map formula between these two metrics. However, we can consolidate our findings into three main observations.

Firstly, when the number of defects identifiable by the first test case matches the number of undetectable defects, both metrics will produce identical results since the α and β will cancel each other out. *In this case, the modified APFD metric essentially reverts to its original form.*

Secondly, when the initial test case can identify a majority of defects, surpassing those that are undetectable, then the value derived from the modified APFD will generally be higher than that from the original APFD metric.

In the third scenario, conversely, when the first test case has a lesser capability to detect defects compared to those that remain undetectable, the modified APFD value will typically fall below the original one.

This dynamic highlights how the modified APFD metric seeks a balance between early fault detection ability and accounting for undetected defects. These key features make this metric more suitable for MBT and mutation testing procedures.

RQ 4.2: How does the TCP process perform when evaluated under the original and modified metrics?

During the prioritisation process, employing either the original or the modified APFD metric as the objective showed no significant differences in effectiveness. However, the modified version provided improvements over the original by adjusting the scale of results, managing undetectable defects, and promoting the early detection of faults. Meanwhile, there are some aspects of the modified metric that deserve clarification. Our modifications aimed to address the constraints of the original metric, notably its inability to span a 0 to 1 range and its approach to managing undetectable faults. For the modified APFD metric, in situations where all faults can be detected, the maximum value could touch 1. But, if some faults remain unattainable, the highest value matches the fault detection rate, noticeable in cases like *B2* and *D3*. *Despite the improvements of the modified version brings over the original and makes it more suitable under the MBT and mutation testing context, it is clear that there still have room for further enhancement of the modified APFD metric.*

Given that the original APFD metric does not account for undetected defects, we have made an assumption during the evaluation process. When we encounter a mutant that cannot be detected, we predict that the final test case will detect it, setting TF_i equal to the total number of test cases in the test suite. This assumption poses a potential threat to the research's validity.

Lastly, the matter of equivalent mutants is also necessary to be discussed. These are mutated specifications that, even after mutation, retain the behaviour of the original specification. Identifying equivalent mutants is recognised as an undecidable problem. Yet, they do influence mutation testing, especially for the results of mutation scores. *In our experimental framework, we have directly classified equivalent mutants as undetectable.*

In summary of RQ 4, in the test case prioritisation process, applying either the original or the modified APFD metric as the guiding objective demonstrated no substantial differences in their overall effectiveness. Nevertheless, the modified APFD metric introduced notable advancements by fine-tuning the scale of measurement, effectively addressing undetectable defects, and enhancing the capacity for early fault detection. This refined version incorporates a reward and penalty mechanism for the test cases that detect defects at the beginning and for those defects that remain undetected. Although there is potential for further refinement of the modified APFD metric, it is better tailored for the context of MBT and mutation testing than the original one.

- **RQ 5: *Efficiency.* What is the time overhead when the TCO processes are applied to the systems whose specifications are expressed in OCL?**

RQ 5.1: What is the overhead for each proposed algorithm during the TCM process?

Based on the results from *Table 7.3, 7.12, 7.21 - 7.23, and 7.39 - 7.41*, it is evident that for the minimisation process, *NSGA-II and MOEA/D consistently top the rankings in terms of time efficiency.* As for its effectiveness, the efficiency ranking of the SPEA2 algorithm showcases inconsistency. Aside from the SPEA2 algorithm, PSO generally surpasses Cuckoo Search in performance, although there are exceptions in the UML2PY case study.

For the notably intricate operation, *D7*, NSGA-II took an average of 17.5 minutes, while MOEA/D finished in 13.7 minutes to complete the minimisation process. On the other hand, the slowest algorithm for this task, PSO, took roughly 27.4 hours. Even though SPEA2 often tops the list in effectiveness, it still consumed 13.6 hours for this operation. Given this evident difference in efficiency, we still recommend that *NSGA-II as the primary choice for the minimisation process.*

RQ 5.2: What is the overhead for each proposed algorithm during the TCP process?

The time needed for the prioritisation process varies considerably, depending on the specific size and complexity of each operation. Nonetheless, from *Table 7.4, 7.13, 7.24 - 7.26, and 7.42 - 7.44*, a consistent ranking of the adapted algorithms emerged across the four studies. *GA consistently topped the list, followed by PSO, Firefly, Cuckoo Search, and Fish School taking the most time.* The exception was in the Bond case study, where Firefly outperformed PSO.

In most cases, GA was able to generate a prioritised test suite swiftly, often within seconds. In our evaluation of the most complicated operation, *D7*, GA took an estimated 14.5 minutes. In contrast, other algorithms took around 15.5 hours for the same task, with Fish School failing to yield results in a practical time frame.

Further, when examining time consumption for prioritising both the original and minimised test suite using GA, as highlighted earlier, *the TCM process consistently sped up the time usage for the TCP process.* This efficiency can be attributed to the TCM process reducing the search space for the prioritisation process.

RQ 5.3: What is the overhead during the pre-process phase preceding the optimisation processes?

In the evaluation process section, we detailed that before the actual optimisation, whether users opt for TCM, TCP, or a combination of both, the TeCO framework necessitates a pre-processing stage to extract essential information. Recognising the overhead of this stage is crucial. The data from *Table 7.6, 7.15, 7.30, and 7.48* gives insight into the overhead of the TCO processes under our default configuration (NSGA-II for TCM and GA for TCP). Across these four studies, the average overheads for the initial phase are 3728.92, 37.6, 74.16 and 16790.48 milliseconds, accounting for 8.1%, 10.2%, 18%, and 0.8% of the total time consumption, respectively.

From this data, it is clear that the initial phase tends to consume a certain proportion of the total overhead, an understandable outcome given that it encompasses input-output operations, which are time-consuming. The Interest Rate and MathLib studies seem to have a higher percentage, mainly because their total overheads fall below one second with the default configuration of the TeCO framework. The MathLib study has more operations with the system specifications, which leads to more input-output actions, explaining its higher percentage of time consumption. As the complexity and size expand, this percentage typically drops as more time gets directed towards the optimisation process, as observed in the UML2PY study. In sum, it is clear that *the time taken for the pre-processing phase is a reasonable fraction of the entire optimisation process.*

In summarising RQ5, it is evident that the time required for both TCP and TCM is influenced by the specific optimisation algorithm employed. Our evaluation, based on four real-world case studies, indicated that for the TCM process, MOEA/D and NSGA-II consistently lead the competition in performance. Considering both its efficiency and effectiveness, NSGA-II is preferred for the minimisation process. While all the adapted TCP algorithms displayed comparable effectiveness, GA appeared the most time-efficient. The initial pre-processing time before the optimisation process takes a certain portion of the total time, but it remains reasonable and acceptable.

7.9 Threats to Validity

This section addresses the potential threats to validation during the evaluation phase.

Threats to Internal Validity: This threat is related to potential inconsistencies in the treatment of case studies that might influence the outcomes [144]. We have mitigated these threats by ensuring all case studies were conducted under the same configuration and environment, although these configuration settings are determined through an informal combinatorial experiment.

Every algorithm was written in JAVA, and each experiment was conducted on the same machine. For both TCM and TCP algorithms, consistent parameters were used across all studies, such as population size and number of maximum iterations, regardless of the differences in OCL specification. These precautions helped minimise the internal validity threats.

While these configurations have been validated through an informal combinatorial analysis, we cannot ensure they are optimal. This research primarily aimed to verify the feasibility of applying TCO processes to systems with specifications in OCL instead of the configurations of the optimisation algorithms. Users of the TeCO framework can adjust these configurations based on their specific needs.

Threats to External Validity: Although we used four case studies and one running example for our experiments, the primary threat to external validity is the limited generalisability of these chosen studies.

As with any empirical evaluation, the OCL specifications in this study may not fully represent the entire population. To mitigate this threat, we selected systems that are expressed in OCL with varying complexity, with generated test cases ranging from a handful to hundreds.

Moreover, as with the OCL specifications, the chosen optimisation algorithms do not encompass the entire population. We selected five algorithms each for the TCM and TCP processes, analysing and comparing them. These algorithms have been well-researched, which helps in mitigating potential external threats.

Threats to Construct Validity: Our work adapted ten optimisation algorithms to navigate the TCM and TCP processes. The stochastic nature of the evolutionary algorithm may result in inconsistent outcomes between different executions. To minimise this threat, we ran each experiment 50 times and used the averaged results.

Another threat is linked to the minimisation algorithms being the multi-objective optimisation algorithms, producing a Pareto set. It is impractical to examine every solution in the Pareto set manually. To deal with this threat, we selected the solution from the result set with the maximum fault detection ability and minimised test size according to the objective of the TCM process.

In the TCM process, we have identified three objectives, whereas the TCP process is guided by a single objective. These selections are based on the inherent purposes of minimisation and prioritisation problems. The adequacy of these objectives or the potential enhancement from integrating more objectives remains an area for further exploration in future research.

Furthermore, some operations within the OCL standard library lacked corresponding mutation operators in this work because, based on our strategies, we were unable to identify suitable mutation operators for these operations, leading us to omit them.

Lastly, our approach of ignoring equivalent mutants and treating them as undetectable also presents a potential validity threat. These gaps might be bridged with contributions from other OCL users and through further research.

Chapter 8

Conclusions & Future Works

8.1 Overview of Thesis

We have explored the feasibility of applying various TCO techniques to the systems whose specifications are expressed in OCL and validated the adapted algorithms through four real-world case studies in this research work.

Through a systematic literature review on the topic of UML-based test case generation, we found numerous methods have been proposed for the generation process. However, post-generation TCO was often ignored. The few studies which did consider TCO highlighted its potential to benefit the testing process. With the evolving OCL standards, its utility in system definitions has amplified, but only limited research tried to bridge the gap.

In this thesis, we adapted five algorithms each for TCM and TCP processes and implemented them within the TeCO framework. Given the abstraction level of OCL is the same as the system model, it remains independent of implementation languages. This independence allows for a one-time TCO process application, benefitting all detailed specification implementations, irrespective of programming language or platform variances. However, this universality poses challenges. For instance, the inability to access real system defects becomes a burden in guiding the TCO. In order to solve this condition, we proposed the full-set mutation operations and corresponding classifications to the OCL standard library.

After the adaption of these algorithms, we evaluated and compared their effectiveness and efficiency through four real-world case studies with various sizes and complexities. Recognising the limitations of the initial APFD metric in the context of MBT and mutation testing, we introduced a modified APFD metric. In general, within the TCM process, it is challenging to single out one superior minimisation algorithm definitively. However, based on the stability, selection rate and time usage, we recommend the NSGA-II could be the first choice. As for the TCP process, all the adapted algorithms presented similar levels of effectiveness when evaluated with both the original and modified APFD metrics. Nevertheless, when considering efficiency, GA consistently ranked highest across five prioritisation algorithms.

When we combined the TCM and TCP processes, NSGA-II and GA, we observed consistent efficiency improvements for the TCP process after the TCM process. This efficiency arises as the minimised test suite reduces the search space during prioritisation. Across four case studies, the total overhead time for the TCO process has been proven acceptable, fluctuating from a few seconds to around 30 minutes based on the complexities, typically less than a lunch break.

Chapter 2 offers a comprehensive background to our research, which includes essential information about the software development process, model-driven engineering, model-based testing and object constraint language. Within the software development section, the traditional and agile development models are both discussed. The chapter then switched its focus to the context of MDE and MBT, explaining their advantages, general process and corresponding development tools. Lastly, the chapter presents an in-depth overview of the OCL, highlighting its development journey, characters and overall structure.

Chapter 3 presents the related works for this research. First, a systematic literature review of the directly related topic, UML-based test case generation, is demonstrated. Our findings revealed that many existing approaches often overlook test case optimisation, motivating us to research the TCO processes. The chapter then discussed the three primary TCO processes, which are test case prioritisation, test case minimisation and test case selection. The chapter concludes with an insightful discussion of mutation testing.

Chapter 4 proposes full-set mutation operators for the OCL standard version 2.4, mainly based on Clause 11 *"OCL standard Library"*. These operators will benefit the testing process when actual system defects are unavailable. We also propose the classifications of the proposed operators, which provide more options to the users of OCL in languages such as ATL, QVT, etc. The classifications will benefit the OCL practitioners in mutation testing and allow them to choose which kinds of mutants will be generated based on their testing purposes.

Chapters 5 and 6 provide a comprehensive exploration of the adapted algorithms, which include the problem definition, solution presentation and default configurations. Additionally, standard evaluation metrics for the TCP and TCM processes are discussed in these chapters. Within Chapter 5, we analysed the limitations of the original APFD metric in the context of MBT and mutation testing and subsequently proposed a modified version of the APFD metric to mitigate these shortcomings.

Chapter 7 begins with a detailed explanation of the research questions to the evaluation process. Then we employed a running example to demonstrate the general process of the TeCO framework. This chapter forwards to the evaluation of the OCL-based TCO processes, utilising four case studies of various complexities. The performances of different optimisation algorithms for the TCP and TCM processes are compared and investigated. We addressed the initial research questions and the corresponding discussions based on the experiment results.

This thesis has delivered the following contributions:

- Undertook a systematic literature review focusing on UML-based test case generation, offering insights into overall methodologies within test case generation.

- Introduced a comprehensive set of mutation operators related to the OCL standard library.

- Categorised the proposed mutation operators into different classifications based on shared logic principles.

- Assessed the shortcomings of the widely-adopted APFD metric under the context of MBT and mutation testing and introduced a modified version to address these limitations.

- Adapted and implemented five optimisation algorithms specific to the TCP process alongside a discussion on relevant evaluation metrics.

- Adapted and implemented five optimisation algorithms for the TCM process, accompanied by a discussion on relevant evaluation metrics.

- Executed a comprehensive evaluation of OCL-based TCO methodologies across four case studies with various complexities.

8.2 Limitation

Our research encountered three primary limitations.

Firstly, the absence of relevant literature on the topic posed a challenge. While TCO techniques are well-studied in broader contexts, there is a noticeable lack of research related to the OCL context. This made it difficult to benchmark and compare our adapted algorithms with established methods in the field. To address this, we shifted our focus, comparing our adapted TCM and TCP algorithms among themselves. For the TCP process, we also compared them with an essential random approach, establishing a baseline for our analysis.

The second limitation revolved around the mutation operators linked to the OCL standard library. We introduced a comprehensive set of operators and their classifications, yet certain operations or operators remained devoid of corresponding mutation operators. We hope that, with collective efforts from the OCL community and our future research, these gaps will be bridged. It is also worth mentioning that we intentionally excluded mutation operators for some new proposed OCL types or operations, primarily because they are not universally supported by all OCL tools.

The last one is about the optimisation algorithms that we utilised. In our study, we adapted and evaluated five TCM and five TCP algorithms. However, these selections do not encompass the full range of available optimisation techniques. There might be other algorithms out there that offer superior performance in terms of both effectiveness and efficiency. Our primary objective was to assess the feasibility of applying TCO processes to systems whose specifications are expressed in OCL. For both the TCP and TCM processes, our parameter configurations might not be optimal. Moreover, in the combined TCO process, we did not exhaustively evaluate all potential combinations of these algorithms. Instead, we employed an informal combinatorial approach to determine these configurations.

8.3 Future Works

The final section outlines potential future research directions.

A promising area for future exploration centres around the optimisation process, notably in the domain of multi-objective optimisation. During our evaluation process, there were scenarios where the minimised test suite was not optimal. This might be attributed to the conflicts between objectives. Addressing these conflicts presents a fascinating research challenge. Moreover, assessing the advantages of integrating more objectives or experimenting with varying objective combinations in TCP and TCM processes remains a topic for more in-depth exploration.

An intriguing outcome was observed during the TCM process, particularly with the Running Example, Case Study 2 and operation C15, where the minimised test suite ended up with a larger number of test cases than the faults they were capable of detecting. Following the minimisation process, it became apparent that the remaining test cases outnumbered the faults within the system. Such an imbalance indicates that the TCM process may not always yield the most efficient or minimal test suite. The reason behind this phenomenon remains unclear despite attempts to understand its underlying causes. This unexpected outcome highlights a need for further in-depth analysis in subsequent research.

To address the limitations of the original APFD metric in the context of MBT and mutation testing, we introduced a modified APFD metric. This modification employed a static reward and penalty system to address the corresponding limitations, rewarding test cases that detect defects in the first place and penalising those undetected defects. Future work could explore a dynamic reward and penalty approach to enhance this modified APFD metric further.

As the OCL standard library expands with newer expressions and operators from the community, system specifications will inevitably incorporate these additions. This presents a continual need to develop mutation operators for these evolving elements. Although our present strategies for the standard library serve their purposes, they can benefit from refinements and enhancements. Notably, there are OCL operations that do not have corresponding mutation operators due to the challenges in finding a proper operator with our current strategies. We hope collaborations with the community and further research will bridge this gap.

Although we neglected the equivalent mutants during our evaluation in this study, their presence undoubtedly presents a captivating avenue for research. The existence of undetectable equivalent mutants can mislead mutation testing results, particularly the mutation score. If an overwhelming number of these mutants are present, it could lead to misjudgments regarding the effectiveness of test cases. Simple equivalent mutants might be easily identified through static analysis. However, for the complex expressions or data types, such as String and Collection, there remains significant exploration to be done.

The field of optimisation is vast, and there is always potential for new algorithms or techniques that might offer better efficiency or effectiveness. Future work can investigate more algorithms for evaluation, especially those that have been proposed in recent studies.

Last, we plan to extend the TCO processes to a broader array of OCL specifications to establish more general findings. We intend to consider a more extensive range of OCL expressions and operators to obtain more precise outcomes, thereby refining our results and increasing the array of system specifications suitable for case study examination.

In conclusion, our journey through the intricacies of testing and optimisation in the context of OCL has revealed several insights and also pointed towards areas that deserve further exploration.

Bibliography

[1] D. Maciel, A. C. Paiva, and A. R. Da Silva, "From requirements to automated acceptance tests of interactive apps: An integrated model-based testing approach." in *ENASE*, 2019, pp. 265–272.

[2] P. R. Srivatsava, B. Mallikarjun, and X.-S. Yang, "Optimal test sequence generation using firefly algorithm," *Swarm and Evolutionary Computation*, vol. 8, pp. 44–53, 2013.

[3] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software testing, verification and reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[4] N. Gupta, A. Sharma, and M. K. Pachariya, "An insight into test case optimization: ideas and trends with future perspectives," *IEEE Access*, vol. 7, pp. 22 310–22 327, 2019.

[5] S. Ali, T. Yue, M. Zohaib Iqbal, and R. K. Panesar-Walawege, "Insights on the use of OCL in diverse industrial applications," in *International conference on system analysis and modeling*. Springer, 2014, pp. 223–238.

[6] A. M. Madni and M. Sievers, "Model-based systems engineering: motivation, current status, and needed ad-

vances," in *Disciplinary convergence in systems engineering research.* Springer, 2018, pp. 311–325.

[7] R. J. Lipton, "Fault diagnosis of computer programs," 1971.

[8] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 654–665.

[9] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proceedings of the 27th international conference on Software engineering*, 2005, pp. 402–411.

[10] Z. Wei, W. Xiaoxue, Y. Xibing, C. Shichao, L. Wenxin, and L. Jun, "Test suite minimization with mutation testing-based many-objective evolutionary optimization," in *2017 International Conference on Software Analysis, Testing and Evolution (SATE).* IEEE, 2017, pp. 30–36.

[11] O. M. Group, "Omg document formal/2014-02-03," in *Object Constraint Language (OCL) Specification. Version 2.4.*, 2014.

[12] M. d. C. de Castro-Cabrera, A. García-Dominguez, and I. Medina-Bulo, "Trends in prioritization of test cases: 2017-2019," in *Proceedings of the 35th annual acm symposium on applied computing*, 2020, pp. 2005–2011.

[13] K. Jin and K. Lano, "Generation of test cases from UML diagrams-a systematic literature review," in *14th*

Innovations in Software Engineering Conference (formerly known as India Software Engineering Conference), 2021, pp. 1–10.

[14] K. Lano, S. Kolahdouz-Rahimi, and K. Jin, "OCL libraries for software specification and representation," in *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, 2022, pp. 894–898.

[15] Y. B. Leau, W. K. Loo, W. Y. Tham, and S. F. Tan, "Software development life cycle agile vs traditional approaches," in *International Conference on Information and Network Technology*, vol. 37, no. 1, 2012, pp. 162–167.

[16] W. W. Royce, "Managing the development of large software systems: concepts and techniques," in *Proceedings of the 9th international conference on Software Engineering*, 1987, pp. 328–338.

[17] B. Shiklo, "8 software development models: Sliced, diced and organized in charts," Nov 2022. [Online]. Available: https://www.scnsoft.com/blog/software-development-models

[18] P. Pulse, "Pulse of the profession." Project Management Institute Newtown Square, PA, 2017.

[19] P. Abrahamsson, O. Salo, J. Ronkainen, and J. Warsta, "Agile software development methods: Review and analysis," *arXiv preprint arXiv:1709.08439*, 2017.

[20] R. Kneuper, "Sixty years of software development life cycle models," *IEEE Annals of the History of Computing*, vol. 39, no. 3, pp. 41–54, 2017.

[21] K. Schwaber and J. Sutherland, “The scrum guide,” *Scrum Alliance*, vol. 21, no. 19, p. 1, 2011.

[22] V. Guntamukkala, H. J. Wen, and J. M. Tarn, “An empirical study of selecting software development life cycle models,” *Human Systems Management*, vol. 25, no. 4, pp. 265–278, 2006.

[23] T. Dybå and T. Dingsøyr, “Empirical studies of agile software development: A systematic review,” *Information and software technology*, vol. 50, no. 9-10, pp. 833–859, 2008.

