Master Thesis

Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computers

A CRUD extension to a path based testing algorithm

Valeriia Chekanova

Supervisor: Ing. Karel Frajták, Ph.D.
Field of study: Software Engineering and Technology
May 2024

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Chekanova Valeriia**	Personal ID number:	**492197**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Computer Science**		
Study program:	**Open Informatics**		
Specialisation:	**Software Engineering**		

II. Master’s thesis details

Master’s thesis title in English:

A CRUD extension to a path based testing algorithm

Master’s thesis title in Czech:

CRUD rozšíření algoritmu pro path based testing

Guidelines:

Oxygen (http://still.felk.cvut.cz/oxygen/) is an open freeware tool for automated generation of path-based test cases for application processes and workflows. In Oxygen you can create a model of an application and use this model to generate process and workflow test cases.
However, the model does not reflect the data requirements. So, for example, you can generate a scenario where you need to display some data that was not created during the passage (e.g., switching to eShop checkout when the shopping cart is empty). At the same time, it is possible to define CRUD matrices in Oxygen. However, the two options are not linked and cannot be combined.
The aim of this work will be to design data structures and an algorithm to generate scenarios with defined constraints. Validate the algorithm on a suitably chosen application (or its part) and compare it with the output of Oxygen tool.
Design the algorithm independently of Oxygen.

Bibliography / sources:

BURES, Miroslav; RECHTBERGER, Vaclav. Dynamic data consistency tests using a crud matrix as an underlying model. In: Proceedings of the 2020 European Symposium on Software Engineering. 2020. p. 72-79.
BURES, Miroslav, et al. Testing the consistency of business data objects using extended static testing of CRUD matrices. Cluster Computing, 2019, 22: 963-976.
BURES, Miroslav; CERNY, Tomas. Static Testing Using Different Types of CRUD Matrices. In: Information Science and Applications 2017: ICISA 2017 8. Springer Singapore, 2017. p. 594-602.

Name and workplace of master’s thesis supervisor:

Ing. Karel Frajták, Ph.D. System Testing IntelLigent Lab FEE

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **05.02.2024** Deadline for master's thesis submission: ___________

Assignment valid until: **21.09.2025**

Ing. Karel Frajták, Ph.D.		prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work. The student must produce her thesis without the assistance of others, with the exception of provided consultations. Within the master’s thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student’s signature

Acknowledgements

I would like to acknowledge my supervisor, Ing. Karel Frajták, Ph.D., for his steady guidance and invaluable advice throughout this project. His support and expertise helped me to overcome several challenges and complete this thesis. His constant feedback provided me with opportunities to improve my work and ensure its quality.

I am grateful to my family for their belief in me. It was a constant source of motivation throughout my academic journey. Their faith in my abilities has a great impact on my achievements.

I would also like to thank my friends Alex and Liza. Their support, especially during the difficult days, gave me the strength to preserve and complete this thesis. Their friendship and encouragement have been invaluable, and I am fortunate to know them.

Declaration

I declare that this text presents my own work, and I have quoted all relevant sources of information used.

To improve text quality, I have used the cloud-based typing assistant Grammarly[1], which reviewed the grammar and spelling of my text.

Prague, 24. May 2024

Valeriia Chekanova

[1] `https://www.grammarly.com/`

Abstract

Software testing plays a crucial role in ensuring the quality and reliability of the system through the whole development cycle. Quality assurance of the application usually involves verifying its behaviour against predefined requirements through the creation and execution of test cases. However, as systems become more complex, traditional testing techniques face challenges in generating accurate test cases due to diverse user environments, complex system interactions, and performance optimization.

Model-based testing addresses these challenges by representing the System Under Test as a model that abstracts its expected behaviour. While several tools exist for test case generation using model-based testing, such as Oxygen[2] or GraphWalker[3], they do not allow us to incorporate data requirements in the model. It leads to appearing in test case step combinations that cannot happen in the real world, making the whole test case infeasible. Such test cases reduce the effectiveness of the testing process, resulting in wasted resources that could be allocated elsewhere.

Based on the concept of Negative Constrained Path-based Testing, this thesis introduces a novel algorithm that integrates constraints to prevent the generation of infeasible test cases. This approach enhances the existing methods by addressing data requirements in system models. The thesis details the development and analysis of this algorithm. Furthermore, it evaluates the results obtained using the developed algorithm against the ones obtained from Oxygen, focusing on metrics such as accuracy, optimality, and the number of infeasible test cases.

By addressing the limitations of existing testing techniques using constraints, this research aims to advance software testing practices by ensuring more reliable testing outcomes.

Keywords: system under test, model-based testing, automated testing, integration tests, path-based testing, data cycle test, integrity constraints, test scenarios, data integrity

Supervisor: Ing. Karel Frajták, Ph.D.

[2]`http://still.felk.cvut.cz/oxygen/`
[3]`https://graphwalker.github.io/`

Abstrakt

Testování softwaru hraje klíčovou roli při zajišťování kvality a spolehlivosti systému v průběhu celého vývojového cyklu. Zajištění kvality aplikace obvykle zahrnuje ověření jejího chování na základě předem definovaných požadavků prostřednictvím vytvoření a spuštění testovacích případů. Jak se však systémy stávají složitějšími, tradiční testovací techniky se potýkají s problémy při vytváření přesných testovacích případů kvůli různorodým uživatelským prostředím, složitým interakcím systému a optimalizaci výkonu.

Model-based testing tyto problémy řeší nahlížením na testovací systém jako na model, který abstrahuje jeho očekávané chování. Ačkoliv nástroje pro generování testovacích případů pomocí této techniky existují, například Oxygen[4] nebo GraphWalker[5], neumožňují nám do modelu zahrnout požadavky na data. To vede k tomu, že se v testovacím případě objevují kombinace kroků, které v reálném světě nemohou nastat. Následně se celý testovací případ stává neproveditelným, což vede ke snížení efektivity testování.

Tato práce vychází z konceptu Negative Constrained Path-based Testing a zavádí nový algoritmus, který integruje constrainty do modelu systému, aby se zabránilo generování nesplnitelných testovacích případů. Poté se výsledky algoritmu hodnotí ve srovnání s výsledky algoritmu Oxygenu, přičemž zaměřuje na metriky jako je optimalita a snížení počtu nesplnitelných testovacích případů.

Diplomová práce se zaměřuje na omezení existujících testovacích technik pomocí zavedení constraintů a klade si za cíl zlepšit stávající přístupy v testování softwaru zajištěním spolehlivějších výsledků testů.

[4]http://still.felk.cvut.cz/oxygen/
[5]https://graphwalker.github.io/

Klíčová slova: system under test, model-based testing, automatizované testování, integrační testy, path-based testing, data cycle test, integritní omezení, testovací scénáře, integrita dat

Překlad názvu: CRUD rozšíření algoritmu pro path based testing

Contents

Figures

Tables

Chapter 1

Introduction

Application testing is an important part of the whole cycle of software development, starting from gathering requirements and concluding with implementation and maintenance. Testing increases the chances that the software meets its requirements before reaching end-users, which in turn reduces the risk of customer dissatisfaction, errors, and potential harm to the company's reputation.

As software becomes more complex, testing becomes more challenging. One of the approaches to ensure the complex system's quality is Model-Based Testing (MBT). The idea is to represent a System Under Test (SUT) as an abstract model reflecting the desired system's behaviour. It allows abstracting from specific implementations and focusing on the logic and interactions between components. Automatic test case generation from such models is generally more efficient than manual, unorganized methods. [1]

Nevertheless, the effectiveness of the testing process relies on the accuracy and completeness of the models used for test generation. Making too many simplifications while modelling, having differences in abstraction level, or changes in the system behaviour over time may introduce inconsistencies between the system and testing model. In turn, it will result in less precise SUT representation, leading to the generation of test cases that inaccurately reflect system behaviour and characteristics. Not only is creating an error-free model often difficult, but creating a complete model can also be challenging due to resource limitations. Nevertheless, even a well-designed model would not be able to guarantee a lack of defects in the SUT. Still, refining the models enhances their accuracy, thereby reducing discrepancies and improving the validity of generated test cases. [1]

The SUT model allows us to design various tests, including unit tests, integration tests, and End–to–End (E2E) tests. However, the thesis focuses on E2E tests as they are essential in identifying defects throughout the application flow by mimicking user interactions. The accuracy of E2E testing heavily depends on the quality of the test data used. In MBT, preparing precise test data that reflect real-world usage can be challenging due to the model's abstraction. As a result, test data and scenarios derived from incomplete models may not accurately represent the expected system behaviour and could even be infeasible as they may never occur in the real world. [1]

1.1 Motivation

Consider the following SUT: an e-commerce platform that allows users to log in, log out, add and remove products from the cart, and go to checkout. For this model, a tool for automatic test case generation may create the following test case: a user logs into the e-shop and immediately navigates to the checkout without adding any items to the shopping cart. This scenario is infeasible because the checkout process requires items in the cart to proceed.

Another example within the same SUT may be a test case where a user logs in and attempts to remove a non-existent item from the cart. Evaluating this test case would incorrectly indicate a failure, as the absent item cannot be removed even though the login and item removal functionalities may be working correctly.

In complex systems, identifying such infeasible test cases may not be straightforward. Testers might spend considerable time investigating these errors and searching for bugs in correctly implemented features. Not only does this waste the tester's time, but it also diverts resources from important issues.

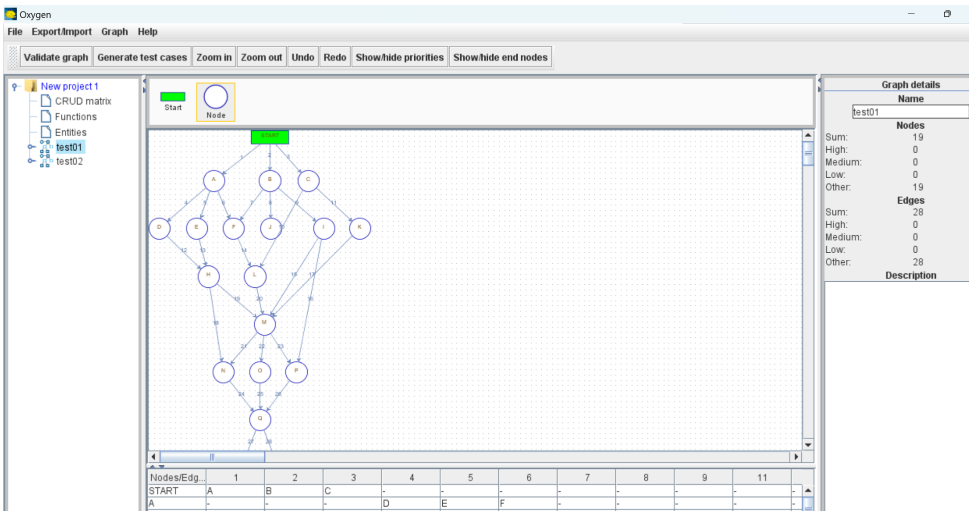
This thesis aims to develop strategies that mitigate test failures caused by incorrect test case generation despite correct feature implementation. By refining the existing testing approaches, this work attempts to mitigate the generation of infeasible test cases, thereby improving the efficiency of software testing.

1.2 Task Definition

There exist several applications for automatically generating test cases, for example, GraphWalker[1]or Oxygen[2] developed by the System Testing IntelLigent Lab [3] at Czech Technical University. In general, MBT allows the representation of SUT in different ways, such as Finite State Machines, UML State Machines, or Pre/Post Models. Oxygen allows the representation of the SUT as a UML activity diagram or as a directed graph. Based on this model, Oxygen automatically generates test cases (testing paths) with specified test coverage (see screenshot 1.1). Despite its advanced capabilities, the existing applications have a significant limitation: they cannot reflect SUT's data requirements. A tester can create a graph representing the user's actions within the SUT, but they cannot define specific data requirements for action execution. For instance, Oxygen might create a test path that requires displaying data that does not yet exist within the test.

Therefore, the task is to develop an algorithm to generate test cases that consider data requirements. The applied constraints will make generated test cases more realistic, thus making the testing process more effective.

[1]`https://graphwalker.github.io/`
[2]`http://still.felk.cvut.cz/oxygen/`
[3]`http://still.felk.cvut.cz/index.html`


Oxygen
File Export/Import Graph Help
Validate graph Generate test cases Zoom in Zoom out Undo Redo Show/hide priorities Show/hide end nodes
New project 1
CRUD matrix
Functions
Entities
test01
test02
Start
Node
Graph details

Name: test01

Nodes
Sum: 19
High: 0
Medium: 0
Low: 0
Other: 19

Edges
Sum: 28
High: 0
Medium: 0
Low: 0
Other: 28

Description

Nodes/Edg...	1	2	3	4	5	6	7	8	9	11	
START	A	B	C	-	-	-	-	-	-	-	-
A	-	-	-	D	E	F	-	-	-	-	-

Figure 1.1: Screenshot of Oxygen GUI

1.3 Objectives

The primary objectives of this thesis are to:

- Understand the factors that lead to the generation of infeasible test cases.
- Design and implement an algorithm that ensures the test cases accurately reflect data requirements.
- Evaluate and compare the results obtained by the implemented algorithm with those from Oxygen. Validate tests generated by the proposed algorithm and Oxygen on a real application as a part of the evaluation.
- Discuss the limitations of the implemented solution and propose future development directions.

1.4 Thesis Structure

The theoretical chapters describe core software testing techniques with a primary focus on test case design. This helps to understand the problem of automatically generated infeasible tests, which occur primarily due to unmet data requirements. The "Related work" chapter provides an overview of research in the area of generating feasible test cases and assesses its suitability for the thesis needs. Subsequently, the "Analysis" chapter details the proposed application design, emphasizing the reasons behind the chosen testing technique, design patterns, and algorithms. The "Development" chapter then describes a solution that employs constraints in test case generation. The algorithm implementation, analysis and testing are described in detail. Finally, the evaluation of the work is presented, including details of achieved objectives, a comparison with Oxygen's results, and future directions for work.

Part I

Theoretical part

Chapter 2

Essential Theory

This chapter contains fundamental theory describing challenges in test case preparation.

The discussion begins with exploring the System Under Test (SUT), highlighting the potential challenges that may arise during modelling and consecutive evaluation of different systems. This helps to understand the impact of SUT's characteristics on testing strategies.

Further, the chapter gives an introduction to Model–Based Testing, describing its representational methods, which form the basis for the subsequent introduction of Path-Based Testing and State Transition Testing. This section highlights the importance of selecting appropriate modelling techniques for effective SUT evaluation.

Additionally, the chapter describes the concept of test cases with their characteristics and examples. This includes examining how test cases are designed and used to validate the functionality of SUT.

Concluding the chapter, a definition of Test Coverage is provided. This part discusses important coverage criteria for reliable and effective testing.

2.1 System Under Test

The System Under Test (SUT) is a system that is being evaluated during testing. Understanding the SUT is necessary for effective test planning and execution, as the complexity of the SUT significantly impacts the choice of testing strategy. [2]

2.1.1 Characteristics of the SUT

The SUT has several characteristics that may influence the testing strategy [1]:

- **Complexity** of the SUT significantly affects the choice of a testing strategy. For example, a complex SUT may require a multilayered testing approach, which involves tests validating the system's behaviour from an external perspective and exploring the internal mechanisms (so-called functional and structural techniques). [1]

- **Dependency** refers to the dependency of the SUT on external systems or APIs that are not part of the main system but are important for its operation. The presence of a dependency on other systems may require verification of the communication between them. [1]
- **Stability** of the SUT is crucial in determining the frequency and intensity of testing phases. Stable systems might require less frequent but thorough testing cycles to verify that existing functionalities perform correctly after modifications. [1]
- **Scalability** of the SUT determines how well the system can handle increased loads without dropping performance. This characteristic influences resource allocation for stress testing and selecting an environment. [1]

2.1.2 Importance of Understanding the SUT

A proper understanding of the SUT allows us to create more effective test strategies, which can reduce wasted resources by focusing on the most critical areas. It has the following objectives:

- **Identifying Critical Test Areas** helps to prioritize different system parts and ensure that the most important parts of the system are still covered even with a limited amount of resources. [3]
- **Effective Test Coverage** metrics can be designed only with a proper understanding of the SUT's functionality priorities. For example, a higher Test Coverage should be chosen for higher-priority methods. [3]
- **Predicting Potential Failures** would be impossible without knowledge of the most error-prone parts of SUT. [2]

Understanding the SUT affects every decision in the testing lifecycle, from selecting the appropriate testing strategy to assessing the final system quality. Thus, a thorough analysis of the SUT is important before initiating the testing process. [3]

2.2 Testing Techniques

Software application testing can be executed using several methods. The suitability of the methods is dependent on the SUT characteristics. [4]

2.2.1 Manual and Automated Testing

The SUT can be tested manually or automatically. The choice of approach depends on the SUT's features and budget, timeline, and testing objectives. [4]

- **Manual Testing** is conducted by human testers and is beneficial due to its flexibility. It allows one to identify issues connected with user experience quickly. Among its disadvantages are labour-intensiveness and proneness to human error. This makes it more suitable for exploratory testing in smaller projects or those with rapidly changing requirements. [4]

- **Automated Testing** utilizes tools and scripts to execute test cases, facilitating rapid test execution and repeatability. This method is optimal for regression testing and large projects where tests must be run repeatedly. Although the initial setup and maintenance of automated tests require some effort, the benefits, such as increased test coverage and precision, often justify the initial setup costs. [4]

2.2.2 Black-box, White-box, and Grey-box Testing

Software quality can be assessed from internal or external perspectives, with Black-box testing evaluating the software externally, while White-box testing focuses on internal system workings. Grey-box testing combines the mentioned techniques, using partial knowledge of internal structures while maintaining an external focus on functionality. [5]

Table 2.1 compares these techniques in terms of their advantages, disadvantages, and suitability to systems.