[24] N. B. Ruparelia, “Software development lifecycle models,” *ACM SIGSOFT Software Engineering Notes*, vol. 35, no. 3, pp. 8–13, 2010.

[25] Y. Bassil, “A simulation model for the waterfall software development life cycle,” *arXiv preprint arXiv:1205.6904*, 2012.

[26] F. Brooks and H. Kugler, *No silver bullet.* April, 1987.

[27] M. Dawson, D. N. Burrell, E. Rahim, and S. Brewster, “Integrating software assurance into the software development life cycle (sdlc),” *Journal of Information Systems Technology and Planning*, vol. 3, no. 6, pp. 49–53, 2010.

[28] R. Soley *et al.*, “Model driven architecture,” *OMG white paper*, vol. 308, no. 308, p. 5, 2000.

[29] D. C. Schmidt *et al.*, “Model-driven engineering,” *Computer-IEEE Computer Society-*, vol. 39, no. 2, p. 25, 2006.

[30] J.-M. Jézéquel, "Taming variability in software engineering: Past, present & future," *MDENet Annual Symposium*, 2022.

[31] P. D. Marinescu and C. Cadar, "make test-zesti: A symbolic execution solution for improving regression testing," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 716–726.

[32] W. Krenn, R. Schlick, S. Tiran, B. Aichernig, E. Jobstl, and H. Brandl, "Momut:: UML model-based mutation testing for UML," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2015, pp. 1–8.

[33] B. Uzun and B. Tekinerdogan, "Model-driven architecture based testing: A systematic literature review," *Information and Software technology*, vol. 102, pp. 30–48, 2018.

[34] (2022) Eclipse papyrus. [Online]. Available: https://www.eclipse.org/papyrus/

[35] (2022) Umbrello project. [Online]. Available: https://umbrello.kde.org

[36] (2022) AgileUML repository. [Online]. Available: https://github.com/eclipse/agileuml/

[37] L. Briand and Y. Labiche, "A UML-based approach to system testing," *Software and systems modeling*, vol. 1, no. 1, pp. 10–42, 2002.

[38] Y. D. Salman and N. L. Hashim, "Automatic test case generation from UML state chart diagram: a survey," in *Advanced Computer and Communication Engineering Technology*. Springer, 2016, pp. 123–134.

[39] W. Linzhang, Y. Jiesong, Y. Xiaofeng, H. Jun, L. Xuandong, and Z. Guoliang, "Generating test cases from UML activity diagram based on gray-box method," in *11th Asia-Pacific software engineering conference.* IEEE, 2004, pp. 284–291.

[40] D. Bork, D. Karagiannis, and B. Pittl, "A survey of modeling language specification techniques," *Information Systems*, vol. 87, p. 101425, 2020.

[41] J. Cabot, "The ultimate object constraint language (OCL) tutorial," May 2020. [Online]. Available: https://modeling-languages.com/ocl-tutorial/

[42] S. Weißleder and D. Sokenou, "Automatic test case generation from UML models and OCL expressions," *Software Engineering 2008*, 2008.

[43] M. Richters and M. Gogolla, "OCL: Syntax, semantics, and tools," in *Object Modeling with the OCL.* Springer, 2002, pp. 42–68.

[44] E. Willink, "An OCL map type," in *OCL Workshops*, 2019.

[45] M. Gogolla, L. Burgueño, and A. Vallecillo, "Refactoring collections in OCL." in *STAF Workshops*, 2021, pp. 142–148.

[46] K. Lano, "Adding regular expression operators to OCL." in *STAF Workshops*, 2021, pp. 162–168.

[47] J. Warmer and K. Objecten, "The future of UML," *OMG Information Day, Amsterdam*, p. 31, 2001.

[48] E. Willink, "Reflections on OCL 2." *J. Object Technol.*, vol. 19, no. 3, pp. 3–1, 2020.

[49] A. Maraee and A. Sturm, "The usage of constraint specification languages: a controlled experiment," in *Enterprise, Business-Process and Information Systems Modeling: 20th International Conference, BPMDS 2019, 24th International Conference, EMMSAD 2019, Held at CAiSE 2019, Rome, Italy, June 3–4, 2019, Proceedings 20.* Springer, 2019, pp. 329–343.

[50] N. Khurana and R. Chillar, "Test case generation and optimization using UML models and genetic algorithm," *Procedia Computer Science*, vol. 57, pp. 996–1004, 2015.

[51] S. Achour and M. Benattou, "A model based testing approach for java bytecode programs." *J. Comput.*, vol. 13, no. 9, pp. 1098–1114, 2018.

[52] C. Pérez and B. Marín, "Automatic generation of test cases from UML models," *CLEI Electron. J*, vol. 21, no. 1, 2018.

[53] S. A. Slaughter, D. E. Harter, and M. S. Krishnan, "Evaluating the cost of software quality," *Communications of the ACM*, vol. 41, no. 8, pp. 67–73, 1998.

[54] A. Hartman, M. Katara, and S. Olvovsky, "Choosing a test modeling language: A survey," in *Haifa Verification Conference.* Springer, 2007, pp. 204–218.

[55] A. Hussain, S. Tiwari, J. Suryadevara, and E. Enoiu, "From modeling to test case generation in the industrial embedded system domain," in *Federation of International Conferences on Software Technologies: Applications and Foundations.* Springer, 2018, pp. 499–505.

[56] M. Shafique and Y. Labiche, “A systematic review of model based testing tool support,” *Carleton University, Canada, Tech. Rep. Technical Report SCE-10-04*, pp. 01–21, 2010.

[57] C. M. Gerpheide, R. R. Schiffelers, and A. Serebrenik, “Assessing and improving quality of qvto model transformations,” *Software Quality Journal*, vol. 24, no. 3, pp. 797–834, 2016.

[58] Y. G. Kim, H. S. Hong, D.-H. Bae, and S. D. Cha, “Test cases generation from UML state diagrams,” *IEE Proceedings-Software*, vol. 146, no. 4, pp. 187–192, 1999.

[59] P. Fröhlich and J. Link, “Automated test case generation from dynamic models,” in *European Conference on Object-Oriented Programming.* Springer, 2000, pp. 472–491.

[60] P. Chevalley and P. Thevenod-Fosse, “Automated generation of statistical test cases from UML state diagrams,” in *25th Annual International Computer Software and Applications Conference. COMPSAC 2001.* IEEE, 2001, pp. 205–214.

[61] D. Seifert, S. Helke, and T. Santen, “Test case generation for UML statecharts,” in *International Andrei Ershov Memorial Conference on Perspectives of System Informatics.* Springer, 2003, pp. 462–468.

[62] M. Riebisch, I. Philippow, and M. Götze, “UML-based statistical test case generation,” in *Objects, Components, Architectures, Services, and Applications for a Networked World: International Conference NetObjectDays, NODe 2002 Erfurt, Germany, October 7–10, 2002 Revised Papers 4.* Springer, 2003, pp. 394–411.

[63] A. Cavarra, C. Crichton, and J. Davies, "A method for the automatic generation of test suites from object models," in *Proceedings of the 2003 ACM symposium on Applied computing*, 2003, pp. 1104–1109.

[64] S. Gnesi, D. Latella, and M. Massink, "Formal test-case generation for UML statecharts," in *Proceedings. Ninth IEEE International Conference on Engineering of Complex Computer Systems.* IEEE, 2004, pp. 75–84.

[65] P. Samuel and R. Mall, "Boundary value testing based on UML models," in *14th Asian Test Symposium (ATS'05).* IEEE, 2005, pp. 94–99.

[66] D. Buchs, L. Pedro, and L. Lúcio, "Formal test generation from UML models," in *Dependable Systems: Software, Computing, Networks.* Springer, 2006, pp. 145–171.

[67] H. Kim, S. Kang, J. Baik, and I. Ko, "Test cases generation from UML activity diagrams," in *Eighth ACIS international conference on software engineering, artificial intelligence, networking, and parallel/distributed computing (SNPD 2007)*, vol. 3. IEEE, 2007, pp. 556–561.

[68] P. Samuel, R. Mall, and P. Kanth, "Automatic test case generation from UML communication diagrams," *Information and software technology*, vol. 49, no. 2, pp. 158–171, 2007.

[69] S. Ali, L. C. Briand, M. J.-u. Rehman, H. Asghar, M. Z. Z. Iqbal, and A. Nadeem, "A state-based approach to integration testing based on UML models,"

Information and Software Technology, vol. 49, no. 11-12, pp. 1087–1106, 2007.

[70] M. Sarma, D. Kundu, and R. Mall, “Automatic test case generation from UML sequence diagram,” in *15th International Conference on Advanced Computing and Communications (ADCOM 2007)*. IEEE, 2007, pp. 60–67.

[71] M. Friske and B.-H. Schlingloff, “Improving test coverage for UML state machines using transition instrumentation,” in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2007, pp. 301–314.

[72] M. Sarma and R. Mall, “Automatic test case generation from UML models,” in *10th International Conference on Information Technology (ICIT 2007)*. IEEE, 2007, pp. 196–201.

[73] B.-L. Li, Z.-s. Li, L. Qing, and Y.-H. Chen, “Test case automate generation from UML sequence diagram and OCL expression,” in *2007 international conference on computational intelligence and security (cis 2007)*. IEEE, 2007, pp. 1048–1052.

[74] E. G. Cartaxo, F. G. Neto, and P. D. Machado, “Test case generation by means of UML sequence diagrams and labeled transition systems,” in *2007 IEEE International Conference on Systems, Man and Cybernetics*. IEEE, 2007, pp. 1292–1297.

[75] P. Samuel, R. Mall, and A. K. Bothra, “Automatic test case generation using unified modeling language (UML) state diagrams,” *IET software*, vol. 2, no. 2, pp. 79–93, 2008.

[76] S. Weißleder and B.-H. Schlingloff, “Deriving input partitions from UML models for automatic test generation,” in *Models in Software Engineering: Workshops and Symposia at MoDELS 2007, Nashville, TN, USA, September 30-October 5, 2007, Reports and Revised Selected Papers 10*. Springer, 2008, pp. 151–163.

[77] M. Sarma and R. Mall, “Automatic generation of test specifications for coverage of system state transitions,” *Information and Software Technology*, vol. 51, no. 2, pp. 418–432, 2009.

[78] T. Clark, “Model based functional testing using pattern directed filmstrips,” in *2009 ICSE Workshop on Automation of Software Test*. IEEE, 2009, pp. 53–61.

[79] F. Zeng, Z. Chen, Q. Cao, and L. Mao, “Research on method of object-oriented test cases generation based on UML and lts,” in *2009 First International Conference on Information Science and Engineering*. IEEE, 2009, pp. 5055–5058.

[80] M. Chen, X. Qiu, W. Xu, L. Wang, J. Zhao, and X. Li, “UML activity diagram-based automatic test case generation for java programs,” *The Computer Journal*, vol. 52, no. 5, pp. 545–556, 2009.

[81] X. Fan, J. Shu, L. Liu, and Q. Liang, “Test case generation from UML subactivity and activity diagram,” in *2009 Second International Symposium on Electronic Commerce and Security*, vol. 2. IEEE, 2009, pp. 244–248.

[82] G. Batra, Y. K. Arora, and J. Sengupta, “Model-based software regression testing for software components,”

in *International Conference on Information Systems, Technology and Management.* Springer, 2009, pp. 138–149.

[83] S. Asthana, S. Tripathi, and S. K. Singh, "A novel approach to generate test cases using class and sequence diagrams," in *International Conference on Contemporary Computing.* Springer, 2010, pp. 155–167.

[84] M. Shirole and R. Kumar, "A hybrid genetic algorithm based test case generation using sequence diagrams," in *International Conference on Contemporary Computing.* Springer, 2010, pp. 53–63.

[85] C. Schwarzl and B. Peischl, "Generation of executable test cases based on behavioral UML system models," in *Proceedings of the 5th Workshop on Automation of Software Test*, 2010, pp. 31–34.

[86] M. Khandai, A. A. Acharya, and D. P. Mohapatra, "A novel approach of test case generation for concurrent systems using UML sequence diagram," in *2011 3rd International Conference on Electronics Computer Technology*, vol. 1. IEEE, 2011, pp. 157–161.

[87] S. Ali, M. Z. Iqbal, A. Arcuri, and L. Briand, "A search-based OCL constraint solver for model-based test data generation," in *2011 11th International Conference on Quality Software.* IEEE, 2011, pp. 41–50.

[88] A. D. Brucker, M. P. Krieger, D. Longuet, and B. Wolff, "A specification-based test case generation method for UML/OCL," in *Models in Software Engineering: Workshops and Symposia at MODELS 2010, Oslo, Norway, October 2-8, 2010, Reports and Revised Selected Papers 13.* Springer, 2011, pp. 334–348.

[89] M. Prasanna and K. Chandran, “Automated test case generation for object oriented systems using UML object diagrams,” in *International Conference on High Performance Architecture and Grid Computing.* Springer, 2011, pp. 417–423.

[90] A. Gantait, “Test case generation and prioritization from UML models,” in *2011 Second International Conference on Emerging Applications of Information Technology.* IEEE, 2011, pp. 345–350.

[91] M. Prasanna, K. R. Chandran, and K. Thiruvenkadam, “Automatic test case generation for UML collaboration diagrams,” *IETE Journal of research*, vol. 57, no. 1, pp. 77–81, 2011.

[92] A. Nayak and D. Samanta, “Synthesis of test scenarios using UML activity diagrams,” *Software & Systems Modeling*, vol. 10, no. 1, pp. 63–89, 2011.

[93] X. Chen, N. Ye, P. Jiang, L. Bu, and X. Li, “Feedback-directed test case generation based on UML activity diagrams,” in *2011 Fifth International Conference on Secure Software Integration and Reliability Improvement-Companion.* IEEE, 2011, pp. 9–10.

[94] V. Sawant and K. Shah, “Construction of test cases from UML models,” in *Technology Systems and Management.* Springer, 2011, pp. 61–68.

[95] L. Briand, Y. Labiche, and Y. Liu, “Combining UML sequence and state machine diagrams for data-flow based integration testing,” in *European Conference on Modelling Foundations and Applications.* Springer, 2012, pp. 74–89.

[96] R. K. Swain, V. Panthi, and P. K. Behera, "Test case design using slicing of UML interaction diagram," *Procedia Technology*, vol. 6, pp. 136–144, 2012.

[97] K. Pechtanun and S. Kansomkeat, "Generation test case from UML activity diagram based on ac grammar," in *2012 International Conference on Computer & Information Science (ICCIS)*, vol. 2. IEEE, 2012, pp. 895–899.

[98] L. Li, X. Li, T. He, and J. Xiong, "Extenics-based test case generation for UML activity diagram," *Procedia Computer Science*, vol. 17, pp. 1186–1193, 2013.

[99] C.-S. Wu and C.-H. Huang, "The web services composition testing based on extended finite state machine and UML model," in *2013 Fifth International Conference on Service Science and Innovation.* IEEE, 2013, pp. 215–222.

[100] V. Chimisliu and F. Wotawa, "Improving test case generation from UML statecharts by using control, data and communication dependencies," in *2013 13th International Conference on Quality Software.* IEEE, 2013, pp. 125–134.

[101] R. Anbunathan and A. Basu, "Dataflow test case generation from UML class diagrams," in *2013 IEEE International Conference on Computational Intelligence and Computing Research.* IEEE, 2013, pp. 1–9.

[102] F. Kurth, S. Schupp, and S. Weißleder, "Generating test data from a UML activity using the ampl interface for constraint solvers," in *International Conference on Tests and Proofs.* Springer, 2014, pp. 169–186.

[103] Y. Li and L. Jiang, "The research on test case generation technology of UML sequence diagram," in *2014 9th International Conference on Computer Science & Education*. IEEE, 2014, pp. 1067–1069.

[104] E. Fourneret, J. Cantenot, F. Bouquet, B. Legeard, and J. Botella, "Setgam: Generalized technique for regression testing based on UML/OCL models," in *2014 Eighth International Conference on Software Security and Reliability (SERE)*. IEEE, 2014, pp. 147–156.

[105] A. K. Jena, S. K. Swain, and D. P. Mohapatra, "A novel approach for test case generation from UML activity diagram," in *2014 International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT)*. IEEE, 2014, pp. 621–629.

[106] M. Bilal, N. Sarwar, and M. S. Saeed, "A hybrid test case model for medium scale web based applications," in *2016 Sixth International Conference on Innovative Computing Technology (INTECH)*. IEEE, 2016, pp. 632–637.

[107] M. Elallaoui, K. Nafil, R. Touahni, and R. Messoussi, "Automated model driven testing using andromda and UML2 testing profile in scrum process," *Procedia Computer Science*, vol. 83, pp. 221–228, 2016.

[108] P. Mahali, S. Arabinda, A. A. Acharya, and D. P. Mohapatra, "Test case generation for concurrent systems using UML activity diagram," in *2016 IEEE Region 10 Conference (TENCON)*. IEEE, 2016, pp. 428–435.

[109] I. S. Meiliana, R. S. Alianto, G. F. Daniel *et al.*, "Automated test case generation from UML activity dia-

gram and sequence diagram using depth first search algorithm," *Procedia Comput Sci*, vol. 116, p. 629â, 2017.

[110] V. Arora, R. Bhatia, and M. Singh, "Synthesizing test scenarios in UML activity diagram using a bio-inspired approach," *Computer Languages, Systems & Structures*, vol. 50, pp. 1–19, 2017.

[111] Y. D. Salman, N. L. Hashim, M. M. Rejab, R. Romli, and H. Mohd, "Coverage criteria for test case generation using UML state chart diagram," in *AIP Conference Proceedings*, vol. 1891, no. 1. AIP Publishing LLC, 2017, p. 020125.

[112] P. Mani and M. Prasanna, "Test case generation for embedded system software using UML interaction diagram," *Journal of Engineering Science and Technology*, vol. 12, no. 4, pp. 860–874, 2017.

[113] V. Panthi and D. P. Mohapatra, "Generating and evaluating effectiveness of test sequences using state machine," *International Journal of System Assurance Engineering and Management*, vol. 8, no. 2, pp. 242–252, 2017.

[114] P. K. Arora and R. Bhatia, "Agent-based regression test case generation using class diagram, use cases and activity diagram," *Procedia Computer Science*, vol. 125, pp. 747–753, 2018.

[115] S. Kamonsantiroj, L. Pipanmaekaporn, and S. Lorpunmanee, "A memorization approach for test case generation in concurrent UML activity diagram," in *Proceedings of the 2019 2nd International Conference on Geoinformatics and Data Analysis*, 2019, pp. 20–25.

[116] H. Sartaj, M. Z. Iqbal, A. A. A. Jilani, and M. U. Khan, “A search-based approach to generate mc/dc test data for OCL constraints,” in *International Symposium on Search Based Software Engineering.* Springer, 2019, pp. 105–120.

[117] S. K. Barisal, S. S. Behera, S. Godboley, and D. P. Mohapatra, “Validating object-oriented software at design phase by achieving mc/dc,” *International Journal of System Assurance Engineering and Management*, vol. 10, no. 4, pp. 811–823, 2019.

[118] S. Pradhan, M. Ray, and S. K. Swain, “Transition coverage based test case generation from state chart diagram,” *Journal of King Saud University-Computer and Information Sciences*, 2019.

[119] W. Y. Kim, H. S. Son, and R. Y. C. Kim, “A study on test case generation based on state diagram in modeling and simulation environment,” in *Advanced Communication and Networking: Third International Conference, ACN 2011, Brno, Czech Republic, August 15-17, 2011. Proceedings.* Springer, 2011, pp. 298–305.

[120] R. K. Sahoo, S. K. Nanda, D. P. Mohapatra, and M. R. Patra, “Model driven test case optimization of UML combinational diagrams using hybrid bee colony algorithm,” *International Journal of Intelligent Systems and Applications*, vol. 9, no. 6, p. 43, 2017.

[121] A. Bader, A. S. M. Sajeev, and S. Ramakrishnan, “Testing concurrency and communication in distributed objects,” in *Proceedings. Fifth International Conference on High Performance Computing (Cat. No. 98EX238).* IEEE, 1998, pp. 422–428.

[122] M. R. Woodward and M. A. Hennell, "On the relationship between two control-flow coverage criteria: all jj-paths and mcdc," *Information and Software Technology*, vol. 48, no. 7, pp. 433–440, 2006.

[123] P. Murthy, P. Anitha, M. Mahesh, and R. Subramanyan, "Test ready UML statechart models," in *Proceedings of the 2006 international workshop on Scenarios and state machines: models, algorithms, and tools*, 2006, pp. 75–82.

[124] M. G. Epitropakis, S. Yoo, M. Harman, and E. K. Burke, "Empirical evaluation of pareto efficient multi-objective regression test case prioritisation," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015, pp. 234–245.

[125] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on software engineering*, vol. 27, no. 10, pp. 929–948, 2001.

[126] J. F. Silva Ouriques, E. G. Cartaxo, and P. D. Lima Machado, "Revealing influence of model structure and test case profile on the prioritization of test cases in the context of model-based testing," *Journal of Software Engineering Research and Development*, vol. 3, no. 1, pp. 1–28, 2015.

[127] S. Li, N. Bian, Z. Chen, D. You, and Y. He, "A simulation study on some search algorithms for regression test case prioritization," in *2010 10th International Conference on Quality Software.* IEEE, 2010, pp. 72–81.

[128] H. Srikanth, L. Williams, and J. Osborne, "System test case prioritization of new and regression test cases," in

2005 International Symposium on Empirical Software Engineering, 2005. IEEE, 2005, pp. 10–pp.

[129] J.-M. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in *Proceedings of the 24th international conference on software engineering*, 2002, pp. 119–129.

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

[131] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: An empirical study," in *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360).* IEEE, 1999, pp. 179–188.

[132] S. Elbaum, A. Malishevsky, and G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization," in *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001.* IEEE, 2001, pp. 329–338.

[133] E. J. Rapos and J. Dingel, "Using fuzzy logic and symbolic execution to prioritize UML-RT test cases," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST).* IEEE, 2015, pp. 1–10.

[134] K.-W. Shin and D.-J. Lim, "Model-based test case prioritization using an alternating variable method for re-

gression testing of a UML-based model," *Applied Sciences*, vol. 10, no. 21, p. 7537, 2020.

[135] T. Pospisil, J. Sobotka, and J. Novak, "Enhanced adaptive random test case prioritization for model-based test suites," *Acta Polytechnica Hungarica*, vol. 17, no. 7, 2020.

[136] R. Pan, M. Bagherzadeh, T. A. Ghaleb, and L. Briand, "Test case selection and prioritization using machine learning: a systematic literature review," *Empirical Software Engineering*, vol. 27, no. 2, pp. 1–43, 2022.

[137] B. Ma, L. Wan, N. Yao, S. Fan, and Y. Zhang, "Evolutionary selection for regression test cases based on diversity," *Frontiers of Computer Science*, vol. 15, no. 2, pp. 1–3, 2021.

[138] P. Rattan, M. Arora, M. Rakhra, V. Goel *et al.*, "A neoteric approach of prioritizing regression test suites using hybrid esdg models," *Annals of the Romanian Society for Cell Biology*, pp. 2965–2973, 2021.

[139] W. Sornkliang and T. Phetkaew, "Target-based test path prioritization for UML activity diagram using weight assignment methods," *International Journal of Electrical and Computer Engineering*, vol. 11, no. 1, p. 575, 2021.

[140] S. Chaudhary and A. Jatain, "Performance evaluation of clustering techniques in test case prioritization," in *2020 International Conference on Computational Performance Evaluation (ComPE).* IEEE, 2020, pp. 699–703.

[141] A. Morozov, K. Ding, T. Chen, and K. Janschek, “Test suite prioritization for efficient regression testing of model-based automotive software,” in *2017 International Conference on Software Analysis, Testing and Evolution (SATE)*. IEEE, 2017, pp. 20–29.

[142] Y. Xing, X. Wang, and Q. Shen, “Test case prioritization based on artificial fish school algorithm,” *Computer Communications*, vol. 180, pp. 295–302, 2021.

[143] M. Mann, P. Tomar, and O. P. Sangwan, “Bio-inspired metaheuristics: evolving and prioritizing software test data,” *Applied Intelligence*, vol. 48, pp. 687–702, 2018.

[144] B. Miranda, E. Cruciani, R. Verdecchia, and A. Bertolino, “Fast approaches to scalable similarity-based test case prioritization,” in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 222–232.

[145] F. Li, J. Zhou, Y. Li, D. Hao, and L. Zhang, “Aga: An accelerated greedy additional algorithm for test case prioritization,” *IEEE Transactions on Software Engineering*, vol. 48, no. 12, pp. 5102–5119, 2021.

[146] C.-a. Sun, B. Liu, A. Fu, Y. Liu, and H. Liu, “Path-directed source test case generation and prioritization in metamorphic testing,” *Journal of Systems and Software*, vol. 183, p. 111091, 2022.

[147] A. Bajaj and O. P. Sangwan, “Study the impact of parameter settings and operators role for genetic algorithm based test case prioritization,” in *Proceedings of International Conference on Sustainable Computing in Science, Technology and Management (SUSCOM), Amity University Rajasthan, Jaipur-India*, 2019.

[148] Z. Li, M. Harman, and R. M. Hierons, “Search algorithms for regression test case prioritization,” *IEEE Transactions on software engineering*, vol. 33, no. 4, pp. 225–237, 2007.

[149] D. Whitley, “A genetic algorithm tutorial,” *Statistics and computing*, vol. 4, pp. 65–85, 1994.

[150] R. Eberhart and J. Kennedy, “A new optimizer using particle swarm theory,” in *MHS’95. Proceedings of the sixth international symposium on micro machine and human science.* Ieee, 1995, pp. 39–43.

[151] X.-S. Yang, “Firefly algorithms for multimodal optimization,” in *Stochastic Algorithms: Foundations and Applications: 5th International Symposium, SAGA 2009, Sapporo, Japan, October 26-28, 2009. Proceedings 5.* Springer, 2009, pp. 169–178.

[152] C. J. Bastos Filho, F. B. de Lima Neto, A. J. Lins, A. I. Nascimento, and M. P. Lima, “A novel search algorithm based on fish school behavior,” in *2008 IEEE international conference on systems, man and cybernetics.* IEEE, 2008, pp. 2646–2651.

[153] X.-S. Yang and S. Deb, “Cuckoo search via lévy flights,” in *2009 World congress on nature & biologically inspired computing (NaBIC).* Ieee, 2009, pp. 210–214.

[154] T. Y. Chen and M. F. Lau, “A new heuristic for test suite reduction,” *Information and Software Technology*, vol. 40, no. 5-6, pp. 347–354, 1998.

[155] ——, “On the divide-and-conquer approach towards test suite reduction,” *Information sciences*, vol. 152, pp. 89–119, 2003.

[156] S. Tallam and N. Gupta, "A concept analysis inspired greedy algorithm for test suite minimization," *ACM SIGSOFT Software Engineering Notes*, vol. 31, no. 1, pp. 35–42, 2005.

[157] S. Nachiyappan, A. Vimaladevi, and C. SelvaLakshmi, "An evolutionary algorithm for regression test suite reduction," in *2010 International Conference on Communication and Computational Intelligence (INCOCCI)*. IEEE, 2010, pp. 503–508.

[158] Y.-k. Zhang, J.-c. Liu, Y.-a. Cui, X.-h. Hei, and M.-h. Zhang, "An improved quantum genetic algorithm for test suite reduction," in *2011 IEEE International Conference on Computer Science and Automation Engineering*, vol. 2. IEEE, 2011, pp. 149–153.

[159] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur, "Effect of test set minimization on fault detection effectiveness," in *Proceedings of the 17th international conference on Software engineering*, 1995, pp. 41–50.

[160] F. Palomo-Lozano, A. Estero-Botaro, I. Medina-Bulo, and M. Núñez, "Test suite minimization for mutation testing of ws-bpel compositions," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2018, pp. 1427–1434.

[161] P. K. Bhatia *et al.*, "Test case minimization in cots methodology using genetic algorithm: a modified approach," in *Proceedings of ICETIT 2019*. Springer, 2020, pp. 219–228.

[162] X. Lin, H. Zhang, H. Xia, L. Yu, X. Fang, X. Chen, and Z. Wang, "Test case minimization for regression testing

of composite service based on modification impact analysis," in *International Conference on Web Information Systems and Applications.* Springer, 2020, pp. 15–26.

[163] N. L. Hashim and Y. S. Dawood, "Test case minimization applying firefly algorithm," *International Journal on Advanced Science, Engineering and Information Technology*, vol. 8, no. 4-2, pp. 1777–1783, 2018.

[164] A. Deneke, B. G. Assefa, and S. K. Mohapatra, "Test suite minimization using particle swarm optimization," *Materials Today: Proceedings*, vol. 60, pp. 229–233, 2022.

[165] L. Li, Y. Zhou, Y. Yuan, and S. Wu, "An extensive study on multi-priority algorithm in test case prioritization and reduction," in *2021 2nd Asia Service Sciences and Software Engineering Conference*, 2021, pp. 48–57.

[166] A. Bajaj, A. Abraham, S. Ratnoo, and L. A. Gabralla, "Test case prioritization, selection, and reduction using improved quantum-behaved particle swarm optimization," *Sensors*, vol. 22, no. 12, p. 4374, 2022.

[167] R. Huang, H. Chen, W. Sun, and D. Towey, "Candidate test set reduction for adaptive random testing: An overheads reduction technique," *Science of Computer Programming*, vol. 214, p. 102730, 2022.

[168] N. Gupta, A. Sharma, and M. K. Pachariya, "Multi-objective test suite optimization for detection and localization of software faults," *Journal of King Saud University-Computer and Information Sciences*, vol. 34, no. 6, pp. 2897–2909, 2022.

[169] A. J. Turner, D. R. White, and J. H. Drake, "Multi-objective regression test suite minimisation for mockito," in *Search Based Software Engineering: 8th International Symposium, SSBSE 2016, Raleigh, NC, USA, October 8-10, 2016, Proceedings 8.* Springer, 2016, pp. 244–249.

[170] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE transactions on evolutionary computation*, vol. 6, no. 2, pp. 182–197, 2002.

[171] Q. Zhang and H. Li, "Moea/d: A multiobjective evolutionary algorithm based on decomposition," *IEEE Transactions on evolutionary computation*, vol. 11, no. 6, pp. 712–731, 2007.

[172] E. Zitzler, M. Laumanns, and L. Thiele, "Spea2: Improving the strength pareto evolutionary algorithm," *TIK-report*, vol. 103, 2001.

[173] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. A. London, "Incremental regression testing," in *1993 Conference on Software Maintenance.* IEEE, 1993, pp. 348–357.

[174] F. I. Vokolos and P. G. Frankl, "Pythia: A regression test selection tool based on textual differencing," in *Reliability, Quality and Safety of Software-Intensive Systems: IFIP TC5 WG5. 4 3rd International Conference on Reliability, Quality and Safety of Software-Intensive Systems (ENCRESS'97), 29th–30th May 1997, Athens, Greece.* Springer, 1997, pp. 3–21.

[175] S. Chen, Z. Chen, Z. Zhao, B. Xu, and Y. Feng, "Using semi-supervised clustering to improve regression test se-

lection techniques," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation.* IEEE, 2011, pp. 1–10.

[176] L. C. Briand, Y. Labiche, and S. He, "Automating regression test selection based on UML designs," *Information and Software Technology*, vol. 51, no. 1, pp. 16–30, 2009.

[177] M. Al-Refai, "Towards model-based regression test selection," Ph.D. dissertation, Colorado State University, 2019.

[178] B. Alkhazi, C. Abid, M. Kessentini, D. Leroy, and M. Wimmer, "Multi-criteria test cases selection for model transformations," *Automated Software Engineering*, vol. 27, pp. 91–118, 2020.

[179] V. Dorcis, F. Bouquet, and F. Dadeau, "Clustering of usage traces for regression test cases selection," in *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW).* IEEE, 2022, pp. 138–145.

[180] G. Guizzo, J. Petke, F. Sarro, and M. Harman, "Enhancing genetic improvement of software with regression test selection," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE).* IEEE, 2021, pp. 1323–1333.

[181] D. A. d'Aragona, F. Pecorelli, S. Romano, G. Scanniello, M. T. Baldassarre, A. Janes, and V. Lenarduzzi, "Catto: Just-in-time test case selection and execution," in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME).* IEEE, 2022, pp. 459–463.

[182] A. Arrieta, P. Valle, J. A. Agirre, and G. Sagardui, "Some seeds are strong: Seeding strategies for search-based test case selection," *ACM Transactions on Software Engineering and Methodology*, 2022.

[183] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, "Mutation testing advances: an analysis and survey," in *Advances in Computers*. Elsevier, 2019, vol. 112, pp. 275–378.

[184] S. Dalal, K. Solanki *et al.*, "Challenges of regression testing: A pragmatic perspective." *International Journal of Advanced Research in Computer Science*, vol. 9, no. 1, 2018.

[185] L. Gutiérrez-Madronal, J. J. Domınguez-Jiménez, and I. Medina-Bulo, "Mutation testing: Guideline and mutation operator classification," *ICCGI 2014*, p. 184, 2014.

[186] E. J. Weyuker, "On testing non-testable programs," *The Computer Journal*, vol. 25, no. 4, pp. 465–470, 1982.

[187] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE transactions on software engineering*, vol. 37, no. 5, pp. 649–678, 2010.

[188] A. S. Ghiduk, M. R. Girgis, and M. H. Shehata, "Higher order mutation testing: A systematic literature review," *Computer Science Review*, vol. 25, pp. 29–48, 2017.

[189] L. Ma, F. Zhang, J. Sun, M. Xue, B. Li, F. Juefei-Xu, C. Xie, L. Li, Y. Liu, J. Zhao *et al.*, "Deepmutation: Mutation testing of deep learning systems," in

2018 IEEE 29th international symposium on software reliability engineering (ISSRE). IEEE, 2018, pp. 100–111.

[190] K. Lano and S. Kolahdouz-Rahimi, "Extending OCL with map and function types," in *International Conference on Fundamentals of Software Engineering*. Springer, 2021, pp. 108–123.

[191] J. Strug, "Classification of mutation operators applied to design models," in *Key Engineering Materials*, vol. 572. Trans Tech Publ, 2014, pp. 539–542.

[192] M. F. Granda, N. Condori-Fernández, T. E. Vos, and O. Pastor, "Mutation operators for UML class diagrams," in *International Conference on Advanced Information Systems Engineering*. Springer, 2016, pp. 325–341.

[193] Z. Ahmed, M. Zahoor, and I. Younas, "Mutation operators for object-oriented systems: A survey," in *2010 The 2nd International Conference on Computer and Automation Engineering (ICCAE)*, vol. 2. IEEE, 2010, pp. 614–618.

[194] J. Hassine, "Design and classification of mutation operators for abstract state machines," *Int. J. Adv. Softw*, vol. 6, no. 1, pp. 80–91, 2013.

[195] L. C. Ascari and S. R. Vergilio, "Mutation testing based on OCL specifications and aspect oriented programming," in *2010 XXIX International Conference of the Chilean Computer Science Society*. IEEE, 2010, pp. 43–50.

[196] A. Ali, H. A. Maghawry, and N. Badr, "Model-based test case generation approach for mobile applications load testing using OCL enhanced activity diagrams," in *2021 Tenth International Conference on Intelligent Computing and Information Systems (ICICIS)*. IEEE, 2021, pp. 493–499.

[197] K. Jin and K. Lano, "Mutation operators for object constraint language specification," in *OCL Workshops*, 2021, pp. 128–134.

[198] Q. Zhu, A. Panichella, and A. Zaidman, "A systematic literature review of how mutation testing supports quality assurance processes," *Software Testing, Verification and Reliability*, vol. 28, no. 6, p. e1675, 2018.

[199] S. Sualim, R. Mohamad, and N. A. Saadon, "Ontology of mutation testing for java operators," *International Journal of Innovative Computing*, vol. 8, no. 2, 2018.

[200] Y.-S. Ma, Y.-R. Kwon, and J. Offutt, "Inter-class mutation operators for java," in *13th International Symposium on Software Reliability Engineering, 2002. Proceedings.* IEEE, 2002, pp. 352–363.

[201] A. J. Offutt and J. Pan, "Detecting equivalent mutants and the feasible path problem," in *Proceedings of 11th Annual Conference on Computer Assurance. COMPASS'96.* IEEE, 1996, pp. 224–236.

[202] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, "Timeaware test suite prioritization," in *Proceedings of the 2006 international symposium on Software testing and analysis*, 2006, pp. 1–12.

[203] M. Li and X. Yao, "Quality evaluation of solution sets in multiobjective optimisation: A survey," *ACM Computing Surveys (CSUR)*, vol. 52, no. 2, pp. 1–38, 2019.

[204] J. J. Durillo and A. J. Nebro, "jmetal: A java framework for multi-objective optimization," *Advances in Engineering Software*, vol. 42, no. 10, pp. 760–771, 2011.

[205] M. Egea and V. Rusu, "Formal executable semantics for conformance in the mde framework," *Innovations in Systems and Software Engineering*, vol. 6, pp. 73–81, 2010.

[206] F. Büttner and M. Gogolla, "On OCL-based imperative languages," *Science of Computer Programming*, vol. 92, pp. 162–178, 2014.

[207] A. D. Brucker, F. Tuong, and B. Wolff, "Featherweight OCL: A proposal for a machine-checked formal semantics for OCL 2.5," in *15th International Workshop on OCL and Textual Modeling co-located with 18th International Conference on Model Driven Engineering Languages and Systems*, 2015, p. 199.

Appendix A

Review Details

In this appendix, detailed reviews of each primary study are presented.

- *Primary Study 1*: This study uses the state diagram to perform the test case generation process. The proposed approach converts the UML state diagram to extended finite state machines and then transfers them to a flow graph. Based on data flow coverage criteria to create the test cases. The reason for not applying the path coverage criteria is that the infeasible path in the flow graph is difficult to estimate.
- *Primary Study 2*: In the previous work of the author, the process of transforming the use case diagram into the state diagram was completed. This study uses the state diagram to generate test cases. The STRIPS technology in AI Planning is used in the whole generation process. The state diagram will be converted into a specific format that can be used in STRIPS to generate test cases. During test case generation, branch coverage guides the whole process and ensures that all pre- and post- conditions are satisfied.
- *Primary Study 3*: This study uses the statistical testing technique to automatically generate test cases from the UML state diagram without using any intermediate format. The proposed approach can be used for test case generation of Java programs and satisfy transition coverage criteria.

Based on the coverage criteria, the proposed algorithm generates different inputs to execute transitions randomly and tries to cover every transition. Finally, the mutation test is used to compare the proposed method with data, which are completely randomly generated, showing certain advantages.

- *Primary Study 4*: In this study, the state diagram is transformed into a common semantic expression, labelled transition system. The main contribution is to define a proper method to represent the semantic interpretation of the UML state machine diagram. A FIFO (first in first out) queue is used to describe the different abstract input parameters that stand for the generated test cases. For parallel events, using multiple test cases to show all the possible combinations of events, although a tremendously large number of test cases may be produced.

- *Primary Study 5*: The proposed approach uses the use case diagram to generate test cases. The main approach is to refine the use cases and convert them into state machine diagrams through model transformations. The state diagram is transformed into the usage model by usage graph. This step requires the involvement of the customer or expert to determine the probability of an event. Finally, the usage model is used to generate test cases. In the generation process, the minimum arc coverage, equivalent to transition coverage, is guaranteed to be implemented. Then test cases in a particular order are randomly generated to form the test suite.

- *Primary Study 6*: The proposed method uses class diagrams, state diagrams, and object diagrams to determine the existing behaviours in the system. The method converts these graphs to an intermediate format in the IF language that describes the communication state machine. The test cases are then automatically generated by using the external

TGV tool. The generated test cases satisfy all the possible values in the IF expression, and the test cases are created that enable the expression to obtain acceptable results.

- *Primary Study 7*: This study used the UML state machine diagram to generate test cases and proposed a theoretical method to perform conformance testing on the state machine diagram. The proposed approach uses an input/out-pairs transition system (IOLTS), which is a modified LTS, to provide a suitable semantic model to state machine diagrams.

- *Primary Study 8*: This study uses the state machine diagram to generate test cases without using any intermediate format. The generated test cases satisfy the full predicate coverage and transition coverage. The proposed approach uses the depth-first search algorithm to perform the creation process. It combines with the boundary testing technique to reduce the number of test cases and avoid test case exploration problems.

- *Primary Study 9*: The proposed approach uses Fondue UML, a dialect of UML, to generate test cases by using the formal method. The proposed approach uses environment, protocol and operation diagrams corresponding to collaboration, state diagrams and OCL expressions in UML. The proposed method contains two phases. The first step transfers Fondue UML to concurrent object-oriented Petri Nets by using model transformation, then uses author-defined language to perform the test cases creation process. In the generation process, the defined language can choose the types of test cases to avoid countless test cases.

- *Primary Study 10*: This study uses the activity diagram to perform the test case generation process. The proposed approach converts activity diagrams to I/O explicit activity diagrams and then transfers the intermediate diagrams to the directed graph. Based on all paths coverage criteria to generate test cases. In the conversion process, through the single stimulus principle to avoid the state explosion problem. And in the test case generation step, the approach only retains the elementary paths, which are the paths without repeat occurrences of any node, to eliminate redundant test cases.

- *Primary Study 11*: In this study, authors use UML communication diagrams to perform the test case generation process. The overall idea of this approach is first to convert communication diagrams into a communication tree. This transformation eliminates the loop paths in communication diagrams, at the meantime, this conversion solves the space exploration problem. Based on the communication tree, the proposed approach uses post-order traverse and boundary coverage to generate test cases. The generated test cases satisfy the message paths and full predicate coverage. As the proposed approach uses the communication tree, the path selection strategy is simple, and the proposed approach can reach a high test coverage rate.

- *Primary Study 12*: This study uses collaboration diagrams and state diagrams to perform systemically test case generation. The main approach is to combine the collaboration and state diagrams and transfer them into an intermediate form called SCOTEM (State Collaboration Test Model). Through the traverse of the graph to generate the test cases, one independent path relates to one test case. This approach uses all-path coverage as the test criteria. The motivation of this approach is to reflect the interactions among the different components of the system under test to avoid integration errors. The authors also perform mutation tests to validate the effectiveness of the proposed approach, and the result shows that all-path coverage criteria can kill most of the mutations.