2.2.3 Testing Pyramid

The Testing Pyramid is a model that helps to structure a testing strategy and focus tests at different application levels. Depending on granularity objectives, the tests may be unit, integration, and end-to-end. The pyramid approach highlights a typical distribution of test types: having more low-level unit tests over fewer high-level end-to-end tests. [8]

- **Unit Testing** level forms the pyramid's base and involves testing individual pieces of code in isolation. Unit tests ensure that each component functions correctly before integrating with others. Unit tests are fast, reliable, and inexpensive to automate, thus facilitating regression mitigation during later development stages. [8]

- **Integration Testing** focuses on the interactions between components. It verifies that different parts of the application work together as expected. This type of testing is important for identifying issues that occur when separately tested units are combined into a full system. [8]

- **End-to-End Testing** is at the top of the pyramid. It involves testing the entire application for dependencies, data integrity, and communication with other systems, interfaces, or databases to ensure it works in a real scenario. This type of testing is usually limited due to its complexity and the required resources. [8]

Testing Type	Definition	Advantages	Disadvantages
Black-box Testing	Tests the functionality of an application against its specifications without internal knowledge [5].	Focuses on user experience; does not require knowledge of the codebase [6].	May miss structural defects since it does not examine the code internally [5].
White-box Testing	Involves detailed testing of the internal logic and code structure [2].	Provides thorough coverage, identifying hidden errors within the code [2].	Requires detailed programming skills and may not assess the user's perspective [2].
Grey-box Testing	Combines black-box and white-box testing with some knowledge of the internal data structures and algorithms but not the full details [7].	Balances between user perspective and code perspective [7].	Less detailed than white-box testing in code coverage may require higher skills than black-box testing [7].

Table 2.1: Comparison of Black-box, White-box, and Grey-box Testing Techniques

Table 2.1 presents a Testing Pyramid diagram illustrating the different layers of testing, from unit tests at the base to E2E tests at the top.

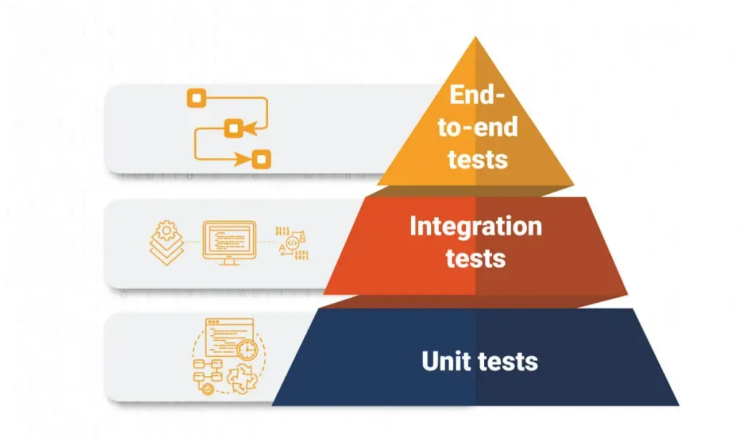


Figure 2.1: Testing Pyramid diagram, source: [9]

2.3 Model-Based Testing

Model-Based Testing (MBT) is an approach to testing SUT by describing its expected behaviour and structure through a model created according to project requirements. Preparing a SUT model during MBT can identify potential errors in the requirements and facilitate early bug detection. [10, 1]

The SUT can be presented in several ways:

- **State Machines** are suitable for systems with distinct state transitions as they offer precise modelling of state behaviour. [11]
- **Petri Nets** are effective for complex, concurrent processes. The concept is adopted from systems with asynchronous behaviours. [12]
- **UML Activity Diagrams** are dynamic models that visually represent the workflow of a system, illustrating the sequence of activities and the conditions controlling the flow. Flowcharts serve a similar purpose and are considered predecessors to Activity Diagrams. [13]
- **Entity-Relationship Diagrams** are primarily used in database systems to visually represent data models, mapping out entities, their attributes, and their relationships. This representation is most suitable for testing database integrity and operations. [14]

Each modelling technique offers unique advantages covering different SUT aspects [1].

2.3.1 Directed Graphs: Unifying MBT Models

Directed graphs offer a visual and analytical way to represent the order of actions that may be executed within the SUT or to model existing dependencies within the SUT. Thus, it serves as a unifying structure that can present different MBT approaches. [15]

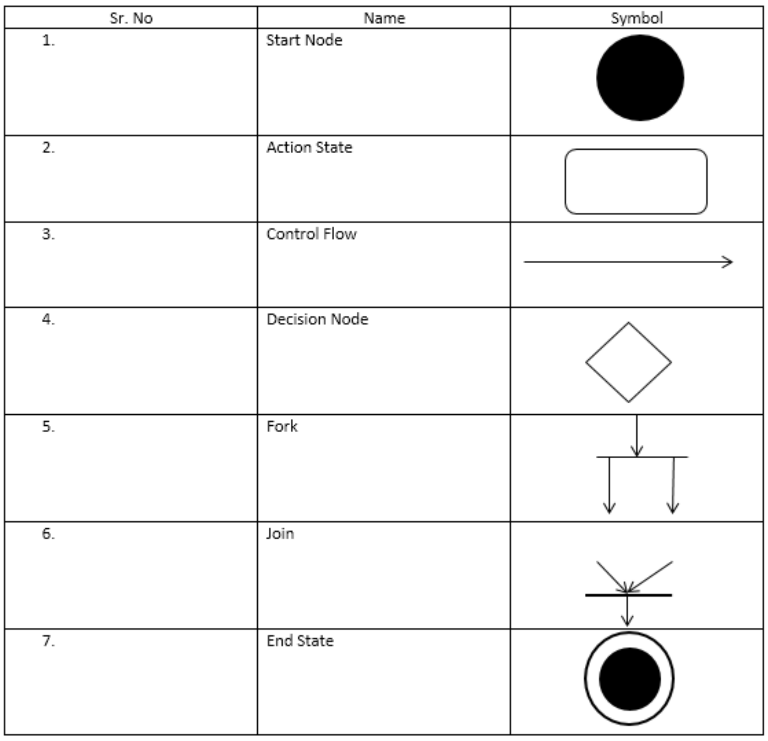
Within MBT, SUT may be represented as a directed graph $G = (N, E)$, where N is a set of nodes, and E is a set of edges between these nodes. A subset $N_e \subseteq N$ contains the end nodes of the graph, and likewise, a subset $N_s \subseteq N$ contains the starting nodes of the graph, where $N_e \neq \emptyset$ and $N_s \neq \emptyset$. A test case t is a sequence of nodes $n_1, n_2, \ldots, n_m$, with a corresponding sequence of edges $e_1, e_2, \ldots, e_{m-1}$, where each edge $e_i = (n_i, n_{i+1}) \in E$, $n_i \in N$, and $n_{i+1} \in N$. The test case t begins with a start node $n_1 \in N_s$ and ends with an end node $n_m \in N_e$. The test case t can be denoted either as a sequence of nodes $n_1, n_2, \ldots, n_m$ or as a sequence of edges $e_1, e_2, \ldots, e_{m-1}$. The test set T is a collection of such test cases. [15]

A state machine may be projected on a graph where states are nodes and transitions are edges. This allows us to use graph algorithms for further analysis and testing. [15]

The stakeholders are not always able to provide a directed graph representation of the SUT. Therefore, understanding how the activity diagram may be transferred to a directed graph is worthwhile. Figure 2.2 presents the notation of the UML Activity Diagram. Transforming a UML activity diagram into a directed graph involves mapping its components to nodes and control flows to edges. Based on the presented representation of SUT, the conversion might look in the following way:

- The **Start Node** is the initial node with only outgoing edges.
- **Action States** are converted into nodes connected by directed edges representing the **Control Flow**.
- **Decision Nodes** and **Fork** constructs incoming nodes with multiple outgoing edges for conditional paths.
- The **End Node** is the terminal node with no outgoing edges. It marks the process as completed.