- *Primary Study 13*: The authors derive test cases from UML sequence diagrams and achieve the all-path coverage criteria. The proposed approach converts sequence diagram to sequence diagram graph, and collect the necessary information from other UML diagrams, like class diagram, use case template or data dictionary. When completing the process of generating the sequence diagram graph, traverse all possible paths to generate test cases. Each independent path corresponds to one specific test case. And the whole process does not require any modification of the UML models or manually set input/output data.
- *Primary Study 14*: The authors enhance the state diagrams to improve the coverage capabilities of an existing commercial test cases generation tool and achieve a coverage level more than MC/DC coverage. The basic idea of the proposed approach is to use the pre-processor to calculate the sequences of transitions by adding additional conditions and counters to achieve a higher coverage level. The post-processor is used to concatenate the test cases to reduce the number of the test suite. In the meantime, the proposed approach does not decrease the coverage level nor lead the system to an invalid trace.
- *Primary Study 15*: This study derives test cases from use case diagrams and sequence diagrams. The proposed approach converts use case diagrams and sequence diagrams to use case and sequence graphs. Then a system testing model is generated through the intermediate models plus the necessary information from class diagrams, use case template, and data dictionary. By using all use cases and all use case dependencies plus all message path coverage criteria, traverse the system testing model to generate the test cases. This

approach mainly avoids system level errors.

- *Primary Study 16*: This study uses the UML sequence diagram and OCL expression to generate test cases automatically. The proposed approach first converts the sequence diagram into a scenarios tree, another representation of the sequence diagram, and then traverses the tree-like graph to find paths to generate test cases. Meanwhile, use OCL expressions to determine the input data and satisfy the pre- and post- condition. The generated test cases achieve the scenario path coverage. The authors reach a high coverage level without excessive test cases.

- *Primary Study 17*: This study uses the sequence diagram to generate test cases, mainly focusing on mobile applications. The proposed approach uses the labelled transition system as the intermediate format, and the depth-first search algorithm is used to traverse the graph to perform the generation process. The labelled transition system uses redundant labels to represent alternative flows and loops. And the generated test cases can satisfy all path coverage criteria. This study uses two case studies to demonstrate the main steps, and the effectiveness of the proposed approach is checked by these case studies.

- *Primary Study 18*: This study uses state machine diagrams to generate test cases from the UML model, and the whole process does not use any intermediate format. The proposed approach mainly focuses on three steps. First, use the depth-first search or breadth-first search algorithm to select predicates. Second, use the predicate transformation process to convert predicates into predicate functions. Finally, the predicate function and boundary test techniques are used to generate test cases. In the generation process, the proposed approach satisfies full predicate coverage criteria and minimises the size of the generated test set. According to the result, this approach can achieve a higher transition path coverage level than random testing.

- *Primary Study 19*: This approach uses class diagram, state machine diagram and OCL expression to perform input partition and boundary testing. The proposed algorithm combines the information gathered from the OCL pre-, post-condition and class diagram, then transfers the state machine diagram into a test case tree. Then, traverse the test case tree to generate test cases, and the entire process satisfies all one-loop path coverage criteria. The authors use the mutation injection technique to compare the proposed algorithm with two different commercials tool and demonstrate the effectiveness of the proposed algorithm.

- *Primary Study 20*: This study uses UML models (use case, sequence, state diagram) to perform the test case generation process. The proposed approach extracts all possible scenarios from the sequence diagram, each use case corresponding to one sequence diagram, and uses possible scenarios to generate the system state graph. According to the transition path coverage criteria, traverse the system state graph to create test cases and combine them with the state diagram to verify the outcome of the test case. In the end, the authors demonstrate four different experiments and use mutation testing to validate the effectiveness of the proposed approach.

- *Primary Study 21*: This study aims to use another format called snapshots and filmstrips to express OCL expression. The new formats can express the constraints and benefit the test case generation process. The idea is to use independent graphs or expressions to reduce the complexity of pre- and post- conditions in OCL expressions. The proposed approach defines a new language to express the constraints and makes the constraints have precise semantics. The authors use many examples to show how the proposed approach works.

- *Primary Study 22*: This study aims to use the state machine diagram to generate test cases for the object-oriented system. The overall idea is to create a UML state machine diagram for the SUT and transfer the state machine diagram to a labelled transition system, then use a graph search algorithm to traverse the labelled transition system to generate test cases. Each test case consists of a sequence of system function calls corresponding to the events. The authors use one case study to demonstrate the effectiveness of the proposed algorithm.

- *Primary Study 23*: This study uses the activity diagram and Java program specification to perform grey box testing. The proposed approach uses an activity diagram to guide the Java program instrument process, then randomly generates a large number of test cases according to the program execution traces. After generating abundant test cases, combined with the activity diagram to match the expected activity diagram behaviour to perform the test case selection process. The selected test cases satisfy the corresponding test criteria. The approach also can check the consistency between the Java program and the activity diagram.

- *Primary Study 24*: This study uses the activity diagram to generate test cases, and the activity diagram may contain the sub-activity diagram. The proposed approach combines the sub-activity diagrams and activity diagrams to generate the composition tree, demonstrating the hierarchical relationship between these diagrams. In the composition tree, the leaf nodes represent the atomic activity diagram, and the parent nodes represent the compound activity diagrams. The test case generation strategy is called the bottom-up strategy, which traverses the tree, generates the test for the bottom level, and then generates the test for a higher level. When all level test cases are generated, use the round-robin method to combine the test cases to construct the final test suite. Compared with the complete combination strategy, the proposed approach can reach a relatively high coverage percentage without the test case explosion problem.

- *Primary Study 25*: This study uses the sequence diagram to perform regression testing. The proposed approach chooses which test cases can be used in a newer system version and which ones should be added. The idea is to construct the control flow graph from the sequence diagram. When there is a new version of the sequence diagram, construct a newer control flow graph from the latest version of the corresponding sequence diagram. Then traverse the control flow graphs, extract test scenarios, compare the test scenarios from two version control flow graphs to decide which test cases should be reused or discarded, and generate the new test cases to cover the new transition.

- *Primary Study 26*: This study aims to generate test cases automatically without using any intermediate format. The proposed approach uses class and sequence diagrams to perform the generation process. The diagrams are exported into XMI formats, and then the algorithm extracts the test sequence from the XMI file of the sequence diagram. Then the proposed approach uses the class diagram to determine the value of variables and finally uses robustness testing to generate test cases. The generated test case satisfies the transition coverage criteria on the sequence diagram. The authors also compare the amount of generated test cases with BVA and Worst Case testing.

- *Primary Study 27*: This study combines the UML sequence diagram and a genetic algorithm to generate test cases automatically. The proposed approach first analysis the sequence diagram to extract the relevant information, like the methods called and the parameters of each method, then encodes this information into chromosomes. There are three types of the method call, which are constructor, simple method invocation and value assignment. Using the fitness function to calculate the fitness value, the individual who can cover more messages will gain a higher fitness value. Through iterative evolution, use the best performance population to generate test cases. The generated test suit can satisfy 100% message sequence coverage from the test results.

- *Primary Study 28*: This study generates executable test cases based on the UML state machine diagram. The proposed approach first transfers the state diagram into a symbolic transition system. In this process, some pseudo states may represent junction, condition, entry, exist and deep history states. The proposed approach generates the initial paths from the transition system through the information (the marked elements) provided by an engineer. Then it extracts the relevant information, like parameters in the state machine diagram. Then the approach generates abstract test cases and transfers them into an executable format. The authors also compare the proposed approach with random testing.

- *Primary Study 29*: This study uses the sequence diagram to generate test cases for a concurrent system. The basic idea of the proposed approach is to transfer the sequence diagram into an intermediate format called the concurrent composition graph. The authors defined the transformation rules between UML diagrams and concurrent composition graphs, then traversed the graph by using depth-first search and breadth-first search to deal with sequential and concurrent messages. The gathered test sequence can directly convert into test cases. The generated test cases satisfy the message sequence path coverage, which can cover each possible path in the concurrent composition graph.

- *Primary Study 30*: This study aims to use OCL expression to generate the test data which can satisfy the OCL constraints. The proposed algorithm uses the heuristic search-based algorithm to perform the generation process, the author defined the fitness value, called branch distance, for OCL expression, and the fitness function is available for both primitive types and collection-related types. There are three different algorithms used in this study, which are the genetic algorithm, $(1 + 1)$ evolutionary algorithm and random searching. The authors show the results of these three algorithms and make comparisons between the algorithms.

- *Primary Study 31*: This study uses the OCL expression to generate cases, and the proposed tool supports the generation of test drivers. The proposed approach first transfers the normal OCL expression to high order logic format, then unfolds high-order logic expression to construct the disjunctive normal form. By traversing each part of the DNF expression to generate test cases. The transformation process also considers the invariants of the OCL expression and combines the invariants, pre-condition and post-condition to construct the high-order logic expression. And the authors use a single linked list as an example to demonstrate how to generate test cases.

- *Primary Study 32*: This study generate test cases from UML object diagram. The overall idea is to transfer the object diagram into a directed graph called the weighted graph. Through the transformation process, determine the messages sequence between objects and assign the weight to each message. The weight is assigned in ascending order. The proposed approach generates test cases from the least weight, traverses the graph until all messages are covered, and generates some invalid test cases (the message weight does not increase consecutively). The authors performed the mutation test to validate the effectiveness of the proposed approach.

- *Primary Study 33*: This study generates test cases and prioritises them from the UML activity diagram. The proposed algorithm identifies possible test flows from the UML activity diagram and transfers the UML activity diagram to an extended activity diagram, which adds the stereotype, like input, output and computes, to some states convenient for generating test steps. By using the depth-first search algorithm to traverse the extended activity diagram to generate test cases. This study also proposed an approach to prioritise the test cases, which assigns the weight for edges in the extended activity diagram to satisfy transition coverage criteria as a prerequisite. Test cases are selected based on the weight of the test flow.

- *Primary Study 34*: This study uses UML 2.0 collaboration diagram to generate test cases automatically. The proposed algorithm transfers the collaboration diagram to an intermediate graph called the weighted graph. In the conversion process, the message is assigned a weight value based on the sequence of the messages. Then traverse the graph to generate the test cases by using Prim's and Dijkstra's algorithms. The authors also use fault-based testing to demonstrate the effectiveness of the proposed approach.

- *Primary Study 35*: This study uses the activity diagram to generate test cases automatically. The proposed approach identifies all possible control constructs, transfers them into composition control constructs, and combines all nested structures into the intermediate testable model. Using the single path in the intermediate testable model generates the test path. Then unfold the nested structure in the single path and combine it with the coverage criteria, which are selection coverage, loop adequacy coverage and concurrent coverage, to generate test cases iteratively. The authors used mutation testing and compared the proposed algorithm with random testing to demonstrate the effectiveness of the proposed approach.

- *Primary Study 36*: This study uses the activity diagram to generate executable test cases without any intermediate format. The overall idea of the proposed approach is to instrument code into JAVA programs and then by tracing the execution of randomly generated test inputs to collect the feedback to predict how the inputs impact the decision nodes. The algorithm combines the feedback and simple path coverage criteria to generate executable test cases. The authors compared the proposed approach with random testing to demonstrate the advantages.

- *Primary Study 37*: This study uses class, sequence, use case diagram and OCL expression to generate test cases automatically. The proposed approach transfers these diagrams into XML files. The approach extracts message flows and condition predicates from the sequence diagram, pre- and post- condition from the use case diagram, OCL expression, and method-relented information from the class diagram, then combines this information into a sequence diagram graph. The approach is based on the sequence diagram graph, using a breadth-first search algorithm to traverse the graph to generate different test cases.

- *Primary Study 38*: This study combines the UML sequence diagram and state diagram to automatically generate test cases. The proposed approach converts the sequence and state diagram to a control flow graph, a graph like an activity diagram that supports most parts of the sequence and state diagram notations. Then the approach is based on coupling-based data flow testing criteria to retrieve test cases. The authors compared the proposed approach with an existing approach, called SCOTEM, to demonstrate the effectiveness and advantages of the proposed approach.

- *Primary Study 39*: This study uses the sequence diagram and condition slicing techniques to perform an automatic test case generation process. The proposed approach transfers the sequence diagram to the message dependency graph to show the relationships and dependencies of the messages. The approach selects conditional predicate from the intermediate diagram, creates the predicates slicing dynamically, and then combines this information with message path coverage, slice coverage and boundary testing criteria to generate test cases. The authors use a simple example to demonstrate the generation process in steps, and the generated test cases satisfy all path coverage criteria.

- *Primary Study 40*: This study converts the activity diagram to activity covert grammar, then uses AC grammar to generate test cases. The proposed approach is based on the transformation rules, which are mainly decided by the node type in the activity diagram, and convert the activity diagram to AC grammar. The approach iteratively adds new test steps from the start to the stop of the production, then generates the test cases, and the generated test set satisfies all path coverage criteria of the corresponding activity diagram.

- *Primary Study 41*: The study aims to generate test cases with minimal size by using the activity diagram. The proposed algorithm is mainly based on extension theory and transfers the activity Euler circuit. In the transformation process, there may be some auxiliary edges are added to make the number of in-degree equals out-degree for the whole graph. Then find the first loop of the Euler circuit, based on the in-degree and out-degree, to generate the test cases. The authors used an example to demonstrate how to generate the test cases, and the generated test cases can satisfy the transition coverage criteria of the activity diagram.

- *Primary Study 42*: This study verifies the correctness of the system based on web service. The proposed approach first transfers the WS-BPEL, a standard integration language, to the sequence diagram and WSDL to the extended state diagram. The two types of UML diagrams combine to generate the intermediate format EFSM-SeTM. Based on whether the web services are stateful or stateless, there are different representations in the EFSM-SeTM. Based on the different coverage criteria, the approach traverses the EFSM-SeTM diagram to generate the test cases, and the proposed approach satisfies all path coverage criteria.

- *Primary Study 43*: This study aims to improve the test case generation process by reducing the number of test cases and still satisfying the coverage criteria. The proposed approach transfers the state machine diagram to a formal representation called LOTOS specifications. The approach identifies the controls, data and communication dependencies on LOTOS specifications, and these specifications can be covert into test purposes. And test purposes combine with the user-defined test purpose to point out which transitions are not necessary for exploitation, then use the TGV tool to generate the test cases. The authors also used one study case to demonstrate the generation process. They used seven different studies, which contain three real-world studies, to validate the effectiveness of the proposed algorithm.

- *Primary Study 44*: The proposed approach generates the state machine diagram based on the UML class diagram and then transfers the state diagram to the control flow graph. From the CFG, the approach first identifies the definition and usage information for all variables and then eliminates all invalid DU pairs. The approach can generate test cases through valid DU (Define & use) pairs by tracing the corresponding variable usage. Through mutation testing, the authors validated the effectiveness of the proposed approach, and the generated test cases satisfy Path and DU pairs coverage.

- *Primary Study 45*: This study uses the UML activity diagram and OCL expression to generate test cases. The proposed approach converts the activity diagram and OCL expression to a mathematical programming language as an intermediate format, then traverse the intermediate format to eliminate the infeasible paths. Then based on the gathered information and the type of the problem, the proposed approach uses the existing tool to generate test cases. The authors performed mutation testing to validate the effectiveness of the proposed approach and show the boundary value does not affect the efficiency of test case generation for this approach.

- *Primary Study 46*: This study uses the UML sequence diagram to generate test cases automatically. The proposed algorithm transfers the sequence diagram to scene test trees. Then the approach traverses the tree to slice the whole tree into different sub-trees, generates the test data for each sub-tree, and combines each part of the test data to construct test cases. The authors used an example to explain the proposed approach, and the generated test cases satisfy the state coverage criteria.

- *Primary Study 47*: This study uses class diagram, state diagram and OCL expression to perform test case selection and test case generation processes for regression testing. The proposed approach analyses the dependence relationship between the new diagram and the original one based on the impact analysis of system behaviours to classify tests. Then the approach uses classification results to determine whether to keep, update, obsolete or generate new test cases. The necessary diagram should be the class diagram with or without the state machine diagram. The authors demonstrate the effectiveness and efficiency of the proposed approach through experiments.

- *Primary Study 48*: The proposed approach combines the UML activity diagram and genetic algorithm to generate and optimise test cases automatically. This approach converts the activity diagram to an activity flow graph via an activity flow table and then uses the depth-first search algorithm to generate the possible paths. Then the approach applies the possible paths to the genetic algorithm to generate test cases, and the fitness value depends on the weight of the paths. The authors used the ATM example to demonstrate the whole process, and the test cases satisfy the path coverage criteria.

- *Primary Study 49*: This study uses the sequence diagram, state machine diagram and the genetic algorithm to generate test cases. The proposed approach transfers the sequence and state machine diagram into the sequence and state diagram graphs. Then use these two graphs to generate the system testing graph. The proposed approach traverses the system testing graph to find possible paths and then applies these paths to the genetic algorithm to generate optimised test cases. The authors used an online voting system to demonstrate the process, but the chromosome definition in the genetic algorithm is not explained clearly.

- *Primary Study 50*: This study uses the activity diagram to automatically perform the test case generation process for medium-scale web-based applications. The proposed approach converts the activity diagram into a weighted base graph, prioritising each transition. By traversing the graph to find the possible path to generate corresponding test cases and calculate the weight of flows. In the meantime, use the weight of flows to calculate the ideal distance and the weights of flows to approximately calculate the effort level. The authors also demonstrate the traders between effort level and ideal distance.

- *Primary Study 51*: This study proposed an approach to generate test cases based on the UML sequence diagram for the agile project process. The proposed approach is mainly based on model transformation techniques. The methodology for this approach coverts the sequence diagram into UML 2.0 testing profile and then uses the AndroMDA framework to generate test cases. The two steps are based on model-to-model transformation and model-to-text transformation, and the authors used an example to demonstrate the proposed approach.

- *Primary Study 52*: This study aims to use the activity diagram to generate test cases for the concurrent system, especially to deal with the deadlock problem. The proposed algorithm generates the input/output activity diagram, which hides non-external I/O operations, from the UML activity diagram. By analysing the dependency relationship, the proposed algorithm traverses the IOAD using the breadth-first search algorithm, then combines with the lock nodes, unlock nodes and the waiting list to generate test cases and avoid deadlock situations. The authors used an online shopping system to show the process, and the generated test cases have the ability to cover all path criteria.

- *Primary Study 53*: This study uses the activity diagram and sequence diagram to generate test cases. The proposed algorithm first transfers the activity and sequence diagram to the corresponding graph and then combines these graphs into the system testing graph. The approach uses a modified depth-first algorithm to traverse the system testing graph to find the possible paths to generate test cases. The authors compared the results which test cases generated from the activity graph, sequence graph and system testing graph. The results show test cases from the system testing graph may contain redundant test cases.

- *Primary Study 54*: This study uses the UML activity diagram and a bio-inspired algorithm to generate test scenarios for the concurrent system. The proposed approach transfers the activity to an intermediate testable graph and then applies the graph to the bio-inspired algorithm to generate the corresponding test scenarios. This study points out the shortage of depth-first or breadth-first search algorithms. The authors also compare the result with the generation and ant colony optimization algorithms. The results show the AOA has advantages when the elements in the generated path are abundant.

- *Primary Study 55*: This study mainly focused on testing coverage criteria in automatic test case generation. The authors explained the various types of criteria for the UML state diagram, and the state diagram first converts to the state machine diagram. The user can use different coverage criteria to generate test cases from the state graph according to the requirements.

- *Primary Study 56*: The proposed approach for generating test cases from the UML interaction diagram using stack array and boundary value techniques yields efficient test cases. And the proposed approach is validated through two case studies.
- *Primary Study 57*: This study uses the state machine diagram to generate and prioritise the test cases. The proposed approach converts the state diagram to the composite control flow graph, and then the approach finds the path from the CCFG to generate the test cases. The approach uses mutation testing and APFD to prioritise the test cases and the prioritisation process based on how many faults can be detected by the test cases. The authors used an ATM approach to demonstrate the whole process of the proposed approach.
- *Primary Study 58*: This study uses the class, activity and use cases diagram to generate and select test cases for regression testing. The class diagram identifies the changes at the static level. At the same time, the use cases and activity diagram provide dynamic behavioural information. The proposed approach converts UML diagrams to XML or XMI format. Then the approach uses different agents for each diagram to compare the old version diagram and the modified diagram to select which test cases should be reserved or reused and which test cases should be generated.
- *Primary Study 59*: This study uses the UML activity diagram to generate test cases for the concurrent system. The proposed approach first converts the activity diagram to the

concurrent activity graph, which shows the relationship between each concurrent node and can derive all possible interaction paths from this graph. Then the approach combines with the user-defines constraints, which avoid the test cases explosion problem, to traverse the graph to generate test cases. The authors compared the proposed approach with the depth-first search and breadth-first search algorithms to demonstrate the advantages.

- *Primary Study 60*: This study aims to generate test data for OCL expression to achieve MC/DC coverage criteria. The proposed approach reformulates the OCL constraints to conjunction format and combines them with case-based reasoning to generate test data. The proposed approach uses case-based reasoning to reuse the generated test cases to construct new test cases and uses a maximum iteration number to detect conflict constraints. The authors performed contrast experiments between four case studies to demonstrate the effectiveness of the proposed approach.

- *Primary Study 61*: The proposed approach uses the UML activity diagram to generate test cases automatically and aims to achieve the MC/DC coverage criteria. The approach uses three open-source and one in-house tool to perform the process. The overall idea is that transfer the activity diagram to the XML file and transfer XML to XSD code, then use JAXB to generate executable JAVA code, and finally use jCUTE to create test cases. The proposed approach uses COPECA to satisfy the MC/DC coverage criteria.

- *Primary Study 62*: This study uses the state diagram to generate test cases based on different coverage criteria automatically. The proposed algorithm transfers the state diagram to the state machine diagram intermediate graph and combines it with the coverage criteria to traverse the graph to generate test cases. The approach provides three coverage criteria, which are all transition, round trip path and all transition pair coverage criteria. The authors used two studies to demonstrate the generation process and showed the most effective coverage criteria is round trip path coverage criteria.