Following these steps, each element and interaction within the activity diagram is represented using a directed graph, facilitating analysis with graph theory.



Sr. No	Name	Symbol
1.	Start Node	●
2.	Action State	▭
3.	Control Flow	→
4.	Decision Node	◇
5.	Fork	
6.	Join	
7.	End State	◉

Figure 2.2: UML Activity Diagram Notation, source: [16]

2.4 Path-Based Testing

Path-Based Testing is an approach to creating test cases based on a directed graph representation of the SUT. [15]

2.4.1 Test Case

A Test Case is a detailed description of a test that contains the steps, conditions, and expected outcomes necessary to validate the functionality of the SUT. [4] The table 2.2 contains a test case for verifying the login functionality of an application.

Test Case ID	Actions	Input Data	Expected Result
TS01	1. Access the login page. 2. Enter username. 3. Enter password. 4. Click the 'Login' button.	**login**: user1, **password**: pass123	Dashboard is displayed.

Table 2.2: Test Case for Login Functionality

Test cases help us to systematically verify system behaviour, find defects, and validate the system against specified requirements. Additionally, test cases may serve as an effective communication tool between the testing team and stakeholders. [4, 1]

Although it is possible to generate test cases manually, this solution is not feasible for extensive or error-sensitive applications (e.g. aviation software) as it may lack precision and repeatability. [1]

2.4.2 Test Coverage Criteria

Test coverage criteria are quantitative measures used to determine the extent to which the SUT has been tested. It may serve as a metric to assess the completeness of the test cases in uncovering potential defects. Higher coverage increases the chance of detecting errors, thereby improving the reliability and quality of the software application. Nevertheless, 100 % coverage does not mean the SUT has no errors. [17]

Coverage criteria are important for several reasons:

- **Identifying untested parts of a program**. Coverage criteria help identify system areas not covered by tests. It may serve as a guide for developers to extend the test base. [8]

- **Reducing risks**. By aiming for higher coverage, developers can reduce the risk of regression by ensuring that code changes do not introduce errors in existing code. [8]
- **Enhancing software maintenance**. The project with a higher test coverage is safer to refactor, making the application more maintainable over time. [8]

There are several test coverage criteria precisely described in [15]:

- **Edge Coverage** requires each edge $e \in E$ to be present in the test set T at least once.
- **All Node Coverage** requires each node $n \in N$ to be present in the test set T at least once.
- **Edge-Pair Coverage** criterion requires T to contain each possible pair of adjacent edges in G.
- **Prime Path** requires each reachable prime path in G to be a sub-path of the test case $t \in T$. A path p from e_1 to e_2 is prime if (1) p is simple, and (2) p is not a sub-path of any other simple path in G. A path p is simple when no node $n \in N$ is present more than once in p (i.e., p does not contain any loops); the only exceptions are e_1 and e_2, which can be identical (p itself can be a loop).

All Node and All Edge Coverage may be used for less intensive testing, which is typically enough for low-risk applications. In contrast, Edge-Pair Coverage and Prime Path Coverage are suitable for critical applications where the risk of failure has significant consequences. [18, 15]

2.5 State Transition Testing

State Transition Testing (STT) is a technique used to verify a SUT's behaviour in reaction to events. The idea is to model SUT as a state machine consisting of many states, transitions between them, and events that trigger them. Testing includes checks for every transition and for the system handling invalid transitions. It helps ensure the correctness of processing state changes. The difference from other testing methods is that the response of the SUT depends on its state, so it is not required that they always correspond 1:1 with the process diagram. STT is mostly useful for systems where events and their historical states influence responses, for example [19]:

- **Embedded Systems**: Based on the sequence of the events, special behaviour might be required by these systems. [19]
- **Interactive Applications**: An example is user interfaces, where the available actions may differ depending on the current state (e.g., options change depending on whether a user is logged in). [19]

State Transition Coverage Criteria measures the adequacy of testing state transitions in a SUT. Although there are similarities with the coverage types used in Path-Based Testing, it is important to define these criteria to prevent ambiguity. The choice of State Transition Coverage Criterion depends on the available resources and the criticality of the SUT. [19]

- **0-switch Coverage**. This type is also referred to as state coverage, which requires each state to be visited at least once. This basic coverage level ensures that every state is reachable and can handle the minimum valid input. This coverage should be used for non-critical applications only. [19]
- **1-switch Coverage**. This level extends 0-switch coverage by ensuring that all direct transitions between states are tested at least once. It verifies the system's behaviour for every possible single-event transition between states. [19]
- **N-switch Coverage**. In N-switch coverage, sequences of N transitions are tested. This type of coverage is more thorough as it checks the system's behaviour over sequences of events, not just single transitions. The most popular is the 2-switch coverage, which tests the transition between states due to a sequence of two events. This type is preferable for critical applications because it provides a high probability of defect detection even at the price of higher resource demands. [19]
- **All-paths Coverage**. All-paths coverage is the strongest criterion, requiring testing every possible path through the state machine. [19]

Chapter 3

Data Handling in Testing

This chapter outlines existing approaches to the management of data in software testing. It begins with the introduction of advanced testing techniques such as Data Flow Analysis (DFA) and Data Cycle Test (DCyT). Further, the chapter addresses the challenge of infeasible test cases by outlining strategies to identify and exclude them. Finally, the Object Constraint Language (OCL) will be described as a tool to present the constraints within the UML models.

Understanding the described methodologies is crucial for selecting the proper testing technique and successful algorithm development.

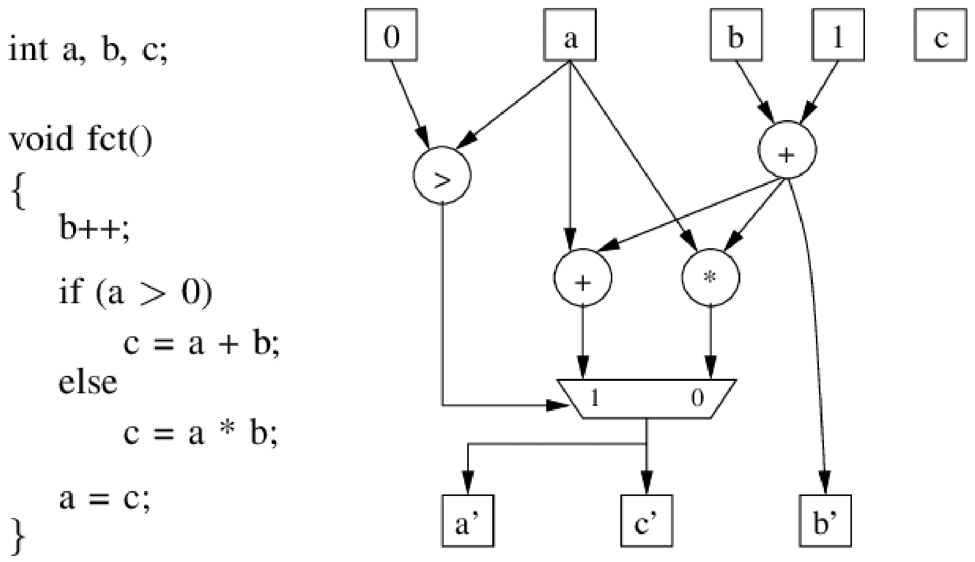
3.1 Data Flow Analysis

Data Flow Analysis (DFA) is a white-box technique for identifying abnormalities in data usage. The main principle is analysing identified critical points in its lifecycle. The critical points are where objects are defined, processed, and destroyed. It may work on the code level as well as on the object level. [20]

The analysis involves identifying several key points in the data flow [20]:

- **Definition points** are the places where the objects are instantiated, or the values are assigned, specifying the starting points of the data paths.
- **Modification points** are places where data is modified.
- **Reference points** are locations where a variable or object is accessed without modification.
- **Evaluation points** refer to a place that computes one or more objects.

Using these points, we can construct data flow graphs that map out each object's flow through the path execution. The graph example is shown in figure 3.1. Graph analysis helps identify anomalies such as unused, undefined, or misused objects. [20]



```
int a, b, c;

void fct()
{
    b++;
    if (a > 0)
        c = a + b;
    else
        c = a * b;
    a = c;
}
```

Figure 3.1: Data-flow graph, source: [20]

DFA can be used in the compiler's design to optimize the code and ensure it is correct. Precisely, it works in dead code elimination, strength reduction, and register allocation. By analyzing the data flow through compilation stages, DFA facilitates the detection of redundant instructions and improves the efficiency of generated machine code. [20]

To conclude, this approach helps to identify where data is defined and used. It can be utilized for ensuring that there is no attempt in generated test cases to use data before its initialization or once it is destroyed. [20]

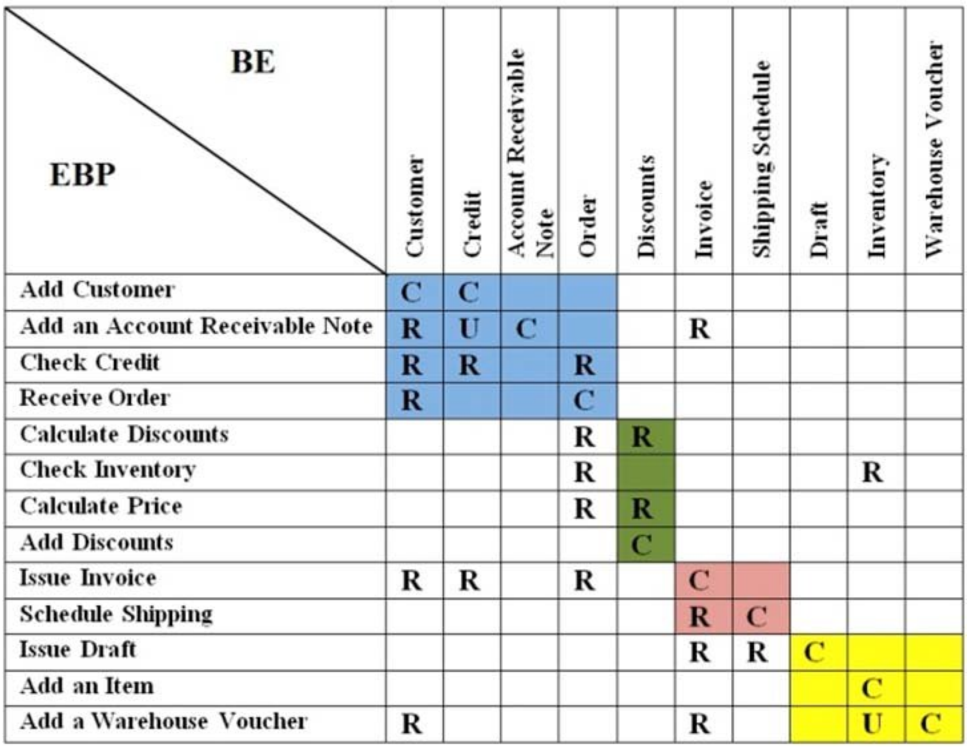
3.2 Data Cycle Test

Data Cycle Test (DCyT) is a testing technique that validates data integrity and corrects data processing concerning the SUT. [15]

In contrast to DFA, DCyT operates abstractly on the general data entities instead of operating on critical points. Furthermore, DCyT integrates a CRUD matrix to check data operations. [15]

Further, DCyT may improve coverage criteria for STT (refer to Section 2.5), where the Update operation of the CRUD matrix usually corresponds to state transitions. However, it must be noted that DCyT is not one-to-one with STT. [15]

The principle behind DCyT is mapping the interactions between system functions and data entities to identify operations allowances for different entities [15]. The example of the CRUD matrix is presented on figure 3.2.



EBP \ BE	Customer	Credit	Account Receivable Note	Order	Discounts	Invoice	Shipping Schedule	Draft	Inventory	Warehouse Voucher
Add Customer	C	C								
Add an Account Receivable Note	R	U	C			R				
Check Credit	R	R		R						
Receive Order	R			C						
Calculate Discounts				R	R					
Check Inventory				R					R	
Calculate Price				R	R					
Add Discounts					C					
Issue Invoice	R	R		R		C				
Schedule Shipping						R	C			
Issue Draft						R	R	C		
Add an Item									C	
Add a Warehouse Voucher	R					R			U	C

Figure 3.2: An example of CRUD matrix, source:[21]

The structure of a CRUD matrix in general is defined as follows [15]:

Let $F = \{f_1, \ldots, f_n\}$ represent the set of all functions within the System Under Test (SUT), and $E = \{e_1, \ldots, e_p\}$ denote the set of all data entities considered for the test design. The CRUD matrix $\mathbf{M}$ is then represented by $\mathbf{M} = (m_{i,j})_{n,p}$, where $n = |F|$ and $p = |E|$. Each element $m_{i,j}$ in the matrix indicates the operations performed by function $f_i \in F$ on entity $e_j \in E$, such that $m_{i,j} = \{o \mid o \in \{C, R, U, D\}\}$ if and only if function f_i is authorized to execute the operation o on entity e_j.

According to the selected methodology, DCyT may apply various rules to verify data integrity [15]:

- It builds scenarios with the SUT present functions Create (C), Read (R), Update (U), and Delete (D) operations.
- The testing scenario should be in the form: Create (C), then either a combination of Read (R) and Update (U), and ending with Delete (D).
- There should be a Read (R) after each Create (C), Update (U), and Delete (D) to keep the consistency.
- If capacity is limited, not all functions can be covered. This may mean the CRUD operations must be prioritized according to the function's importance.

In the context of the thesis, the CRUD matrix incorporated by DCyT may improve accuracy of generated test cases. This may be achieved by ensuring that abstract graph models used for test generation accurately represent the real-world operations on entities. As a result, test engineers may more effectively translate data requirements presented by the CRUD matrices into the constraints form.

3.3 Infeasible Test Cases

Infeasible test cases describe a certain state that cannot happen in the real world due to violating data requirements or action orders. [22]

To better understand the problem of infeasible test cases, consider the following examples:

1. **Creating the order with an empty cart** A user logs in to the e-shop and goes to checkout without having added any items previously to the cart. The scenario is invalid: creating an order with an empty cart is meaningless.

2. **Removal of the item without prior addition** A user tries to remove an item from their shopping cart that was never added. Removing an item that does not exist in the cart is impossible.

Identifying and excluding such infeasible test cases are important to keep the testing process flawless. It can be done in the following ways:

- **Validation of Preconditions** is a process that checks that all conditions are met before a test step can be executed. This includes checking state prerequisites and data availability. [22]

- **Model Refinement** means updating the model to reflect infeasible states and transitions in the application. [22]

- **Use of Constraint Solvers** refers to the application of automated tools that use constraints to determine whether a specific test case is feasible. They help check if the conditions of a test case can be met. [22]

3.4 Constraints in UML Diagrams

Constraints allow us to describe how the operations change the state of the SUT. When used properly, the constraint prevents the system from producing undesired behaviour and ensures that it adheres to business and logical rules. Thus, it simplifies SUT maintenance for developers, testers, and other stakeholders and increases system reliability. [23]

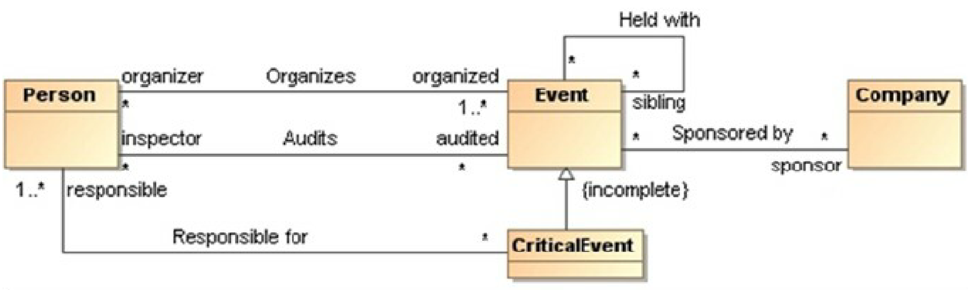
Constraints are usually applied to activity diagrams by annotating transitions with conditions that must be met for the transition to occur. These conditions are based on the values of attributes or the occurrence of specific events. It ensures that transitions occur in appropriate contexts. [23]

In the thesis context, constraints may be defined on the SUT to prevent particular edge combinations from appearing in the testing paths.