Appendix B

Specification - Case Study 1: Bond

```
package app5 {
    class Bond {
    stereotype persistent;

        invariant term > 0 & term <= 100;
        invariant coupon >= 0 & coupon <= 100;
        invariant duration > 0 & duration <= 100;
        invariant frequency >= 0 & frequency <= 400;
        invariant price >= 0 & price <= 500;

        attribute name identity : String;
        attribute term : double;
        attribute coupon : double;
        attribute price : double;
        attribute frequency : int;
        attribute yield derived : double;
        attribute duration derived : double;

        operation discount(amount : double , r : double , time : double
            ) : double
        pre: r > -1 & r <= 1 & time >= 0
        post: result = amount / ( ( 1 + r )->pow(time) );

        operation value(r : double ) : double
        pre: r > -1 & r <= 1
        post: upper = ( term * frequency )->floor()->oclAsType(int) & c
            = coupon / frequency & period = 1.0 / frequency & result =
            Integer.subrange(1,upper)->collect( i | self.discount(c,r,
           i * period) )->sum() + self.discount(100,r,term);

        operation timeDiscount(amount : double , r : double , time :
           double ) : double
        pre: r > -1 & r <= 1 & time >= 0
        post: result = ( amount * time ) / ( ( 1 + r )->pow(time) );

        operation macaulayDuration(r : double ) : double
        pre: r > -1 & r <= 1
        post: upper = ( term * frequency )->floor()->oclAsType(int) & c
            = coupon / frequency & period = 1.0 / frequency & result =
            ( Integer.subrange(1,upper)->collect( i | self.
           timeDiscount(c,r,i * period) )->sum() + self.timeDiscount
           (100,r,term) ) / self.value(r);
        operation bisection(r : double , rl : double , ru : double ) :
            double
        pre: r > -1 & rl > -1 & ru > -1 & ru <= 1 & rl <= 1 & r <= 1
        post: v = value(r) & result = (if ru - rl < 0.001 then r else (
            if v > price then self.bisection(( ru + r ) / 2,r,ru) else
            self.bisection(( r + rl ) / 2,rl,r) endif) endif);
    }

    usecase findDuration : String {

        parameter bond : Bond;

        ::
         true => bond.duration = bond.macaulayDuration(bond.yield) &
             result = "Duration is: " + bond.duration
    }

    usecase findYield : String {
        extendedBy findDuration;

        parameter bond : Bond;

        ::
         true => bond.yield = bond.bisection(0.25,-0.5,1) & result = "
             Yield is: " + bond.yield
    }
}
```

Appendix C

Specification - Case Study 2: Interest Rate

```
package nsapp {
    class InterestRate {

        attribute maturity : double;
        attribute rate : double;
        attribute interestrateId : String;

        static operation nelsonseigal(t : double , v1 : double , v2 :
            double , v3 : double , lambda1 : double ) : double
        pre: t >= 0 & t <= 100 & v1 >= -5 & v1 <= 5 & v2 >= -5 & v2 <=
            5 & v3 >= -5 & v3 <= 5 & lambda1 >= -5 & lambda1 <= 5
        post: true
        activity:   var res : double ; res := 0.0   ;   var tscaled :
            double ; tscaled := 0.0   ; tscaled := t / lambda1   ;   var
             exptscaled : double ; exptscaled := 0.0   ; exptscaled :=
            (-tscaled)->exp()   ;   var expratio : double ; expratio :=
            0.0   ; expratio := ( 1 - exptscaled ) / tscaled   ; res :=
            v1 + v2 * expratio + v3 * ( expratio - exptscaled )   ;
            return res ;

        static operation ns(t : double , v1 : double , v2 : double , v3
             : double , lambda1 : double ) : double
        pre: t >= 0 & t <= 100 & v1 >= -5 & v1 <= 5 & v2 >= -5 & v2 <=
            5 & v3 >= -5 & v3 <= 5 & lambda1 >= -5 & lambda1 <= 5
        post: true
        activity:   var res : double ; res := 0.0   ; if ( t = 0 ) then
             res := v1 + v2   else ( ( if ( t > 0 ) then res :=
            InterestRate.nelsonseigal(t,v1,v2,v3,lambda1)   else skip
              ) ) ; return res ;
    }

    usecase plotNS : Sequence(double){
        parameter time : double;
        parameter v1 : double;
        parameter v2 : double;
        parameter v3 : double;
        parameter lambda1 : double;

        ::
          true =>  result = Integer.subrange(1,time->floor())->collect(
              ind | InterestRate.ns( ind, v1, v2, v3, lambda1));
    }
}
```

Appendix D

Specification - Case Study 3: MathLib

```
package mathlib {
    class MathLib {

        static attribute ix : int;
        static attribute iy : int;
        static attribute iz : int;
        static attribute hexdigit : Sequence(String);

        static operation initialiseMathLib() : void
        pre: true
        post: true
        activity:
          (MathLib.hexdigit := Sequence{ "0", "1", "2", "3", "4", "5",
             "6", "7", "8", "9", "A", "B", "C", "D", "E", "F" } ;
             MathLib.setSeeds(1001, 781, 913) );

        static query pi() : double
        pre: true
        post: result = 3.14159265;

        static query piValue() : double
        pre: true
        post: result = 3.14159265;

        static query eValue() : double
        pre: true
        post: result = 1->exp();

        static operation setSeeds(x : int , y : int , z : int ) : void
        pre: true
        post: MathLib.ix = x & MathLib.iy = y & MathLib.iz = z;

        static operation nrandom() : double
        pre: true
        post: true
        activity: ( MathLib.ix := ( MathLib.ix * 171 ) mod 30269 ;
           MathLib.iy := ( MathLib.iy * 172 ) mod 30307 ; MathLib.iz
           := ( MathLib.iz * 170 ) mod 30323 ; return ( MathLib.ix /
           30269.0 + MathLib.iy / 30307.0 + MathLib.iz / 30323.0 ) );
```

```
static query random() : double
pre: true
post: r = MathLib.nrandom() & result = ( r - r->floor() );

static query combinatorial(n : int , m : int ) : long
pre: n >= m & m >= 0
post: ( n - m < m => result = Integer.Prd(m + 1,n,i,i) /
   Integer.Prd(1,n - m,j,j) ) & ( n - m >= m => result =
   Integer.Prd(n - m + 1,n,i,i) / Integer.Prd(1,m,j,j) );

static query factorial(x : int ) : long
pre: true
post: ( x < 2 => result = 1 ) & ( x >= 2 => result = Integer.
   Prd(2,x,i,i) );

static query asinh(x : double ) : double
pre: true
post: result = ( x + ( x * x + 1 )->sqrt() )->log();

static query acosh(x : double ) : double
pre: x >= 1
post: result = ( x + ( x * x - 1 )->sqrt() )->log();

static query atanh(x : double ) : double
pre: x /= 1
post: result = 0.5 * ( ( 1 + x ) / ( 1 - x ) )->log();

static query decimal2bits(x : long ) : String
pre: true
post: if x = 0 then result = "" else result = MathLib.
   decimal2bits(x / 2) + "" + ( x mod 2 ) endif;

static query decimal2binary(x : long ) : String
pre: true
post: if x < 0 then result = "-" + MathLib.decimal2bits(-x)
   else if x = 0 then result = "0" else result = MathLib.
   decimal2bits(x) endif endif;

static query decimal2oct(x : long ) : String
pre: true
post: if x = 0 then result = "" else result = MathLib.
   decimal2oct(x / 8) + "" + ( x mod 8 ) endif;

static query decimal2octal(x : long ) : String
pre: true
post: if x < 0 then result = "-" + MathLib.decimal2oct(-x) else
    if x = 0 then result = "0" else result = MathLib.
   decimal2oct(x) endif endif;

static query decimal2hx( x : long) : String
pre: true
post: if x = 0 then result = "" else result = MathLib.
   decimal2hx(x/16) + ("" + MathLib.hexdigit->at((x mod 16)->
   oclAsType(int) + 1)) endif;

static query decimal2hex(x : long ) : String
pre: true
```

```
post: if x < 0 then result = "-" + MathLib.decimal2hx(-x) else
   if x = 0 then result = "0" else result = MathLib.decimal2hx
   (x) endif endif;

static query bytes2integer(bs : Sequence(int)) : long
pre: true
post: (bs->size() = 0 => result = 0) & (bs->size() = 1 =>
   result = bs->at(1)) & (bs->size() = 2 => result = 256*(bs->
   at(1)) + bs->at(2)) & (bs->size() > 2 => result = 256*
   MathLib.bytes2integer(bs->front()) + bs->last());

static query integer2bytes(x : long) : Sequence(int)
pre: true
post: if (x/256) = 0 then result = Sequence{(x mod 256)} else
   result = MathLib.integer2bytes(x/256)->append(x mod 256)
   endif;

static query integer2Nbytes(x : long, n : int) : Sequence(int)
pre: true
post: bs = MathLib.integer2bytes(x) & ((bs->size() < n =>
   result = (Integer.subrange(1,n-(bs->size()))->collect(0))->
   concatenate(bs)) & (bs->size() >= n => result = bs));

static query bitwiseAnd(x : int, y : int) : int
pre: true post: true
activity:
    var x1 : int ; x1 := x;
    var y1 : int ; y1 := y;
    var k : int ; k := 1;
    var res : int ; res := 0;
    while (x1 > 0 & y1 > 0)
        do
        ( if x1 mod 2 = 1 & y1 mod 2 = 1
          then
            res := res + k
          else skip ;
       k := k*2;
       x1 := x1/2;
       y1 := y1/2
        ) ;
    return res;

static query bitwiseOr(x : int, y : int) : int
pre: true post: true
activity:
   var x1 : int ; x1 := x;
   var y1 : int ; y1 := y;
   var k : int ; k := 1;
   var res : int ; res := 0;
   while (x1 > 0 or y1 > 0)
       do
        ( if x1 mod 2 = 1 or y1 mod 2 = 1
          then
            res := res + k
          else skip ;
       k := k*2;
       x1 := x1/2;
       y1 := y1/2
```

```
            ) ;
        return res;

    static query bitwiseXor(x : int, y : int) : int
    pre: true post: true
    activity:
       var x1 : int ; x1 := x;
       var y1 : int ; y1 := y;
       var k : int ; k := 1;
       var res : int ; res := 0;
       while (x1 > 0 or y1 > 0)
           do
            ( if (x1 mod 2) /= (y1 mod 2)
              then
                res := res + k
              else skip ;
           k := k*2;
           x1 := x1/2;
           y1 := y1/2
            ) ;
        return res;

    static query bitwiseNot(x : int) : int
    pre: true
    post: result = -(x+1);

    static query toBitSequence(x : long) : Sequence(boolean)
    pre: true post: true
    activity:
        var x1 : long ; x1 := x ;
        var res : Sequence(boolean) ; res := Sequence{} ;
        while x1 > 0
        do
            ( if x1 mod 2 = 0
              then
                res := res->prepend(false)
              else
                res := res->prepend(true) ;
              x1 := x1/2
            ) ;
        return res;

    static query modInverse(n : long, p : long) : long
    pre: p > 0 post: true
    activity:
        var x : long ; x := (n mod p) ;
        var i : int;
        i := 1;
        while i < p
            do
              ( if ((i*x) mod p) = 1
                then return i
                else skip ;
                i := i+1 );
        return 0;

    static query modPow(n : long, m : long, p : long) : long
    pre: p > 0 post: true
```

```
        activity:
            var res : long ; res := 1 ;
            var x : long ; x := (n mod p) ;
            var i : int ;
            i := 1 ;
                while i <= m
                do
                  ( res := ((res*x) mod p) ; i := i + 1) ;
            return res;

        static query doubleToLongBits(d : double) : long
        pre: true
        post: true;

        static query longBitsToDouble(x : long) : double
        pre: true
        post: true;

    }
}
```