The constraints may be presented in several ways, namely:

- **Natural Language**. Natural language constraints are written in plain language, making them understandable for stakeholders who usually are not familiar with formal notation. This approach is beneficial during the early design and requirement analysis stages, though it may lead to ambiguities later. [24]
- **Pseudocode**. Pseudocode provides a balance between formal and informal languages. It is useful for bridging the design and implementation gap in a structured format. [25]
- **Graphical Notations**. Graphical notations use visual symbols or extensions to UML for constraints representation. This method effectively expresses simple conditions but may not be optimal for more complex ones. [26]
- **Object Constraint Language (OCL)**. OCL is one of the most precise methods for defining constraints. It is used for detailed, unambiguous constraint specification, ensuring precision in defining system requirements and behaviours. However, its effectiveness depends on the author of the constraints and its accuracy. Moreover, It may not be optimal for simple systems, as the initial time invested in learning this complex notation (essentially a new programming language) may not be worth it. [27]

Figure 3.3 presents an example of a UML diagram with OCL constraints.



```
Integrity constraints

  1. context Person inv: organized->exists(oclIsTypeOf(CriticalEvent) and
       sponsor->isEmpty())

  2. context Event inv: sibling->notEmpty() implies not oclIsTypeOf(CriticalEvent)

  3. context Person inv: audited->forall(sponsor->notEmpty())

  4. context CriticalEvent inv: not responsible.organized-> isEmpty()

  5. context Company inv: event->select(oclIsTypeOf(CriticalEvent))->notEmpty() or
     event->exists(sibling->isEmpty() or sibling->select(sponsor->isEmpty())->
     notEmpty())
```

Figure 3.3: A UML diagram with OCL constraints, source: [28]

3.4.1 OCL

The Object Constraint Language (OCL) is a formal language developed by IBM that enhances UML. Its purpose is to precisely specify conditions and rules that cannot be expressed graphically on UML models. Constraints can present business rules and logical conditions to ensure the system functions correctly. [27, 29] The OCL has several advantages [29]:

- **Precision**. OCL offers high precision for the specification of complex rules, which is important for validating the behaviour of especially complex SUTs.
- **No Side Effects**. OCL expressions do not affect the state of the SUT but assert truths about the model, thus ensuring design consistency and correctness.
- **Formal Syntax and Semantics**. OCL has a well-defined syntax and semantics that avoid ambiguity.

Syntax Overview

Programming languages heavily inspire OCL's syntax. The structure of an OCL expression is based on the object-oriented approach, where expressions are formed concerning objects and their properties. It is important to note that OCL is a declarative language specifying which conditions must be held without defining how they are maintained. [29]

The **context** specifies the element within a UML model to which a constraint applies. Typically, it is a class, an attribute, or an operation. [29] For example:

```
context Person
```

OCL introduces basic types such as **Integer**, **Real**, **Boolean**, and **String** and collection types like **Set**, **Bag**, **Sequence**, and **OrderedSet**. Collections can be manipulated through operations like

```
    select, reject, collect, forAll, exists
```

facilitating complex data manipulation directly in the model's constraints [29].

Common Expressions

OCL has several elements representations [29]:

- **Invariants** must always hold for class instances. For example, in a banking application, an invariant might ensure that the balance of any account must never be negative:

```
context Account inv: self.balance >= 0
```

- **Preconditions** must be met before operation execution. They ensure the system is in the correct state before the operation starts. For instance, before a withdrawal operation, one might check that the amount to be withdrawn is less than the current balance:

  ```
  context Account::withdraw(amount : Real)
  pre: amount > 0 and self.balance >= amount
  ```

- **Post-conditions** specify the system's state after an operation. For example, after a deposit operation, the balance should be increased by the deposited amount:

  ```
  context Account::deposit(amount : Real)
  post: self.balance = self.balance@pre + amount
  ```

- **Guard Conditions** control the operations flow. A guard condition specifies that a state transition can only occur if a certain condition is met, e.g., transitioning from a ‘Checking‘ to an ‘Approved‘ state is allowed only if a document's status is ’Valid’. Although it is not a part of OCL itself, it is often used in state and activity UML diagrams. [30] An example of the mentioned situation described using OCL is the following:

  ```
  context Document::approve()
  pre: self.status = 'Valid'
  ```

OCL language may be used as an extension of the proposed algorithm to formally define constraints that prevent the generation of infeasible test paths. Employing OCL provides a precise way to express conditions, leading to a better definition of constraints closer to real application requirements.

Part II

Analysis

Chapter 4

Related work

This chapter reviews existing academic papers that focus on problems related to data treatment in automatic test case generation.

4.1 Negative Constrained Path–based Testing

Authors of [31] suggest extending traditional path–based testing by introducing negative constraints to prevent the execution of particular system parts.

The introduced constraints are :

- **Type EXCLUSIVE:** An action A must NOT be followed by an action B in a test case in a test.
- **Type ONLY ONCE:** An action A can be followed by an action B in a test case in a test set only once.

Although Negative Constrained Path–based Testing (NCPT) may look similar to the DFT, its usage is not limited to positive constraints and provides thorough test coverage. It helps to ensure that testing is possible under various system states and that asynchronous operations are efficiently done. Moreover, it allows us to mimic outages in the test environment.

The paper outlines the theoretical foundation of the idea but lacks details on its implementation. Additionally, while the paper introduces an approach that may help solve the situation, the introduced constraints are insufficient to address the thesis motivation.

4.2 Condition–Classification Tree Method

Condition–Classification Tree Method (CCTM), an enhancement in test case generation from UML activity diagrams, is introduced in [32]. This method transforms decision points and guard conditions from activity diagrams into a structured form called condition classification trees. It aims to identify defects while optimising testing resources. 4.1 illustrates the main steps of the methods.

The CCTM algorithm works in three main steps:

1. **Generation of Condition–Classification Trees:** Each decision point in the UML activity diagram and its associated guard conditions are transformed into a condition–classification tree. This tree reflects the branching logic of the activity diagram.
2. **Creation of Test Case Tables:** Once the trees are created, they are used to construct test case tables. The table is structured to reflect the hierarchical and conditional relationships defined in the condition–classification trees.
3. **Test Case Generation:** Finally, test cases are generated from the test case table by selecting combinations of conditions that represent valid paths through the software's workflow.

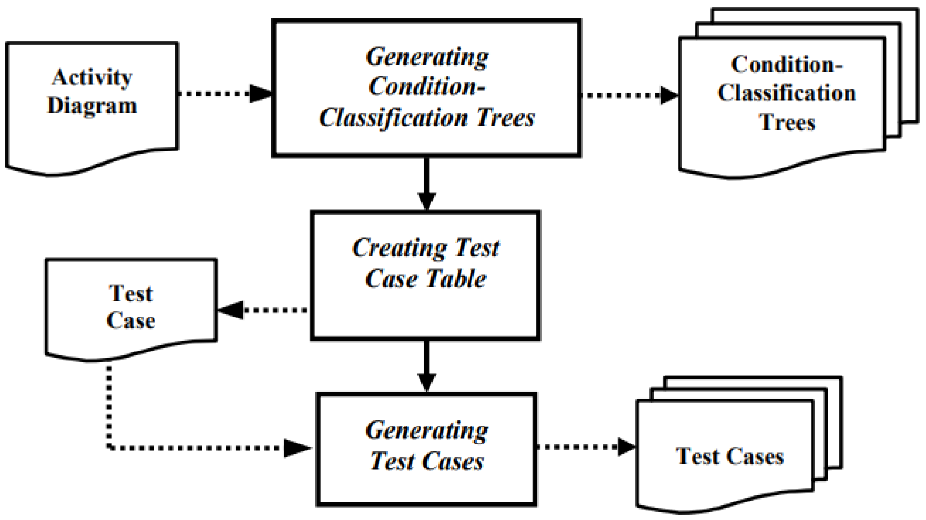


Figure 4.1: Condition–Classification Tree Method, source: [32]

Nevertheless, CCTM cannot prevent the occurrence of specific action combinations in the resulting test scenarios.

4.3 Heuristics–based infeasible path detection

Authors of [33] describe a method of identifying infeasible paths using heuristics.

In this approach, the program is initially executed with arbitrary inputs. It allows the generation of execution traces, analyzed using predefined empirical properties. It helps to identify common patterns presented in infeasible paths. An example of such a pattern could be a consistent failure to meet certain conditions even with varying inputs.

The method's main idea is to predict path infeasibility by applying heuristic rules based on the observed patterns of infeasibility. If a path is marked as infeasible, the test generation for this path is terminated. Then the inputs are adjusted accordingly. The implementation is developed in Java and uses evolutionary algorithms.

However, while this method improves test data quality, it does not solve the problem of mitigating infeasible test path generation.

4.4 Detecting Interprocedural Infeasible Paths

Authors of [34] propose to detect infeasible paths by integrating interprocedural dataflow analysis with symbolic propagation mapping (ISPM). It allows modelling the value–passing process at each call site across generated whole program paths (WPPs). This technique aims to improve the consistency and accuracy of static testing by finding discrepancies early.

The ISPM facilitates examining the connections between formal and actual parameters at places of function calls. Also, it maps the links between call site variables and the return values of the invoked functions. This mapping is crucial for controlling the flow of values across function boundaries. As a result, it allows for identifying infeasible paths that extend over multiple procedures and loops.

This method allows us to evaluate the feasibility of paths incrementally as they are generated. This can be achieved by using the mapped data. Consequently, a path is marked as infeasible if, at any point, no values can satisfy the path conditions.

Although this approach helps us mitigate impractical paths, implementing its concepts from the code level to the UML activity diagram is challenging due to the lack of a clear relation between the variables and entities. Additionally, the suggested approach is not easily customizable, making it difficult for test developers to specify the conditions that make the test infeasible.

4.5 Cross–verification of CRUD matrices

Cross-verification of CRUD matrices, as introduced in [35], involves comparing matrices created by different stakeholders at various stages of development. This technique aims to improve the consistency and accuracy of static testing by identifying discrepancies early on.

Cross-verification, along with other enhanced static testing techniques, has significantly reduced the number of undetected data consistency defects. These methods greatly improve the overall quality and efficiency of the testing process. This is especially true in complex systems such as enterprise and IoT environments, where consistency is paramount.

However, while this method enhances data quality in test scenarios, it does not address the generation of infeasible test cases.

4.6 Extension of CRUD matrix

Authors of [36] propose to extend CRUD matrices by new operations that aim to improve data relationship management and optimize test coverage:

- **Operation Influenced (I)**: Data entity e_1 is influenced (denoted as $I(e_2)$ by the function $f \in F$ if the function f uses a data entity e_2 through one or more of the C, U, D operations, and the particular data content of entity e_1 is changed as a consequence of function f, which primarily uses the entity e_2. In a CRUD matrix, the situation described above is captured by operation $I(e_2)$ in the cell for the function f and entity e1.
- **Operation Best Read (B)**: This operation helps select the most significant read operation based on various criteria, such as the extent of data attributes covered, frequency of access, and importance from a testing perspective.

Consequently, the sequence of operations in the test cases is dynamically created based on the interactions defined in the CRUD matrix.

However, expanding CRUD matrices adds complexity and emphasizes static data relationships. This approach may not be ideal for dynamic interactions and state changes in certain software systems. Generating paths for Activity diagram–based models may require a more dynamic approach.

Although the proposed extension can improve data management, it does not directly solve the problem of generating infeasible paths, which is a key focus of this thesis. Consequently, this method is not suitable for achieving the thesis's objective.

4.7 Summary

This chapter has reviewed various methodologies from recent academic research that address challenges associated with data treatment in automatic test case generation. Each method tackles distinct parts of the problem and offers specialized solutions to mitigate the issue of infeasible test paths.

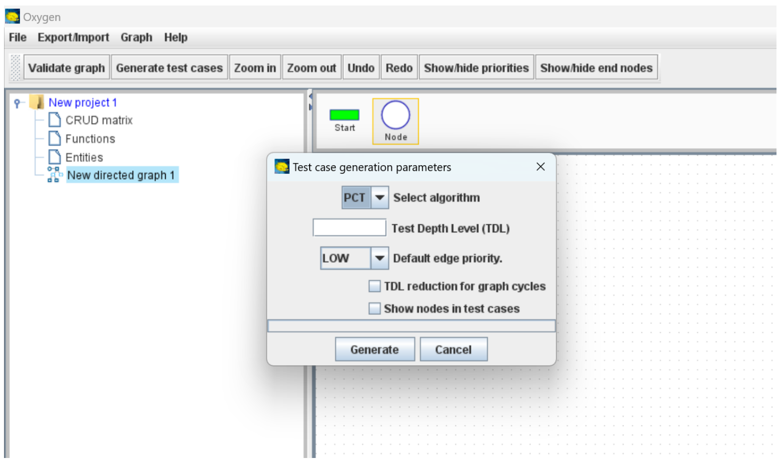
Among the methods reviewed, the concepts introduced in [31] are shown to be the most suitable for thesis needs. While methods like the Extension of CRUD Matrix focus on static data relationships, the proposed method is designed to handle dynamic interactions and state changes effectively. The proposed method analyzes feasibility directly during the test case generation process, reducing the occurrence of infeasible test paths more efficiently than heuristics-based approaches. In contrast with the interprocedural dataflow analysis approach, the proposed method is easily customizable. Thus, test developers can easily specify conditions for test infeasibility determination.

Chapter 5

Analysis

The chapter justifies the reasons for the chosen testing technique and technologies. Also, it describes Oxygen capabilities for generating test scenarios. Moreover, it introduces a constraint that helps to solve the example from the thesis motivation. In conclusion, it analyzes the limitations and usage areas of the proposed solution.

Starting from this chapter, the terms "test cases" and "path in the graph" are used interchangeably, while a "test step" in the generated test case is considered to mean the same as a graph's edge.



PCT Select algorithm

Test Depth Level (TDL)

LOW Default edge priority.

☐ TDL reduction for graph cycles

☐ Show nodes in test cases

Figure 5.1: Selection of parameters for test case generation in Oxygen

5.1 Oxygen

Oxygen[1] is an experimental platform for research in MBT developed by the Faculty of Electrical Engineering at the Czech Technical University in Prague[2]. The platform–independent application is written in Java and supports XML, CSV and JSON formats for importing/exporting graphs and test cases. The test cases may be generated using SUT representations as a directed graph

[1]http://still.felk.cvut.cz/oxygen/

[2]http://still.felk.cvut.cz/

or activity diagram. Section 2.3.1 describes how these representations can be used interchangeably. To obtain test cases, a user should specify the desired test coverage TDL–N and choose between the Process Cycle Test or the Prioritized Process Test technique. Figure 5.1 shows the abilities for parameters setup.

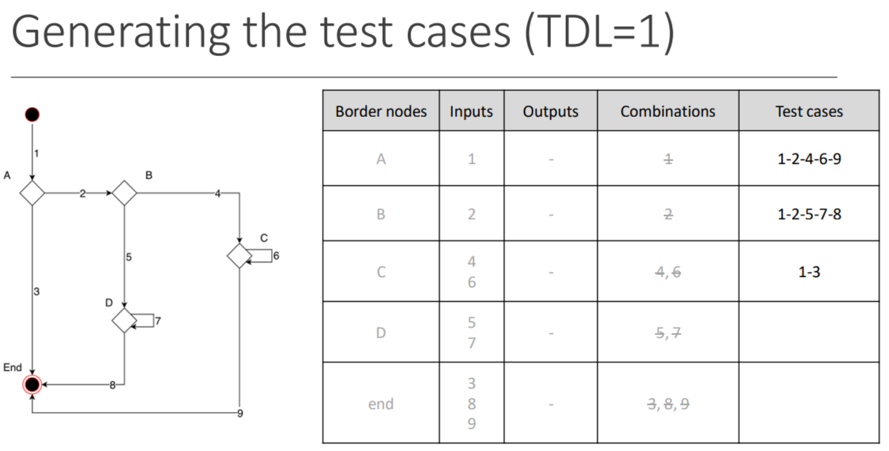
Recall the situation stated in the thesis motivation 1.1: e–shop is a SUT where users can log in, sign up, browse items, add items to the card, and create an order. An activity diagram modelled for such a SUT using Oxygen is presented on figure 5.3. Test paths generated using test coverage TDL–1 for this example are presented in the table 5.1.

Path	Steps	Feasibility
1	START - web loading - is logged in - does not add items to the cart - creating the order - order is created – END	Infeasible
2	START - web loading - is not logged in - is not registered - signing up - successful registration – signing in - items browsing - adds items to the cart - adding items to the cart - finishing the order - creating the order - order is created – END	Feasible
3	START - web loading - is not logged in - is registered - signing in - items browsing - does not add items to the cart - creating the order – order is created - END	Infeasible

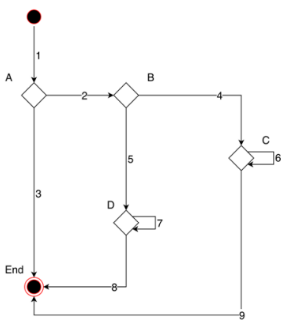
Table 5.1: Test Paths for motivational example 1.1 generated by Oxygen employing TDL–1 coverage

The principle of test path generation for TDL–1 can be generalized for TDL–N. The idea is to generate all possible edge combinations for each decision point. Then, we need to combine the generated combinations into test paths so each path starts in the ‘Start‘ node and ends in one of the ‘End‘ nodes. One can see the principle as close to the Dominoes game. The process is executed until all edges are used. The example presented on 5.2 can be seen in the following way:

Firstly, we generate all edge combinations for each vertex and write them in the row "combinations" for the appropriate node. Then, we need to combine the obtained combinations into test cases. As can be seen, we can directly obtain the test path `1 - 2 - 4 - 6 - 9`; note that there is no way to extend this path as the `9` edge already leads to the finish vertex. After this operation, there are only edges `5, 7, 3, 8, 9` left. Let’s start by employing edge `5` and use as many combinations as possible while creating a test path. This way, we can get `2 - 5 - 7 - 8`. Nevertheless, the path should start from the starting vertex, so we need to add edge `1` to the beginning of the path and get `1 - 2 - 5 - 7 - 8`. After this step, we only need to cover edge `3`. Again, the path should start from the starting vertex, so we append edge `1` to it, getting `1 - 3`. As a result, we have paths `1 - 2 - 4 - 6 - 9`, `1 - 2 - 5 - 7 - 8` and `1 - 3` that cover the presented graph using test coverage TDL-1.