Appendix E

Specification - Case Study 4: UML2PY

```
package app {
    enumeration UMLKind {
        literal value;
        literal attribute;
        literal role;
        literal variable;
        literal constant;
        literal function;
        literal queryop;
        literal operation;
        literal classid;
    }

    abstract class NamedElement {
    stereotype abstract;

        attribute name : String;
    }

    abstract class Relationship extends NamedElement {
    stereotype abstract;
    }

    abstract class Feature extends NamedElement {
    stereotype abstract;

        reference type : Type;
        reference elementType : Type;
    }

    abstract class Type extends NamedElement {
    stereotype abstract;

        attribute typeId identity : String;

        operation defaultInitialValue() : String
        pre: true
        post: true;

        operation toPython() : String
        pre: true
        post: result = name;

        operation isStringType() : boolean
        pre: true
        post: ( name = "String" => result = true );
    }

    class Association extends Relationship {

        attribute addOnly : boolean;
        attribute aggregation : boolean;

        reference memberEnd[*] ordered : Property;
    }

    class Generalization extends Relationship {

        reference specific : Entity oppositeOf generalization;
        reference general : Entity oppositeOf specialization;
    }

    abstract class Classifier extends Type {
    stereotype abstract;
    }

    abstract class DataType extends Classifier {
    stereotype abstract;
    }

    class Enumeration extends DataType {

        reference ownedLiteral[*] ordered : EnumerationLiteral;

        operation defaultInitialValue() : String
        pre: true
        post: result = name + "." + ownedLiteral[1].name;

        operation toPython() : String
        pre: true
        post: result = name;

        operation literals() : String
        pre: true
        post: result = ownedLiteral->collect( lt | "    " + lt.name + " =
            " + ownedLiteral->indexOf(lt) + "\n" )->sum();
    }

    class EnumerationLiteral extends NamedElement {
    }

    class PrimitiveType extends DataType {

        static operation isPrimitiveType(s : String ) : boolean
        pre: true
        post: ( s = "double" => result = true ) & ( s = "long" =>
            result = true ) & ( s = "String" => result = true ) & ( s =
            "boolean" => result = true ) & ( s = "int" => result =
         true ) & ( s = "OclType" => result = true ) & ( s = "
         OclVoid" => result = true );

      static operation isPythonPrimitiveType(s : String ) : boolean
      pre: true
      post: ( s = "float" => result = true ) & ( s = "int" => result
          = true ) & ( s = "str" => result = true ) & ( s = "bool" =>
           result = true );

      operation defaultInitialValue() : String
      pre: true
      post: ( name = "double" => result = "0.0" ) & ( name = "String"
           => result = "\"\"" ) & ( name = "boolean" => result = "
         False" ) & ( name = "int" => result = "0" ) & ( name = "
         long" => result = "0" ) & ( true => result = "None" );

      operation toPython() : String
      pre: true
      post: ( name = "double" => result = "float" ) & ( name = "long"
           => result = "int" ) & ( name = "String" => result = "str"
         ) & ( name = "boolean" => result = "bool" ) & ( name = "
         OclType" => result = "type" ) & ( name = "OclVoid" =>
         result = "None" ) & ( name = "OclException" => result = "
         BaseException" ) & ( true => result = name );
  }

  class Property {

      attribute name : String;
      attribute lower : int;
      attribute upper : int;
      attribute isOrdered : boolean;
      attribute isUnique : boolean;
      attribute isDerived : boolean;
      attribute isReadOnly : boolean;
      attribute isStatic : boolean;

      reference qualifier[0-1] : Property;
      reference type : Type;
      reference initialValue : Expression;
      reference owner : Entity oppositeOf ownedAttribute;

      operation initialisation() : String
      pre: true
      post: ( type->oclIsUndefined() => result = "    self." + name +
           " = None\n" ) & ( qualifier.size > 0 => result = "    self
         ." + name + " = dict({})\n" ) & ( qualifier.size = 0 =>
         result = "    self." + name + " = " + type.
         defaultInitialValue() + "\n" );

      operation getPKOp(ent : String ) : String
      pre: true
      post: e = ent.toLowerCase & result = "def get" + ent + "ByPK(
         _ex) :\n" + "  if (_ex in " + ent + "." + e + "_index) :\n"
          + "    return " + ent + "." + e + "_index[_ex]\n" + "
         else :\n" + "    return None\n\n";

      operation getPKOps(ent : String ) : String
```

```
        pre: true
        post: e = ent.toLowerCase & result = "def get" + ent + "ByPKs(
            _exs) :\n" + "   result = []\n" + "   for _ex in _exs :\n" +
            "     if (_ex in " + ent + "." + e + "_index) :\n" + "
            result.append(" + ent + "." + e + "_index[_ex])\n" + "
            return result\n\n";
    }

    class Operation extends BehaviouralFeature {

        attribute isQuery : boolean;
        attribute isAbstract : boolean;
        attribute isCached : boolean;

        reference owner : Entity oppositeOf ownedOperation;
        reference definers[*] ordered : Entity;

        operation displayStaticCachedOperation(indent : int , p :
            String , obj : String ) : void
        pre: true
        post: ( Statement.tab(indent) + "def " + name + "(" + p + ") :"
             )->display() & ( Statement.tab(indent + 2) + "if str(" + p
             + ") in " + obj + "." + name + "_cache :" )->display() & (
             Statement.tab(indent + 4) + "return " + obj + "." + name +
             "_cache[str(" + p + ")]" )->display() & ( Statement.tab(
            indent + 2) + "result = " + obj + "." + name + "_uncached("
             + p + ")" )->display() & ( Statement.tab(indent + 2) + obj
             + "." + name + "_cache[str(" + p + ")] = result" )->
            display() & ( Statement.tab(indent + 2) + "return result" )
            ->display() & "\n"->display() & ( Statement.tab(indent) + "
            def " + name + "_uncached(" + p + ") :" )->display() &
            activity.toPython(indent + 2)->display() & "\n"->display();

        operation displayInstanceCachedOperation(indent : int , p :
            String , obj : String ) : void
        pre: true
        post: ( Statement.tab(indent) + "def " + name + "(self, " + p +
             ") :" )->display() & ( Statement.tab(indent + 2) + "if str
            (" + p + ") in " + obj + "." + name + "_cache :" )->display
            () & ( Statement.tab(indent + 4) + "return " + obj + "." +
            name + "_cache[str(" + p + ")]" )->display() & ( Statement.
            tab(indent + 2) + "result = " + obj + "." + name + "
            _uncached(" + p + ")" )->display() & ( Statement.tab(indent
             + 2) + obj + "." + name + "_cache[str(" + p + ")] = result
            " )->display() & ( Statement.tab(indent + 2) + "return
            result" )->display() & "\n"->display() & ( Statement.tab(
            indent) + "def " + name + "_uncached(self, " + p + ") :" )
            ->display() & activity.toPython(indent + 2)->display() & "\
            n"->display();

        operation displayStaticOperation(indent : int ) : void
        pre: true
        post: ( isCached = true & parameters.size = 1 =>
            displayStaticCachedOperation(indent,parameters[1].name,
            owner.name) ) & ( ( isCached = false or parameters.size /=
            1 ) => ( Statement.tab(indent) + "def " + name + "(" +
            Expression.tolist(parameters.name) + ") :" )->display() &
            activity.toPython(indent + 2)->display() );
```

```
      operation displayInstanceOperation(indent : int ) : void
      pre: true
      post: ( isCached = true & parameters.size = 1 =>
          displayInstanceCachedOperation(indent,parameters[1].name,"
          self") ) & ( ( isCached = false or parameters.size /= 1 )
          => ( Statement.tab(indent) + "def " + name + "(self" +
          parameters->collect( p | ", " + p.name )->sum() + ") :" )->
          display() & activity.toPython(indent + 2)->display() );

      operation displayOperation(indent : int ) : void
      pre: true
      post: ( isStatic = true => displayStaticOperation(indent) ) & (
           isStatic = false => displayInstanceOperation(indent) );
  }

  class Entity extends Classifier {

      attribute isAbstract : boolean;
      attribute isInterface : boolean;
      attribute stereotypes : Sequence(String);

      reference generalization[*] : Generalization oppositeOf
          specific;
      reference specialization[*] : Generalization oppositeOf general
          ;
      reference ownedOperation[*] ordered : Operation oppositeOf
          owner;
      reference ownedAttribute[*] : Property oppositeOf owner;
      reference superclass[0-1] : Entity oppositeOf subclasses;
      reference subclasses[*] : Entity oppositeOf superclass;

      operation allLeafSubclasses() : Set(Entity)
      pre: true
      post: ( specialization.size = 0 => result = Set{self} ) & (
          specialization.size > 0 => result = specialization->
          unionAll(specific.allLeafSubclasses()) );

      operation allProperties() : Set(Property)
      pre: true
      post: ( generalization.size = 0 => result = ownedAttribute ) &
          ( generalization.size > 0 => result = ownedAttribute->union
          (generalization.general.allProperties()) );

      operation allOperations() : Set(Operation)
      pre: true
      post: oonames = self.ownedOperation->collect(name) & ( (
          generalization.size = 0 => result = self.ownedOperation ) &
           ( generalization.size > 0 => result = self.ownedOperation
          ->union(generalization.general.allOperations()->select( op
          | op.name /: oonames )) ) );

      operation isApplicationClass() : boolean
      pre: true
      post: ( "external" /: stereotypes & "component" /: stereotypes
          ) => result = true;

      operation isActiveClass() : boolean
```

```
        pre: true
        post: ( "active" : stereotypes => result = true );

        operation isSingleValued(d : String ) : boolean
        pre: true
        post: allProperties()->exists( p | p.name = d & p.lower = 1 & p
            .upper = 1 ) => result = true;

        operation isSetValued(d : String ) : boolean
        pre: true
        post: allProperties()->exists( p | p.name = d & ( p.lower /= 1
            or p.upper /= 1 ) & p.isOrdered = false ) => result = true;

        operation isSequenceValued(d : String ) : boolean
        pre: true
        post: allProperties()->exists( p | p.name = d & ( p.lower /= 1
            or p.upper /= 1 ) & p.isOrdered = true ) => result = true;

        operation defaultInitialValue() : String
        pre: true
        post: result = "None";

        operation initialisations() : String
        pre: true
        post: result = allProperties()->select( a | a.isStatic = false
            )->collect( x | x.initialisation() )->sum();

        operation generalisationString() : String
        pre: true
        post: ( generalization.size = 0 => result = "" ) & (
            generalization.size > 0 => result = "(" + generalization.
            any.general.name + ")" );

        operation classHeader() : String
        pre: true
        post: ( superclass.size = 0 => result = "class " + name + " : "
             ) & ( superclass->exists( c | c.isInterface = false ) =>
            result = "class " + name + "(" + superclass.any.name + ") :
             " ) & ( true => result = "class " + name + " : " );

        operation allStaticCaches() : String
        pre: true
        post: result = ownedOperation->select( x | x.isStatic & x.
            isCached & x.parameters.size = 1 )->collect( y | "  " + y.
            name + "_cache = dict({})" + "\n" )->sum();

        operation allInstanceCaches() : String
        pre: true
        post: result = ownedOperation->select( x | x.isStatic = false &
             x.isCached & x.parameters.size = 1 )->collect( y | "
            self." + y.name + "_cache = dict({})" + "\n" )->sum();

        operation staticAttributes() : String
        pre: true
        post: result = "  " + name.toLowerCase() + "_instances = []\n"
           + "  " + name.toLowerCase() + "_index = dict({})\n" +
            allProperties()->select( x | x.isStatic )->collect( y | "
            " + y.name + " = " + y.type.defaultInitialValue() + "\n" )
```

```
   ->sum() + allStaticCaches();

operation classConstructor() : String
pre: true
post: result = "  def __init__(self):\n" + initialisations() +
   "    " + name + "." + name.toLowerCase() + "_instances.
   append(self)\n" + allInstanceCaches() + "\n";

operation abstractClassConstructor() : String
pre: true
post: result = "  def __init__(self):\n" + "    " + name + "."
   + name.toLowerCase() + "_instances.append(self)\n";

operation callOp() : String
pre: true
post: ( isActiveClass() => result = "  def __call__(self):\n" +
    "    self.run()\n\n" ) & ( true => result = "" );

operation createOp() : String
pre: true
post: result = "def create" + name + "():\n" + "  " + name.
   toLowerCase() + " = " + name + "()\n" + "  return " + name.
   toLowerCase() + "\n";

operation createPKOp(key : String ) : String
pre: true
post: result = "def createByPK" + name + "(_value):\n" + "
   result = get" + name + "ByPK(_value)\n" + "  if (result !=
   None) : \n" + "    return result\n" + "  else :\n" + "
   result = " + name + "()\n" + "    result." + key + " =
   _value\n" + "    " + name + "." + name.toLowerCase() + "
   _index[_value] = result\n" + "    return result\n";

static operation deleteOp() : String
pre: true
post: result = "def free(x):\n" + "  del x\n\n";

static operation instancesOps(leafs : Set(Entity) ) : String
pre: true
post: ( leafs.size = 1 => result = "allInstances_" + leafs.
   first.name + "()" ) & ( leafs.size > 1 => result = "ocl.
   union(allInstances_" + leafs.first.name + "(), " + Entity.
   instancesOps(leafs.tail) + ")" );

operation createOclTypeOp() : String
pre: true
post: typename = name.toLowerCase() + "_OclType" & result =
   typename + " = createByPKOclType(\"" + name + "\")\n" +
   typename + ".instance = create" + name + "()\n" + typename
   + ".actualMetatype = type(" + typename + ".instance)\n";

operation allInstancesOp() : String
pre: true
post: ( specialization.size = 0 => result = "def allInstances_"
    + name + "():\n" + "  return " + name + "." + name.
   toLowerCase() + "_instances\n" ) & ( specialization.size >
   0 => result = "def allInstances_" + name + "():\n" + "
   return " + Entity.instancesOps(allLeafSubclasses()) );
```

```
        static operation displayOps() : String
        pre: true
        post: result = "def displayint(x):\n" + "   print(str(x))\n\n" +
            "def displaylong(x):\n" + "   print(str(x))\n\n" + "def
            displaydouble(x):\n" + "   print(str(x))\n\n" + "def
            displayboolean(x):\n" + "   print(str(x))\n\n" + "def
            displayString(x):\n" + "   print(x)\n\n" + "def
            displaySequence(x):\n" + "   print(x)\n\n" + "def displaySet
            (x):\n" + "   print(x)\n\n" + "def displayMap(x):\n" + "
            print(x)\n\n";
    }

    class CollectionType extends DataType {

        reference elementType : Type;
        reference keyType : Type;

        operation defaultInitialValue() : String
        pre: true
        post: ( name = "Sequence" => result = "[]" ) & ( name = "Set"
            => result = "set({})" ) & ( name = "Map" => result = "dict
            ({})" ) & ( name = "Function" => result = "None" );
    }

    abstract class Expression {
    stereotype abstract;

        attribute needsBracket : boolean;
        attribute umlKind : UMLKind;
        attribute expId identity : String;
        attribute isStatic : boolean;

        reference type : Type;
        reference elementType : Type;

        static operation tolist(s : Sequence(String) ) : String
        pre: true
        post: ( s.size = 0 => result = "" ) & ( s.size = 1 => result =
            s->first() ) & ( s.size > 1 => result = s->first() + "," +
            Expression.tolist(s->tail()) );

        static operation maptolist(s : Sequence(String) ) : String
        pre: true
        post: ( s.size = 0 => result = "" ) & ( s.size = 1 => result =
            s->first() ) & ( s.size > 1 => result = s->first() + "," +
            Expression.maptolist(s->tail()) );

        static operation indexstring(ax : Sequence(String) , aind : Set
            (String) ) : String
        pre: true
        post: ( aind.size > 0 & ax->includes("String") => result = "["
            + aind->any() + "]" ) & ( aind.size > 0 & not(( ax->
            includes("String") )) => result = "[" + aind->any() + "
            -1]" ) & ( aind.size = 0 => result = "" );

        static operation parstring(pars : Sequence(String) ) : String
        pre: true
        post: ( pars.size = 1 => result = "(" + pars->first() + ")" ) &
             ( pars.size > 1 => result = "(" + pars->first() + pars->
            tail()->collect( p | ", " + p )->sum() + ")" ) & ( pars.
            size = 0 => result = "()" );

        static operation leftBracket(tn : String ) : String
        pre: true
        post: ( tn = "Set" => result = "set({" ) & ( tn = "Sequence" =>
             result = "[" ) & ( tn = "Map" => result = "dict({" );

        static operation rightBracket(tn : String ) : String
        pre: true
        post: ( tn = "Set" => result = "})" ) & ( tn = "Sequence" =>
            result = "]" ) & ( tn = "Map" => result = "})" );

        operation addReference(x : BasicExpression ) : Expression
        pre: true
        post: true;

        operation isCollection() : boolean
        pre: true
        post: ( type->oclIsUndefined() => result = false ) & ( type.
            name = "Set" => result = true ) & ( type.name = "Sequence"
            => result = true );

        operation isMap() : boolean
        pre: true
        post: ( type->oclIsUndefined() => result = false ) & ( type.
            name = "Map" => result = true );

        operation isNumeric() : boolean
        pre: true
        post: ( type->oclIsUndefined() => result = false ) & ( type.
            name = "int" or type.name = "long" or type.name = "double"
            => result = true );

        operation isString() : boolean
        pre: true
        post: ( type->oclIsUndefined() => result = false ) & ( type.
            name = "String" => result = true );

        operation isEnumeration() : boolean
        pre: true
        post: ( type->oclIsUndefined() => result = false ) & (
            Enumeration->collect(name)->includes(type.name) => result =
             true );

        static operation isUnaryCollectionOp(fname : String ) : boolean
        pre: true
        post: ( fname = "->size" or fname = "->any" or fname = "->
            reverse" or fname = "->front" or fname = "->tail" or fname
            = "->first" or fname = "->last" or fname = "->sort" or
            fname = "->asSet" or fname = "->asSequence" or fname = "->
            asOrderedSet" or fname = "->asBag" ) => result = true;

        operation toPython() : String
        pre: true
        post: true;
```

```
    }

    class BasicExpression extends Expression {

        attribute data : String;
        attribute prestate : boolean;

        reference parameters[*] ordered : Expression;
        reference referredProperty[0-1] : Property;
        reference context[*] : Entity;
        reference arrayIndex[0-1] : Expression;
        reference objectRef[0-1] : Expression;

        static operation isMathFunction(fname : String ) : boolean
        pre: true
        post: ( fname = "sqrt" or fname = "exp" or fname = "log" or
            fname = "sin" or fname = "cos" or fname = "tan" or fname =
            "pow" or fname = "log10" or fname = "cbrt" or fname = "tanh
            " or fname = "cosh" or fname = "sinh" or fname = "asin" or
            fname = "acos" or fname = "atan" or fname = "ceil" or fname
             = "round" or fname = "floor" or fname = "abs" => result =
            true );

        operation noContextnoObject(obs : Set(String) ) : boolean
        pre: true
        post: obs.size = 0 & context.size = 0 => result = true;

        operation contextAndObject(obs : Set(String) ) : boolean
        pre: true
        post: context.size > 0 & obs.size > 0 => result = true;

        operation isOclExceptionCreation(obs : Set(String) ) : boolean
        pre: true
        post: ( data = "newOclException" or data = "
            newAssertionException" or data = "newProgramException" or
            data = "newSystemException" or data = "newIOException" or
            data = "newCastingException" or data = "
            newNullAccessException" or data = "newIndexingException" or
             data = "newArithmeticException" or data = "
            newIncorrectElementException" or data = "
            newAccessingException" or data = "newOclDate" ) => result =
             true;

        operation mapTypeExpression(ainds : Set(String) ) : String
        pre: true
        post: ( data = "long" => result = "int" ) & ( data = "double"
            => result = "float" ) & ( data = "boolean" => result = "
            bool" ) & ( data = "String" => result = "str" ) & ( data =
            "OclType" & ainds.size > 0 => result = "getOclTypeByPK(" +
            ainds->any() + ")" ) & ( data = "OclType" & ainds.size = 0
            => result = "type" ) & ( data = "OclVoid" => result = "None
            " ) & ( data = "OclAny" => result = "None" ) & ( data = "
            OclException" => result = "BaseException" ) & ( data = "
            ProgramException" => result = "Exception" ) & ( data = "
            SystemException" => result = "OSError" ) & ( data = "
            IOException" => result = "IOError" ) & ( data = "
            CastingException" => result = "TypeError" ) & ( data = "
            NullAccessException" => result = "AttributeError" ) & (
    data = "IndexingException" => result = "LookupError" ) & (
    data = "ArithmeticException" => result = "ArithmeticError"
    ) & ( data = "IncorrectElementException" => result = "
    ValueError" ) & ( data = "AssertionException" => result = "
    AssertionError" ) & ( data = "AccessingException" => result
     = "OSError" ) & ( true => result = data );

operation mapValueExpression(aind : Set(String) ) : String
pre: true
post: ( data = "true" => result = "True" ) & ( data = "false"
    => result = "False" ) & ( data = "null" => result = "None"
    ) & ( data = "Math_NaN" => result = "math.nan" ) & ( data =
     "Math_PINFINITY" => result = "math.inf" ) & ( data = "
    Math_NINFINITY" => result = "-math.inf" ) & ( aind.size > 0
     => result = data + "[" + aind->any() + "]" ) & ( self.
    isEnumeration() => result = type.name + "." + data ) & (
    true => result = data );

operation mapVariableExpression(obs : Set(String) , aind : Set(
    String) , pars : Sequence(String) ) : String
pre: true
post: ( obs.size = 0 & aind.size = 0 & pars.size = 0 => result
    = data ) & ( obs.size = 0 & aind.size = 0 & pars.size > 0
    => result = data + Expression.parstring(pars) ) & ( obs.
    size = 0 & aind.size > 0 & arrayIndex.type.name->includes("
    String") => result = data + "[" + aind->any() + "]" ) & (
    obs.size = 0 & aind.size > 0 => result = data + "[" + aind
    ->any() + " -1]" ) & ( obs.size > 0 => result = obs->any()
    + "." + mapVariableExpression(Set{},aind,pars) );

operation mapStaticAttributeExpression(obs : Set(String) , aind
     : Set(String) , pars : Sequence(String) ) : String
pre: true
post: ( obs.size = 0 => result = data + Expression.indexstring(
    arrayIndex->collect( expr | expr.type.name ),aind) ) & (
    contextAndObject(obs) => result = objectRef.any.data + "."
    + data + Expression.indexstring(arrayIndex->collect( expr |
     expr.type.name ),aind) );

operation mapReferencedAttributeExpression(obs : Set(String) ,
    aind : Set(String) , pars : Sequence(String) ) : String
pre: true
post: ( objectRef.any.type /: CollectionType => result = obs->
    any() + "." + data + Expression.indexstring(arrayIndex->
    collect( expr | expr.type.name ),aind) ) & ( type.name = "
    Sequence" => result = "[ _x." + data + Expression.
    indexstring(arrayIndex->collect( expr | expr.type.name ),
    aind) + " for _x in " + obs->any() + "]" ) & ( type.name =
    "Set" => result = "set({ _x." + data + Expression.
    indexstring(arrayIndex->collect( expr | expr.type.name ),
    aind) + " for _x in " + obs->any() + "})" );

operation mapAttributeExpression(obs : Set(String) , aind : Set
    (String) , pars : Sequence(String) ) : String
pre: true
post: ( isStatic = true => result =
    mapStaticAttributeExpression(obs,aind,pars) ) & (
    noContextnoObject(obs) => result = data + Expression.
```

```
        indexstring(arrayIndex->collect( expr | expr.type.name ),
        aind) ) & ( obs.size = 0 => result = "self." + data +
        Expression.indexstring(arrayIndex->collect( expr | expr.
        type.name ),aind) ) & ( contextAndObject(obs) => result =
        mapReferencedAttributeExpression(obs,aind,pars) ) & ( obs.
        size > 0 => result = obs->any() + "." + data + Expression.
        indexstring(arrayIndex->collect( expr | expr.type.name ),
        aind) );

    operation mapErrorCall(obs : Set(String) , pars : Sequence(
        String) ) : String
    pre: true
    post: ( obs.size > 0 => result = objectRef.any.
        mapTypeExpression(Set{}) + Expression.parstring(pars) ) & (
         obs.size = 0 => result = data );

    operation mapStaticOperationExpression(obs : Set(String) , aind
         : Set(String) , pars : Sequence(String) ) : String
    pre: true
    post: ( isOclExceptionCreation(obs) => result = mapErrorCall(
        obs,pars) ) & ( obs.size > 0 => result = objectRef.any.data
         + "." + data + Expression.parstring(pars) + Expression.
        indexstring(arrayIndex->collect( expr | expr.type.name ),
        aind) ) & ( obs.size = 0 => result = data + Expression.
        parstring(pars) + Expression.indexstring(arrayIndex->
        collect( expr | expr.type.name ),aind) );

    operation mapInstanceOperationExpression(obs : Set(String) ,
        aind : Set(String) , pars : Sequence(String) ) : String
    pre: true
    post: ( noContextnoObject(obs) => result = data + Expression.
        parstring(pars) + Expression.indexstring(arrayIndex->
        collect( expr | expr.type.name ),aind) ) & (
        contextAndObject(obs) => result = obs->any() + "." + data +
         Expression.parstring(pars) + Expression.indexstring(
        arrayIndex->collect( expr | expr.type.name ),aind) ) & (
        context.size > 0 => result = "self." + data + Expression.
        parstring(pars) + Expression.indexstring(arrayIndex->
        collect( expr | expr.type.name ),aind) ) & ( obs.size > 0
        => result = obs->any() + "." + data + Expression.parstring(
        pars) + Expression.indexstring(arrayIndex->collect( expr |
        expr.type.name ),aind) );

    operation mapOperationExpression(obs : Set(String) , aind : Set
        (String) , pars : Sequence(String) ) : String
    pre: true
    post: ( isStatic = true => result =
        mapStaticOperationExpression(obs,aind,pars) ) & ( isStatic
        = false => result = mapInstanceOperationExpression(obs,aind
        ,pars) );

    operation mapIntegerFunctionExpression(obs : Set(String) , aind
         : Set(String) , pars : Sequence(String) ) : String
    pre: obs.size > 0
    post: arg = obs->any() & ( "subrange" = data & pars.size() > 1
        => result = "range(" + pars->first() + ", " + pars->at(2) +
         " +1)" ) & ( "Sum" = data => result = "ocl.sum([(" + pars
        [4] + ") for " + pars[3] + " in range(" + pars[1] + ", " +
        pars[2] + " + 1)])" ) & ( "Prd" = data => result = "ocl.prd
        ([(" + pars[4] + ") for " + pars[3] + " in range(" + pars
        [1] + ", " + pars[2] + " + 1)])" );

    operation mapInsertAtFunctionExpression(obs : Set(String) ,
        aind : Set(String) , pars : Sequence(String) ) : String
    pre: obs.size > 0
    post: arg = obs->any() & ( type.name = "String" => result = "
        ocl.insertAtString(" + arg + ", " + pars[1] + ", " + pars
        [2] + ")" ) & ( type.name /= "String" => result = "ocl.
        insertAt(" + arg + ", " + pars[1] + ", " + pars[2] + ")" );

    operation mapSetAtFunctionExpression(obs : Set(String) , aind :
         Set(String) , pars : Sequence(String) ) : String
    pre: obs.size > 0
    post: arg = obs->any() & ( type.name = "String" => result = "
        ocl.setAtString(" + arg + ", " + pars[1] + ", " + pars[2] +
         ")" ) & ( type.name /= "String" => result = "ocl.setAt(" +
         arg + ", " + pars[1] + ", " + pars[2] + ")" );

    operation mapSubrangeFunctionExpression(obs : Set(String) ,
        aind : Set(String) , pars : Sequence(String) ) : String
    pre: obs.size > 0
    post: arg = obs->any() & ( pars->size() > 1 => result = arg +
        "[(" + pars[1] + "-1):" + pars[2] + "]" ) & ( pars->size()
        <= 1 => result = arg + "[(" + pars[1] + "-1):]" );

    operation mapFunctionExpression(obs : Set(String) , aind : Set(
        String) , pars : Sequence(String) ) : String
    pre: obs.size > 0
    post: arg = obs->any() & ( data = "allInstances" => result = "
        allInstances_" + arg + "()" ) & ( "Integer" = objectRef.any
        .data => result = mapIntegerFunctionExpression(obs,aind,
        pars) ) & ( data = "ceil" or data = "floor" => result = "
        int(math." + data + "(" + arg + "))" ) & ( data = "sqr" or
        data = "cbrt" => result = "ocl." + data + "(" + arg + ")" )
         & ( BasicExpression.isMathFunction(data) => result = "math
        ." + data + "(" + arg + ")" ) & ( data = "replace" =>
        result = "ocl.replace(" + arg + ", " + pars[1] + ", " +
        pars[2] + ")" ) & ( data = "replaceFirstMatch" => result =
        "ocl.replaceFirstMatch(" + arg + ", " + pars[1] + ", " +
        pars[2] + ")" ) & ( data = "replaceAllMatches" => result =
        "ocl.replaceAllMatches(" + arg + ", " + pars[1] + ", " +
        pars[2] + ")" ) & ( data = "replaceAll" => result = "ocl.
        replaceAll(" + arg + ", " + pars[1] + ", " + pars[2] + ")"
        ) & ( data = "insertAt" => result =
        mapInsertAtFunctionExpression(obs,aind,pars) ) & ( data = "
        insertInto" => result = "ocl.insertInto(" + arg + ", " +
        pars[1] + ", " + pars[2] + ")" ) & ( data = "setAt" =>
        result = mapSetAtFunctionExpression(obs,aind,pars) ) & (
        data = "oclIsUndefined" => result = "(" + arg + " == None)"
         ) & ( data = "oclAsType" => result = pars->first() + "(" +
         arg + ")" ) & ( data = "sum" => result = "ocl.sum(" + arg
        + ")" ) & ( data = "prd" => result = "ocl.prd(" + arg + ")"
         ) & ( data = "max" => result = "ocl.max" + type.name + "("
         + arg + ")" ) & ( data = "min" => result = "ocl.min" +
        type.name + "(" + arg + ")" ) & ( data = "front" => result
        = "(" + arg + ")[0:-1]" ) & ( data = "tail" => result = "("
        + arg + ")[1:]" ) & ( data = "first" => result = "(" + arg
        + ")[0]" ) & ( data = "last" => result = "(" + arg + ")
        [-1]" ) & ( data = "sort" => result = "ocl.sort" + type.
        name + "(" + arg + ")" ) & ( data = "size" => result = "len
        (" + arg + ")" ) & ( data = "toLowerCase" => result = "ocl.
        toLowerCase(" + arg + ")" ) & ( data = "toUpperCase" =>
        result = "ocl.toUpperCase(" + arg + ")" ) & ( data = "
        reverse" => result = "ocl.reverse" + type.name + "(" + arg
        + ")" ) & ( data = "subrange" => result =
        mapSubrangeFunctionExpression(obs,aind,pars) ) & ( true =>
        result = "ocl." + data + "(" + arg + ")" );

    operation mapClassArrayExpression(obs : Set(String) , aind :
        Set(String) , pars : Sequence(String) ) : String
    pre: umlKind = classid
    post: ( arrayIndex.any.type : CollectionType => result = "get"
        + elementType.name + "ByPKs(" + aind->any() + ")" ) & (
        arrayIndex.any.type /: CollectionType => result = "get" +
        elementType.name + "ByPK(" + aind->any() + ")" );

    operation mapClassExpression(obs : Set(String) , aind : Set(
        String) , pars : Sequence(String) ) : String
    pre: umlKind = classid
    post: ( arrayIndex.size > 0 => result = mapClassArrayExpression
        (obs,aind,pars) ) & ( arrayIndex.size = 0 => result = "
        allInstances_" + data + "()" );

    operation mapBasicExpression(ob : Set(String) , aind : Set(
        String) , pars : Sequence(String) ) : String
    pre: true
    post: ( data = "skip" => result = "pass" ) & ( data = "super"
        => result = "super()" ) & ( PrimitiveType.isPrimitiveType(
        data) => result = mapTypeExpression(aind) ) & ( umlKind =
        value => result = mapValueExpression(aind) ) & ( umlKind =
        variable => result = mapVariableExpression(ob,aind,pars) )
        & ( umlKind = attribute or umlKind = role => result =
        mapAttributeExpression(ob,aind,pars) ) & ( umlKind =
        operation => result = mapOperationExpression(ob,aind,pars)
        ) & ( umlKind = function => result = mapFunctionExpression(
        ob,aind,pars) ) & ( umlKind = classid => result =
        mapClassExpression(ob,aind,pars) );

    operation toPython() : String
    pre: true
    post: result = mapBasicExpression(objectRef.toPython(),
        arrayIndex.toPython(),parameters.toPython());

    operation addClassIdReference(x : BasicExpression ) :
        Expression
    pre: true
    post: BasicExpression->exists( e | e.expId = expId + "_" + x.
        data & e.data = data & e.prestate = prestate & e.umlKind =
        umlKind & e.objectRef = objectRef & e.arrayIndex =
        arrayIndex.addReference(x) & e.parameters = parameters.
        addReference(x) & result = e );

    operation addBEReference(x : BasicExpression ) : Expression
    pre: true
```

```
        post: BasicExpression->exists( e | e.expId = expId + "_" + x.
            data & e.data = data & e.prestate = prestate & e.umlKind =
            umlKind & e.objectRef = Set{x} & e.arrayIndex = arrayIndex.
            addReference(x) & e.parameters = parameters.addReference(x)
             & result = e );

        operation addSelfBEReference(x : BasicExpression ) : Expression
        pre: true
        post: BasicExpression->exists( e | e.expId = expId + "_" + x.
            data & e.data = data & e.prestate = prestate & e.umlKind =
            umlKind & e.objectRef = Set{x} & e.arrayIndex = arrayIndex.
            addReference(x) & e.parameters = parameters.addReference(x)
             & result = e );

        operation addObjectRefBEReference(x : BasicExpression ) :
            Expression
        pre: true
        post: BasicExpression->exists( e | e.expId = expId + "_" + x.
            data & e.data = data & e.prestate = prestate & e.umlKind =
            umlKind & e.objectRef = objectRef.addReference(x) & e.
            arrayIndex = arrayIndex.addReference(x) & e.parameters =
            parameters.addReference(x) & result = e );

        operation addReference(x : BasicExpression ) : Expression
        pre: true
        post: result = (if ( umlKind = classid ) then
            addClassIdReference(x) else if ( objectRef.size = 0 ) then
            addBEReference(x) else if ( "self" = objectRef->any() + ""
            ) then addSelfBEReference(x) else addObjectRefBEReference(x
            ) endif endif endif);
    }

    class BinaryExpression extends Expression {

        attribute operator : String;
        attribute variable : String;

        reference left : Expression;
        reference right : Expression;
        reference accumulator : Property;

        static operation isComparitor(fname : String ) : boolean
        pre: true
        post: ( fname = "=" or fname = "/=" or fname = "<>" or fname =
            "<" or fname = ">" or fname = "<=" or fname = ">=" or fname
             = "<>=" => result = true );

        static operation isInclusion(fname : String ) : boolean
        pre: true
        post: ( fname = ":" or fname = "->includes" or fname = "<:" or
            fname = "->includesAll" => result = true );

        static operation isExclusion(fname : String ) : boolean
        pre: true
        post: ( fname = "/:" or fname = "/<:" or fname = "->excludes"
            or fname = "->excludesAll" => result = true );

        static operation isBooleanOp(fname : String ) : boolean
```

```
pre: true
post: ( fname = "&" or fname = "or" or fname = "=>" or fname =
    "->exists" or fname = "->forAll" or fname = "->exists1" =>
    result = true );

static operation isStringOp(fname : String ) : boolean
pre: true
post: ( fname = "->indexOf" or fname = "->count" or fname = "->
    hasPrefix" or fname = "->hasSuffix" or fname = "->after" or
     fname = "->before" or fname = "->lastIndexOf" or fname =
    "->equalsIgnoreCase" or fname = "->split" or fname = "->
    isMatch" or fname = "->hasMatch" or fname = "->allMatches"
    or fname = "->firstMatch" or fname = "->excludingAt" =>
    result = true );

static operation isCollectionOp(fname : String ) : boolean
pre: true
post: ( fname = "->including" or fname = "->excluding" or fname
     = "->excludingAt" or fname = "->excludingFirst" or fname =
     "->append" or fname = "->count" or fname = "->indexOf" or
    fname = "->lastIndexOf" or fname = "->union" or fname = "->
    intersection" or fname = "^" or fname = "->isUnique" or
    fname = "->at" ) => result = true;

static operation isDistributedIteratorOp(fname : String ) :
    boolean
pre: true
post: ( fname = "->sortedBy" or fname = "->unionAll" or fname =
     "->concatenateAll" or fname = "->intersectAll" or fname =
    "->selectMinimals" or fname = "->selectMaximals" => result
    = true );

static operation isIteratorOp(fname : String ) : boolean
pre: true
post: ( fname = "->collect" or fname = "->select" or fname =
    "->reject" or fname = "->iterate" ) => result = true;

operation mapDividesExpression(ls : String , rs : String ) :
    String
pre: true
post: ( type.name = "int" or type.name = "long" => result = ls
    + "/" + rs ) & ( true => result = ls + "/" + rs );

operation mapNumericExpression(ls : String , rs : String ) :
    String
pre: true
post: ( operator = "/" => result = mapDividesExpression(ls,rs)
    ) & ( operator = "mod" => result = ls + " % " + rs ) & (
    operator = "div" => result = ls + " / " + rs ) & ( operator
     = "=" => result = ls + " == " + rs ) & ( operator = "<>="
    => result = ls + " is " + rs ) & ( operator = "/=" =>
    result = ls + " != " + rs ) & ( operator = "->pow" =>
    result = "math.pow(" + ls + ", " + rs + ")" ) & ( operator
    = "->gcd" => result = "ocl.gcd(" + ls + ", " + rs + ")" ) &
     ( needsBracket => result = "(" + ls + " " + operator + " "
     + rs + ")" ) & ( true => result = ls + " " + operator + "
    " + rs );
```

```
operation mapComparitorExpression(ls : String , rs : String ) :
     String
pre: true
post: ( operator = "=" => result = ls + " == " + rs ) & (
    operator = "<>=" => result = ls + " is " + rs ) & (
    operator = "/=" => result = ls + " != " + rs ) & ( operator
     = "<>" => result = ls + " != " + rs ) & ( needsBracket =>
    result = "(" + ls + " " + operator + " " + rs + ")" ) & (
    true => result = ls + " " + operator + " " + rs );

operation mapStringExpression(ls : String , rs : String ) :
    String
pre: true
post: ( operator = "=" => result = ls + " == " + rs ) & (
    operator = "<>=" => result = ls + " is " + rs ) & (
    operator = "/=" => result = ls + " != " + rs ) & ( operator
     = ":" => result = ls + " in " + rs ) & ( operator = "/:"
    => result = ls + " not in " + rs ) & ( operator = "->
    includes" => result = rs + " in " + ls ) & ( operator = "->
    excludes" => result = rs + " not in " + ls ) & ( operator =
     "-" => result = "ocl.subtractString(" + ls + ", " + rs +
    ")" ) & ( operator = "->hasPrefix" => result = ls + ".
    startswith(" + rs + ")" ) & ( operator = "->hasSuffix" =>
    result = ls + ".endswith(" + rs + ")" ) & ( operator = "->
    equalsIgnoreCase" => result = "ocl.equalsIgnoreCase(" + ls
    + "," + rs + ")" ) & ( operator = "->indexOf" => result =
    "(" + ls + ".find(" + rs + ") + 1)" ) & ( operator = "->
    lastIndexOf" => result = "ocl.lastIndexOf(" + ls + ", " +
    rs + ")" ) & ( operator = "->count" => result = ls + ".
    count(" + rs + ")" ) & ( operator = "->isMatch" => result =
     "ocl.isMatch(" + ls + ", " + rs + ")" ) & ( operator = "->
    hasMatch" => result = "ocl.hasMatch(" + ls + ", " + rs + ")
    " ) & ( operator = "->allMatches" => result = "ocl.
    allMatches(" + ls + ", " + rs + ")" ) & ( operator = "->
    firstMatch" => result = "ocl.firstMatch(" + ls + ", " + rs
    + ")" ) & ( operator = "->before" => result = "ocl.before("
     + ls + ", " + rs + ")" ) & ( operator = "->after" =>
    result = "ocl.after(" + ls + ", " + rs + ")" ) & ( operator
     = "->split" => result = "ocl.split(" + ls + ", " + rs + ")
    " ) & ( operator = "->excludingAt" => result = "ocl.
    removeAtString(" + ls + ", " + rs + ")" ) & ( true =>
    result = ls + " " + operator + " " + rs );

operation mapStringPlus(ls : String , rs : String ) : String
pre: true
post: ( left.isString() => result = ls + " + str(" + rs + ")" )
     & ( right.isString() => result = "str(" + ls + ") + " + rs
     );

operation mapBooleanExpression(ls : String , rs : String ) :
    String
pre: true
post: ( operator = "&" => result = ls + " and " + rs ) & (
    operator = "=>" => result = "((" + rs + ") if (" + ls + ")
    else (True))" ) & ( operator = "or" => result = ls + " or "
     + rs ) & ( operator = "->exists" => result = "ocl.exists("
     + ls + ", lambda " + variable + " : " + rs + ")" ) & (
    operator = "->forAll" => result = "ocl.forAll(" + ls + ",
        lambda " + variable + " : " + rs + ")" ) & ( operator = "->
        exists1" => result = "ocl.exists1(" + ls + ", lambda " +
        variable + " : " + rs + ")" ) & ( true => result = ls + " "
         + operator + " " + rs );

    operation mapBinaryCollectionExpression(ls : String , rs :
        String , lt : String , rt : String ) : String
    pre: true
    post: ( operator = ":" => result = ls + " in " + rs ) & (
        operator = "->includes" => result = rs + " in " + ls ) & (
        operator = "=" => result = ls + " == " + rs ) & ( operator
        = "<>=" => result = ls + " is " + rs ) & ( operator = "/="
        => result = ls + " != " + rs ) & ( operator = "/:" =>
        result = ls + " not in " + rs ) & ( operator = "->excludes"
         => result = rs + " not in " + ls ) & ( operator = "<:" =>
        result = "ocl.includesAll(" + rs + ", " + ls + ")" ) & (
        operator = "/<:" => result = "(not ocl.includesAll(" + rs +
         ", " + ls + "))" ) & ( operator = "->includesAll" =>
        result = "ocl.includesAll(" + ls + ", " + rs + ")" ) & (
        operator = "->excludesAll" => result = "ocl.excludesAll(" +
         rs + ", " + ls + ")" ) & ( operator = "->including" =>
        result = "ocl.including" + lt + "(" + ls + ", " + rs + ")"
        ) & ( operator = "->excluding" => result = "ocl.excluding"
        + lt + "(" + ls + ", " + rs + ")" ) & ( operator = "->
        excludingFirst" => result = "ocl.excludingFirst(" + ls + ",
         " + rs + ")" ) & ( operator = "->excludingAt" => result =
        "ocl.removeAt(" + ls + ", " + rs + ")" ) & ( operator = "->
        union" => result = "ocl.union" + lt + "(" + ls + ", " + rs
        + ")" ) & ( operator = "->intersection" => result = "ocl.
        intersection" + lt + "(" + ls + ", " + rs + ")" ) & (
        operator = "^" => result = "ocl.concatenate(" + ls + ", " +
         rs + ")" ) & ( operator = "->prepend" => result = "ocl.
        prepend(" + ls + ", " + rs + ")" ) & ( operator = "->append
        " => result = "ocl.append(" + ls + ", " + rs + ")" ) & (
        operator = "-" => result = "ocl.excludeAll" + lt + "(" + ls
         + ", " + rs + ")" ) & ( operator = "->indexOf" & right.
        type.name = "Sequence" => result = "ocl.indexOfSubSequence
        (" + ls + "," + rs + ")" ) & ( operator = "->indexOf" &
        right.type.name /= "Sequence" => result = "ocl.
        indexOfSequence(" + ls + ", " + rs + ")" ) & ( operator =
        "->lastIndexOf" & right.type.name = "Sequence" => result =
        "ocl.lastIndexOfSubSequence(" + ls + ", " + rs + ")" ) & (
        operator = "->lastIndexOf" & right.type.name /= "Sequence"
        => result = "ocl.lastIndexOfSequence(" + ls + ", " + rs +
        ")" ) & ( operator = "->count" => result = "(" + ls + ").
        count(" + rs + ")" ) & ( operator = "->at" => result = "("
        + ls + ")[" + rs + " - 1]" ) & ( true => result = ls + " "
        + operator + " " + rs );

    operation mapBinaryMapExpression(ls : String , rs : String ) :
        String
    pre: true
    post: ( operator = ":" => result = ls + " in " + rs ) & (
        operator = "->includes" => result = rs + " in " + ls ) & (
        operator = "=" => result = ls + " == " + rs ) & ( operator
        = "<>=" => result = ls + " is " + rs ) & ( operator = "/="
        => result = ls + " != " + rs ) & ( operator = "/:" =>
        result = ls + " not in " + rs ) & ( operator = "->excludes"
```

```
         => result = rs + " not in " + ls ) & ( operator = "<:" =>
        result = "ocl.includesAllMap(" + rs + ", " + ls + ")" ) & (
         operator = "/<:" => result = "(not ocl.includesAllMap(" +
        rs + ", " + ls + "))" ) & ( operator = "->includesAll" =>
        result = "ocl.includesAllMap(" + ls + ", " + rs + ")" ) & (
         operator = "->excludesAll" => result = "ocl.excludesAllMap
        (" + rs + ", " + ls + ")" ) & ( operator = "->including" =>
         result = "ocl.includingMap(" + ls + ", " + rs + ", " +
        variable + ")" ) & ( operator = "->excluding" => result = "
        ocl.excludingMapValue(" + ls + ", " + rs + ")" ) & (
        operator = "->excludingAt" => result = "ocl.excludingAtMap
        (" + ls + ", " + rs + ")" ) & ( operator = "->union" =>
        result = "ocl.unionMap(" + ls + ", " + rs + ")" ) & (
        operator = "->intersection" => result = "ocl.
        intersectionMap(" + ls + ", " + rs + ")" ) & ( operator =
        "->restrict" => result = "ocl.restrict(" + ls + ", " + rs +
         ")" ) & ( operator = "->antirestrict" => result = "ocl.
        antirestrict(" + ls + ", " + rs + ")" ) & ( operator = "-"
        => result = "ocl.excludeAllMap(" + ls + ", " + rs + ")" ) &
         ( operator = "->at" => result = "(" + ls + ")[" + rs + "]"
         ) & ( true => result = ls + " " + operator + " " + rs );

    operation mapDistributedIteratorExpression(ls : String , rs :
        String , rexp : Expression ) : String
    pre: true
    post: ( operator = "->sortedBy" => result = "sorted(" + ls + ",
         key = lambda " + expId + " : " + rexp.toPython() + ")" ) &
         ( operator = "->concatenateAll" => result = "ocl.
        concatenateAll([" + rexp.toPython() + " for " + expId + "
        in " + ls + "])" ) & ( operator = "->unionAll" & type.name
        = "Sequence" => result = "ocl.concatenateAll([" + rexp.
        toPython() + " for " + expId + " in " + ls + "])" ) & (
        operator = "->unionAll" & type.name = "Set" => result = "
        ocl.unionAll([" + rexp.toPython() + " for " + expId + " in
        " + ls + "])" ) & ( operator = "->intersectAll" => result =
         "ocl.intersectAll" + type.name + "([" + rexp.toPython() +
        " for " + expId + " in " + ls + "])" ) & ( operator = "->
        selectMaximals" => result = "ocl.selectMaximals" + left.
        type.name + "(" + ls + ", lambda " + expId + " : " + rexp.
        toPython() + ")" ) & ( operator = "->selectMinimals" =>
        result = "ocl.selectMinimals" + left.type.name + "(" + ls +
         ", lambda " + expId + " : " + rexp.toPython() + ")" );

    operation mapMapIteratorExpression(ls : String , rs : String )
        : String
    pre: true
    post: ( operator = "->select" => result = "ocl.selectMap(" + ls
         + ", lambda " + variable + " : " + rs + ")" ) & ( operator
         = "->reject" => result = "ocl.rejectMap(" + ls + ", lambda
         " + variable + " : " + rs + ")" ) & ( operator = "->
        collect" => result = "ocl.collectMap(" + ls + ", lambda " +
         variable + " : " + rs + ")" );

    operation mapIteratorExpression(ls : String , rs : String , tn
        : String ) : String
    pre: true
    post: lbracket = Expression.leftBracket(tn) & rbracket =
        Expression.rightBracket(tn) & ( tn = "Map" => result =
        mapMapIteratorExpression(ls,rs) ) & ( operator = "->iterate
        " => result = "ocl.iterate(" + ls + "," + accumulator.
        initialValue.toPython() + ", lambda " + variable + ", " +
        accumulator.name + " : (" + rs + "))" ) & ( operator = "->
        select" => result = lbracket + variable + " for " +
        variable + " in " + ls + " if " + rs + rbracket ) & (
        operator = "->reject" => result = lbracket + variable + "
        for " + variable + " in " + ls + " if not " + rs + rbracket
         ) & ( operator = "->collect" => result = "[" + rs + " for
        " + variable + " in " + ls + "]" );

    operation mapTypeCastExpression(ls : String , rs : String ) :
        String
    pre: true
    post: ( PrimitiveType.isPythonPrimitiveType(rs) => result =
        type.toPython() + "(" + ls + ")" ) & ( true => result = ls
        );

    operation mapCatchExpression(ls : String , rs : String ) :
        String
    pre: true
    post: ( ":" = operator => result = right.mapTypeExpression(Set
        {}) + " as " + ls ) & ( true => result =
        mapBinaryExpression(ls,rs) );

    operation bothNumeric() : boolean
    pre: true
    post: left.isNumeric() & right.isNumeric() => result = true;

    operation bothString() : boolean
    pre: true
    post: left.isString() & right.isString() => result = true;

    operation isStringPlus() : boolean
    pre: true
    post: operator = "+" & ( left.isString() or right.isString() )
        => result = true;

    operation mapBinaryExpression(ls : String , rs : String ) :
        String
    pre: true
    post: ( operator = ":" => result = ls + " in " + rs ) & (
        operator = "->apply" => result = "(" + ls + ")(" + rs + ")"
         ) & ( operator = "->oclAsType" => result =
        mapTypeCastExpression(ls,rs) ) & ( operator = "->
        oclIsTypeOf" => result = "(type(" + ls + ") == " + rs + ")"
         ) & ( operator = "->oclIsKindOf" => result = ls + " in " +
         rs ) & ( operator = "->compareTo" => result = "ocl.
        compareTo(" + ls + "," + rs + ")" ) & ( operator = "|->" =>
         result = ls + ":" + rs ) & ( operator = "->restrict" =>
        result = "ocl.restrict(" + ls + "," + rs + ")" ) & (
        operator = "->antirestrict" => result = "ocl.antirestrict("
         + ls + "," + rs + ")" ) & ( BinaryExpression.isComparitor(
        operator) => result = mapComparitorExpression(ls,rs) ) & (
        BinaryExpression.isDistributedIteratorOp(operator) =>
        BasicExpression->exists( be | be.data = expId & be.expId =
        expId + "_variable" & result =
        mapDistributedIteratorExpression(ls,rs,right.addReference(
        be)) ) ) & ( bothNumeric() => result = mapNumericExpression
        (ls,rs) ) & ( bothString() => result = mapStringExpression(
        ls,rs) ) & ( left.isString() & operator = "->excludingAt"
        => result = "ocl.removeAtString(" + ls + "," + rs + ")" ) &
         ( BinaryExpression.isIteratorOp(operator) => result =
        mapIteratorExpression(ls,rs,left.type.name) ) & (
        BinaryExpression.isBooleanOp(operator) => result =
        mapBooleanExpression(ls,rs) ) & ( isStringPlus() => result
        = mapStringPlus(ls,rs) ) & ( ( left.isCollection() or right
        .isCollection() ) => result = mapBinaryCollectionExpression
        (ls,rs,left.type.name,right.type.name) ) & ( ( left.isMap()
         or right.isMap() ) => result = mapBinaryMapExpression(ls,
        rs) ) & ( operator = "->at" => result = "(" + ls + ")[" +
        rs + "]" ) & ( true => result = ls + " " + operator + " " +
         rs );

    operation toPython() : String
    pre: true
    post: result = mapBinaryExpression(left.toPython(),right.
        toPython());

    operation addReference(x : BasicExpression ) : Expression
    pre: x.data /= self.variable
    post: BinaryExpression->exists( e | e.expId = expId + "_" + x.
        data & e.operator = operator & e.variable = self.variable &
         e.umlKind = umlKind & e.left = left.addReference(x) & e.
        right = right.addReference(x) & result = e );
}

class ConditionalExpression extends Expression {

    reference test : Expression;
    reference ifExp : Expression;
    reference elseExp : Expression;

    operation mapConditionalExpression(ts : String , ls : String ,
        rs : String ) : String
    pre: true
    post: result = "(" + ls + " if " + ts + " else " + rs + ")";

    operation toPython() : String
    pre: true
    post: result = mapConditionalExpression(test.toPython(),ifExp.
        toPython(),elseExp.toPython());

    operation addReference(x : BasicExpression ) : Expression
    pre: true
    post: ConditionalExpression->exists( e | e.expId = expId + "_"
        + x.data & e.test = test.addReference(x) & e.umlKind =
        umlKind & e.ifExp = ifExp.addReference(x) & e.elseExp =
        elseExp.addReference(x) & result = e );
}

class UnaryExpression extends Expression {

    attribute operator : String;
    attribute variable : String;
```

```
reference argument : Expression;

static operation isUnaryStringOp(fname : String ) : boolean
pre: true
post: ( fname = "->size" or fname = "->first" or fname = "->
    last" or fname = "->front" or fname = "->tail" or fname =
    "->reverse" or fname = "->display" or fname = "->
    toUpperCase" or fname = "->toLowerCase" or fname = "->
    toInteger" or fname = "->toReal" or fname = "->toLong" or
    fname = "->toBoolean" or fname = "->trim" => result = true
    );

static operation isReduceOp(fname : String ) : boolean
pre: true
post: ( fname = "->min" or fname = "->max" or fname = "->sum"
    or fname = "->prd" ) => result = true;

operation mapNumericExpression(arg : String ) : String
pre: true
post: ( operator = "->sqrt" => result = "math.sqrt(" + arg + ")
    " ) & ( operator = "->sin" => result = "math.sin(" + arg +
    ")" ) & ( operator = "->tan" => result = "math.tan(" + arg
    + ")" ) & ( operator = "->cos" => result = "math.cos(" +
    arg + ")" ) & ( operator = "->sqr" => result = "ocl.sqr(" +
     arg + ")" ) & ( operator = "->floor" => result = "int(math
    .floor(" + arg + "))" ) & ( operator = "->ceil" => result =
     "int(math.ceil(" + arg + "))" ) & ( operator = "->round"
    => result = "round(" + arg + ")" ) & ( operator = "->cbrt"
    => result = "ocl.cbrt(" + arg + ")" ) & ( operator = "->
    display" => result = "print(str(" + arg + "))" ) & (
    operator = "->abs" => result = "math.fabs(" + arg + ")" ) &
     ( operator = "->exp" => result = "math.exp(" + arg + ")" )
     & ( operator = "->log" => result = "math.log(" + arg + ")"
     ) & ( operator = "->log10" => result = "math.log10(" + arg
     + ")" ) & ( operator = "->asin" => result = "math.asin(" +
     arg + ")" ) & ( operator = "->acos" => result = "math.acos
    (" + arg + ")" ) & ( operator = "->atan" => result = "math.
    atan(" + arg + ")" ) & ( operator = "->sinh" => result = "
    math.sinh(" + arg + ")" ) & ( operator = "->cosh" => result
     = "math.cosh(" + arg + ")" ) & ( operator = "->tanh" =>
    result = "math.tanh(" + arg + ")" );

operation mapStringExpression(arg : String ) : String
pre: true
post: ( operator = "->size" => result = "len(" + arg + ")" ) &
    ( operator = "->front" => result = "(" + arg + ")[0:-1]" )
    & ( operator = "->tail" => result = "(" + arg + ")[1:]" ) &
     ( operator = "->first" => result = "(" + arg + ")[0:1]" )
    & ( operator = "->last" => result = "(" + arg + ")[-1:]" )
    & ( operator = "->toLowerCase" => result = "ocl.toLowerCase
    (" + arg + ")" ) & ( operator = "->toUpperCase" => result =
     "ocl.toUpperCase(" + arg + ")" ) & ( operator = "->
    characters" => result = "ocl.characters(" + arg + ")" ) & (
     operator = "->trim" => result = "ocl.trim(" + arg + ")" )
    & ( operator = "->reverse" => result = "ocl.reverseString("
     + arg + ")" ) & ( operator = "->display" => result = "
    print(" + arg + ")" ) & ( operator = "->isInteger" =>
    result = "isdigit(" + arg + ")" ) & ( operator = "->isLong"
```

```
          => result = "isdigit(" + arg + ")" ) & ( operator = "->
         isReal" => result = "isdecimal(" + arg + ")" ) & ( operator
          = "->toInteger" => result = "ocl.toInteger(" + arg + ")" )
          & ( operator = "->toLong" => result = "ocl.toInteger(" +
         arg + ")" ) & ( operator = "->toReal" => result = "float("
         + arg + ")" ) & ( operator = "->toBoolean" => result = "ocl
         .toBoolean(" + arg + ")" );

    operation mapReduceExpression(arg : String , tn : String , et :
         Type ) : String
    pre: true
    post: ( operator = "->sum" & et->oclIsUndefined() => result = "
         ocl.sum(" + arg + ")" ) & ( operator = "->sum" & et.
         isStringType() => result = "ocl.sumString(" + arg + ")" ) &
          ( operator = "->sum" & not((et.isStringType())) => result
         = "ocl.sum(" + arg + ")" ) & ( operator = "->prd" => result
          = "ocl.prd(" + arg + ")" ) & ( operator = "->max" =>
         result = "ocl.max" + tn + "(" + arg + ")" ) & ( operator =
         "->min" => result = "ocl.min" + tn + "(" + arg + ")" );

    operation mapUnaryCollectionExpression(arg : String , tn :
         String ) : String
    pre: true
    post: ( operator = "->size" => result = "len(" + arg + ")" ) &
         ( operator = "->isEmpty" => result = "(len(" + arg + ") ==
         0)" ) & ( operator = "->notEmpty" => result = "(len(" + arg
          + ") > 0)" ) & ( operator = "->asSet" => result = "set(" +
          arg + ")" ) & ( operator = "->asSequence" => result = "
         list(" + arg + ")" ) & ( operator = "->asOrderedSet" =>
         result = "ocl.asOrderedSet(" + arg + ")" ) & ( operator =
         "->asBag" => result = "ocl.sortSequence(" + arg + ")" ) & (
          operator = "->unionAll" & type.name = "Sequence" => result
          = "ocl.concatenateAll(" + arg + ")" ) & ( operator = "->
         unionAll" & type.name = "Set" => result = "ocl.unionAll(" +
          arg + ")" ) & ( operator = "->intersectAll" => result = "
         ocl.intersectAll" + type.name + "(" + arg + ")" ) & (
         operator = "->concatenateAll" => result = "ocl.
         concatenateAll(" + arg + ")" ) & ( operator = "->isUnique"
         => result = "ocl.isUnique(" + arg + ")" ) & ( operator =
         "->isDeleted" => result = "del " + arg ) & ( operator = "->
         copy" => result = "copy.copy(" + arg + ")" ) & ( operator =
          "->any" => result = "ocl.any(" + arg + ")" ) & ( operator
         = "->reverse" => result = "ocl.reverse" + tn + "(" + arg +
         ")" ) & ( operator = "->front" => result = "(" + arg + ")
         [0:-1]" ) & ( operator = "->tail" => result = "(" + arg +
         ")[1:]" ) & ( operator = "->first" => result = "(" + arg +
         ")[0]" ) & ( operator = "->last" => result = "(" + arg + ")
         [-1]" ) & ( operator = "->sort" => result = "ocl.sort" + tn
          + "(" + arg + ")" ) & ( operator = "->display" => result =
          "print(str(" + arg + "))" );

    operation mapUnaryMapExpression(arg : String , tn : String ) :
         String
    pre: true
    post: ( operator = "->size" => result = "len(" + arg + ")" ) &
         ( operator = "->isEmpty" => result = "(len(" + arg + ") ==
         0)" ) & ( operator = "->notEmpty" => result = "(len(" + arg
          + ") > 0)" ) & ( operator = "->asSet" => result = "set(" +
          arg + ")" ) & ( operator = "->asSequence" => result = "
          list(" + arg + ")" ) & ( operator = "->asOrderedSet" =>
          result = "ocl.asOrderedSet(" + arg + ")" ) & ( operator =
          "->asBag" => result = "ocl.sortSequence(" + arg + ")" ) & (
           operator = "->unionAll" & type.name = "Sequence" => result
           = "ocl.concatenateAll(" + arg + ")" ) & ( operator = "->
          unionAll" & type.name = "Set" => result = "ocl.unionAll(" +
           arg + ")" ) & ( operator = "->intersectAll" => result = "
          ocl.intersectAll" + type.name + "(" + arg + ")" ) & (
          operator = "->concatenateAll" => result = "ocl.
          concatenateAll(" + arg + ")" ) & ( operator = "->isUnique"
          => result = "ocl.isUnique(" + arg + ")" ) & ( operator =
          "->isDeleted" => result = "del " + arg ) & ( operator = "->
          copy" => result = "copy.copy(" + arg + ")" ) & ( operator =
           "->any" => result = "ocl.anyMap(" + arg + ")" ) & (
          operator = "->reverse" => result = "ocl.reverse" + tn + "("
           + arg + ")" ) & ( operator = "->front" => result = "(" +
          arg + ")[0:-1]" ) & ( operator = "->tail" => result = "(" +
           arg + ")[1:]" ) & ( operator = "->first" => result = "(" +
           arg + ")[0]" ) & ( operator = "->last" => result = "(" +
          arg + ")[-1]" ) & ( operator = "->sort" => result = "ocl.
          sort" + tn + "(" + arg + ")" ) & ( operator = "->display"
          => result = "print(str(" + arg + "))" );

      operation mapUnaryExpression(arg : String ) : String
      pre: true
      post: ( operator = "-" => result = "-" + arg ) & ( operator =
          "?" => result = "id(" + arg + ")" ) & ( operator = "not" =>
           result = "not " + arg ) & ( operator = "->isDeleted" =>
          result = "del " + arg ) & ( operator = "->copy" => result =
           "copy.copy(" + arg + ")" ) & ( operator = "->
          oclIsUndefined" => result = "(" + arg + " == None)" ) & (
          operator = "->oclIsInvalid" => result = "math.isnan(" + arg
           + ")" ) & ( operator = "->oclType" => result = "type(" +
          arg + ")" ) & ( operator = "lambda" => result = "lambda " +
           variable + " : " + arg ) & ( operator = "->byte2char" =>
          result = "chr(" + arg + ")" ) & ( operator = "->char2byte"
          => result = "ord(" + arg + ")" ) & ( operator = "->keys" =>
           result = "ocl.keys(" + arg + ")" ) & ( operator = "->
          values" => result = "ocl.values(" + arg + ")" ) & (
          argument.isNumeric() => result = mapNumericExpression(arg)
          ) & ( argument.isString() => result = mapStringExpression(
          arg) ) & ( UnaryExpression.isReduceOp(operator) => result =
           mapReduceExpression(arg,argument.type.name,argument.
          elementType) ) & ( argument.isCollection() => result =
          mapUnaryCollectionExpression(arg,argument.type.name) ) & (
          argument.isMap() => result = mapUnaryMapExpression(arg,
          argument.type.name) ) & ( operator = "->display" => result
          = "print(str(" + arg + "))" ) & ( operator = "->toBoolean"
          => result = "ocl.toBoolean(" + arg + ")" );

      operation toPython() : String
      pre: true
      post: result = mapUnaryExpression(argument.toPython());

      operation addReference(x : BasicExpression ) : Expression
      pre: true
```

```
        post: UnaryExpression->exists( e | e.expId = expId + "_" + x.
            data & e.operator = operator & e.umlKind = umlKind & e.
            argument = argument.addReference(x) & result = e );
    }

    class CollectionExpression extends Expression {

        attribute isOrdered : boolean;

        reference elements[*] : Expression;

        operation mapCollectionExpression(elems : Sequence(String) , tn
             : String ) : String
        pre: true
        post: ( tn = "Set" => result = "set({" + Expression.tolist(
            elems) + "})" ) & ( tn = "Sequence" => result = "[" +
            Expression.tolist(elems) + "]" ) & ( tn = "Map" => result =
             "dict({" + Expression.maptolist(elems) + "})" ) & ( tn = "
            Ref" & elems.size > 0 => result = "[" + elementType.
            defaultInitialValue() + "]*" + elems[1] );

        operation toPython() : String
        pre: true
        post: result = mapCollectionExpression(elements.toPython(),type
            .name);

        operation addReference(x : BasicExpression ) : Expression
        pre: true
        post: CollectionExpression->exists( e | e.expId = expId + "_" +
             x.data & e.isOrdered = isOrdered & e.umlKind = umlKind & e
            .elements = elements.addReference(x) & result = e );
    }

    abstract class Statement {
        stereotype abstract;

        attribute statId identity : String;

        static operation tab(indent : int ) : String
        pre: true
        post: ( indent <= 0 => result = "" ) & ( indent > 0 => result =
             " " + Statement.tab(indent - 1) );

        operation toPython(indent : int ) : String
        pre: true
        post: true;
    }

    class ReturnStatement extends Statement {

        reference returnValue[0-1] : Expression;

        operation toPython(indent : int ) : String
        pre: true
        post: ( returnValue.size = 0 => result = Statement.tab(indent)
            + "return" + "\n" ) & ( returnValue.size > 0 => result =
            Statement.tab(indent) + "return " + returnValue.any.
            toPython() + "\n" );
```

```
}

class AssertStatement extends Statement {

    reference condition : Expression;
    reference message[0-1] : Expression;

    operation toPython(indent : int ) : String
    pre: true
    post: ( message.size = 0 => result = Statement.tab(indent) + "
        assert " + condition.toPython() + "\n" ) & ( message.size >
         0 => result = Statement.tab(indent) + "assert " +
        condition.toPython() + ", " + message.any.toPython() + "\n"
         );
}

class ErrorStatement extends Statement {

    reference thrownObject : Expression;

    operation toPython(indent : int ) : String
    pre: true
    post: obs = thrownObject.objectRef & pars = thrownObject.
        parameters => result = Statement.tab(indent) + "raise " +
        thrownObject.mapErrorCall(obs.toPython(),pars.toPython()) +
         "\n";
}

class CatchStatement extends Statement {

    reference caughtObject : Expression;
    reference action : Statement;

    operation toPython(indent : int ) : String
    pre: true
    post: cleft = caughtObject.left & cright = caughtObject.right &
         result = Statement.tab(indent) + "except " + caughtObject.
        mapCatchExpression(cleft.toPython(),cright.toPython()) +
        ":\n" + action.toPython(indent + 2);
}

class FinalStatement extends Statement {

    reference body : Statement;

    operation toPython(indent : int ) : String
    pre: true
    post: result = Statement.tab(indent) + "finally :\n" + body.
        toPython(indent + 2);
}

class UseCase {

    attribute name : String;

    reference parameters[*] ordered : Property;
    reference resultType : Type;
    reference classifierBehaviour : Statement;
```

```
        operation mapUseCase() : String
        pre: true
        post: ( classifierBehaviour->oclIsUndefined() = false &
            resultType.name /= "void" => result = "def " + name + "(" +
             Expression.tolist(parameters.name) + ") :\n" +
            classifierBehaviour.toPython(2) + "  return result\n" ) & (
             classifierBehaviour->oclIsUndefined() = false & resultType
            .name = "void" => result = "def " + name + "(" + Expression
            .tolist(parameters.name) + ") :\n" + classifierBehaviour.
            toPython(2) + "\n" );
    }

    class BreakStatement extends Statement {

        operation toPython(indent : int ) : String
        pre: true
        post: result = Statement.tab(indent) + "break" + "\n";
    }

    class ContinueStatement {

        operation toPython(indent : int ) : String
        pre: true
        post: result = Statement.tab(indent) + "continue" + "\n";
    }

    class OperationCallStatement extends Statement {

        attribute assignsTo : String;

        reference callExp : Expression;

        operation toPython(indent : int ) : String
        pre: true
        post: ( callExp->oclIsUndefined() => result = Statement.tab(
            indent) + "pass\n" ) & ( true => result = Statement.tab(
            indent) + callExp.toPython() + "\n" );
    }

    class ImplicitCallStatement extends Statement {

        attribute assignsTo : String;

        reference callExp : Expression;

        operation toPython(indent : int ) : String
        pre: true
        post: result = Statement.tab(indent) + callExp.toPython() + "\n
            ";
    }

    abstract class LoopStatement extends Statement {
    stereotype abstract;

        reference test : Expression;
        reference body : Statement;
    }
```

```
class BoundedLoopStatement extends LoopStatement {

    reference loopRange : Expression;
    reference loopVar : Expression;

    operation toPython(indent : int ) : String
    pre: true
    post: result = Statement.tab(indent) + "for " + test.toPython()
        + " :\n" + body.toPython(indent + 2);
}

class UnboundedLoopStatement extends LoopStatement {

    operation toPython(indent : int ) : String
    pre: true
    post: result = Statement.tab(indent) + "while " + test.toPython
        () + " :\n" + body.toPython(indent + 2);
}

class AssignStatement extends Statement {

    reference type[0-1] : Type;
    reference left : Expression;
    reference right : Expression;

    operation toPython(indent : int ) : String
    pre: true
    post: result = Statement.tab(indent) + left.toPython() + " = "
        + right.toPython() + "\n";
}

class SequenceStatement extends Statement {

    attribute kind : int;

    reference statements[*] ordered : Statement;

    operation toPython(indent : int ) : String
    pre: true
    post: ( statements.size = 0 => result = Statement.tab(indent) +
        "pass\n" ) & ( statements.size > 0 => result = statements
        ->collect( s | s.toPython(indent) )->sum() );
}

class TryStatement extends Statement {

    reference catchClauses[*] ordered : Statement;
    reference body : Statement;
    reference endStatement[0-1] : Statement;

    operation toPython(indent : int ) : String
    pre: true
    post: ( endStatement.size = 0 => result = Statement.tab(indent)
        + "try :\n" + body.toPython(indent + 2) + catchClauses->
        collect( s | s.toPython(indent) )->sum() ) & ( endStatement
        .size() > 0 => result = Statement.tab(indent) + "try :\n" +
        body.toPython(indent + 2) + catchClauses->collect( s | s.
        toPython(indent) )->sum() + endStatement.any.toPython(
        indent) );
}

class ConditionalStatement extends Statement {

    reference test : Expression;
    reference ifPart : Statement;
    reference elsePart[0-1] : Statement;

    operation elsecode(indent : int ) : String
    pre: true
    post: ( elsePart.size = 0 => result = "" ) & ( elsePart.size >
       0 => result = Statement.tab(indent) + "else :\n" + elsePart
       .any.toPython(indent + 2) );

    operation toPython(indent : int ) : String
    pre: true
    post: result = Statement.tab(indent) + "if " + test.toPython()
       + " :\n" + ifPart.toPython(indent + 2) + elsecode(indent);
}

class CreationStatement extends Statement {

    attribute createsInstanceOf : String;
    attribute assignsTo : String;

    reference type : Type;
    reference elementType : Type;

    operation toPython(indent : int ) : String
    pre: true
    post: ( type : Entity => result = Statement.tab(indent) +
       assignsTo + " = None\n" ) & ( type->oclIsUndefined() =>
       result = Statement.tab(indent) + assignsTo + " = None\n" )
       & ( true => result = Statement.tab(indent) + assignsTo + "
       = " + type.defaultInitialValue() + "\n" );
}

abstract class BehaviouralFeature extends Feature {
stereotype abstract;

    attribute isStatic : boolean;

    reference parameters[*] ordered : Property;
    reference activity : Statement;
}

usecase printcode : void {

    ::
     true => "import ocl"->display() & "import math"->display() & "
        import re"->display() & "import copy"->display() & ""->
        display() & "from mathlib import *"->display() & "from
        oclfile import *"->display() & "from ocltype import *"->
        display() & "from ocldate import *"->display() & "from
        oclprocess import *"->display() & "from ocliterator import
         *"->display() & "from ocldatasource import *"->display()
```

```
            & "from enum import Enum"->display() & ""->display() &
            Entity.deleteOp()->display() & Entity.displayOps()->
            display();

        Enumeration::
         true => ( "class " + name + "(Enum) :" )->display() & literals
            ()->display() & "\n"->display();

        Entity::
         isInterface = true & isApplicationClass() => classHeader()->
            display() & staticAttributes()->display() &
            abstractClassConstructor()->display() & self.allOperations
            ()->forAll( op | op.displayOperation(2) );

        Entity::
         isInterface = false & isApplicationClass() => classHeader()->
            display() & staticAttributes()->display() &
            classConstructor()->display() & callOp()->display() & self
            .allOperations()->forAll( op | op.displayOperation(2) );

        Entity::
         isInterface = false & isApplicationClass() => createOp()->
            display() & allInstancesOp()->display() & ""->display() &
            createOclTypeOp()->display() & ""->display();

        Entity::
         isInterface = true & isApplicationClass() => allInstancesOp()
            ->display() & ""->display();

        Entity::
         isApplicationClass() & allProperties()->exists( k | k.isUnique
              ) & key = allProperties()->select(isUnique)->any() => key
            .getPKOp(name)->display() & key.getPKOps(name)->display()
            & self.createPKOp(key.name)->display();

        Operation::
         isStatic = true & owner->oclIsUndefined() => self.
            displayOperation(0) & ""->display();

        UseCase::
         true => mapUseCase()->display() & ""->display() & ""->display
            () & ""->display();
    }
}
```