Generating the test cases (TDL=1)



Border nodes	Inputs	Outputs	Combinations	Test cases
A	1	-	~~1~~	1-2-4-6-9
B	2	-	~~2~~	1-2-5-7-8
C	4 6	-	~~4~~, ~~6~~	1-3
D	5 7	-	~~5~~, ~~7~~	
end	3 8 9	-	~~3~~, ~~8~~, ~~9~~	

Figure 5.2: The principle of test path generation for TDL–1 employed in Oxygen. Source: Lab 11, course BE4M36ZKS at CTU FEL

It can be seen that the 1st and 3rd test scenarios are infeasible – an order cannot be created when the cart is empty. Explicitly, the sequence of edges does `not add items to the cart -- order is created` is infeasible. It can also be seen this way: an edge `adds items to the cart` should always precede `order is created`.

A graph traversal resulting in a feasible test scenario (№2) is shown on 5.5, while 5.6 and 5.4 present graph traversals resulting in infeasible test scenarios.

5.2 Solution proposal

This section describes the requirements on the implemented algorithm and provides details on the constraints definition, its usage and limitations.

5.2.1 Graph structure

As shown in 5.1 for a SUT modelled as a directed graph, Oxygen generates test cases represented as a sequence of edges, not vertices. Therefore, making the proposed algorithm generate test cases as a sequence of edges is intuitive, thus facilitating the comparison results obtained from these algorithms. Oxygen allows a graph to have cycles by implementing a separate algorithm that executes TDL reduction for graph cycles. Also, it allows parallel edges in the graph by an internal representation using a dummy node. Nevertheless, these factors do not significantly affect the application of the constraints in the proposed algorithm. Therefore, I consider a Directed Acyclic Graph without parallel edges in the implemented algorithm.

5.2.2 Constraints

I propose implementing a newly defined constraint `FOLLOWS` together with constraints described in [31] to prevent particular edge combinations from occurring in the generated test cases.

The proposed constraint `FOLLOWS` specifies that if an edge A occurs in a test path, another B must appear before it in the path. For example, the constraint `E FOLLOWS B` means that edge B always precedes edge E in any path. For example, for the mentioned constraint, a path `A --> B --> E` and `B --> C --> E` are valid (B precedes E), while `A --> E` is invalid (there is no B before E). Analogously, `D EXCLUDES E` would mean that edge E cannot follow D in the path (even non–consequently), so `D --> B --> C` is valid, while `D --> E --> C` and `D --> C --> E` are not. The proposed constraint representation is the optimal approach in the current development phase as it is not as complex as OCL and facilitates debugging and application testing. Still, the proposed implementation leaves space for an easy extension to OCL in future work.

Currently, the implementation supports `FOLLOWS`, `EXCLUDES`, and `ONLY_ONE` constraints. However, the `ONLY_ONE` constraint at the moment does not prevent any combinations from occurring in the path. This is because of the graph's structure (recall, it is DAG), which prevents any edge from appearing more than once in a path.

This notation will be used for the following examples: $Create - C$, $Read - R$, $Update - U$ and $Delete - D$.

The reason for the infeasibility of the early presented test case may be generalized in the following way: the object is read before it was created. It is forbidden to $Read$, $Update$ or $Delete$ objects before $Creation$. The described rule can be generalized further: the C operation on an object should precede any other operation in a testing path. Moreover, we should avoid situations where the C operation is called after executing C and before D. For example, `C --> U --> C` is not allowed, while `C --> D --> C` is allowed.

The presented observation may be a good starting point for a tester to identify constraints for the SUT. These constraints then need to be applied to the SUT.

5.2.3 Path Generation

Developing the desired algorithm may be separated into two parts: all possible path generation in the first step and filtering infeasible paths by constraints application in the second step. This approach is not the most effective, as large inputs will result in high computational demands caused by test case re–generation. Therefore, the approach when the constraints are applied during path generation, without the need for additional re–evaluation, is desirable.

Evaluation of Result paths:

The computational effectiveness of the developed algorithm is not stated in the thesis objectives. Moreover, the computational complexity of the algorithm implemented in Oxygen is unknown. Thus, the only thing considered in the evaluation is the quality of generated test cases, described in terms of generated paths, the total number of edges, and the number of infeasible test cases based on predefined constraints.

Test coverage: As a test coverage, Edge Cover was chosen, as it is supported by Oxygen, facilitating the evaluation of the results.

Considering the requirements on test cases for TDL–1 without applied constraints, which start in a starting node and end in one of the finishing vertices, the observations for a graph representation can be formulated more formally:

Objective: Find the paths set T that cover all edges in a directed, acyclic and unweighted graph $G = (N, E)$. Each path should start in a starting node *start* and end in one of the ending nodes $e_i \in N_e$. Neither cycles nor parallel edges are allowed.

There might be several options to implement this algorithm by utilizing genetic algorithms similar to those described in [37, 38, 39]. Another option to provide an effective time complexity is modelling the problem as a well–known optimization problem and reusing an existing solution. Several representations for the current problem may exist, such as Minimal Flow problem [40], Minimum Path Cover or Hamiltonian Cycle Problem. A similar approach is presented in [41].

Algorithm Overview: The idea is to generate all possible paths using a Depth–first search (DFS) while considering the constraints during generation. Then, the set of generated test cases should be optimized based on the predefined criteria, namely in terms of test path amount and the total length of the test set.

As the core algorithm step is to find all paths from the starting node to all end vertices, there is also a need to have defined starting and ending vertices. To make it closer to Oxygen representation, the user only chooses the start node; 'finish' vertices are automatically recognized as the vertices that do not have outgoing edges.

It is worth mentioning that ensuring the desired coverage TDL–1 (appearing of all edges in the resulting test paths) may not be possible for some constraints in particular graphs. Consider the example from 5.3. If the constraint `"Order is created" FOLLOWS "Adds items to the cart"`,
is applied, the edge `"Does not add items to the cart` with the proposed constraint definition will never appear in test cases. The reason is that the only possible way of getting from this edge to the 'End' vertex is by passing edge 'Does not add items to the cart', which requires edge 'Adds items to the cart' to be present in the current path. This is impossible for the presented graph's structure.

If the user has just recently registered and logged into the e-shop for the first time, the presented action combination `"Does not add items to the cart" -> "Order is created"` is indeed infeasible. Nevertheless, the combination becomes feasible when an item has been added to the cart during the previous user's session. This situation may be reflected in future constraint extensions, as it depends on the state of the SUT. Moreover, the presented constraint definition can be seen as static because of its inability to be readjusted during test generation. Making constraints be processed dynamically is another direction for future work.

5.2.4 Choice of Technologies

- **Java 22**. Although there is no requirement for integration with Oxygen yet, I decided to implement the solution in Java. This way, it will be easier to integrate in the future. The chosen (not LTS [3] version) is caused by my eagerness to check new features, such as unnamed variables and string patterns. Apart from that, Java 22 offers improved performance, new language features, and enhanced security. [42].
- **Maven**. This build tool allows me to manage libraries, which I use for Spring, JUnit, etc. Maven provides a structured approach to project management, making dependency management and build processes more streamlined than other tools like Gradle. [43]
- **JUnit 5** testing framework allows me to check that the application operates as intended. It provides a flexible testing framework that supports features like parameterized tests. Details on JUnit 5 usage in the project can be found in Chapter 6.3. [44]
- **SonarQube** helps make the code more secure and improves its quality by enforcing coding standards. [45]
- **Spring Boot** is used for convenient logging and property handling. It simplifies the configuration and setup process, allowing me to speed up the development. Additionally, this framework offers web capabilities, so in future, the application can be easily extended with web functionality if needed. [46]
- **Cypress** was chosen for its ease of setup (compared to Selenium, for example), real-time reloading, and ability to work with modern JavaScript frameworks. Cypress provides a user-friendly environment, unlike Puppeteer, primarily focusing on headless browser testing. It is used to validate the implemented algorithm for the e-shop `https://e-shop.webowky.cz/` and compare the results with Oxygen.

[3] `https://en.wikipedia.org/wiki/Java_version_history`

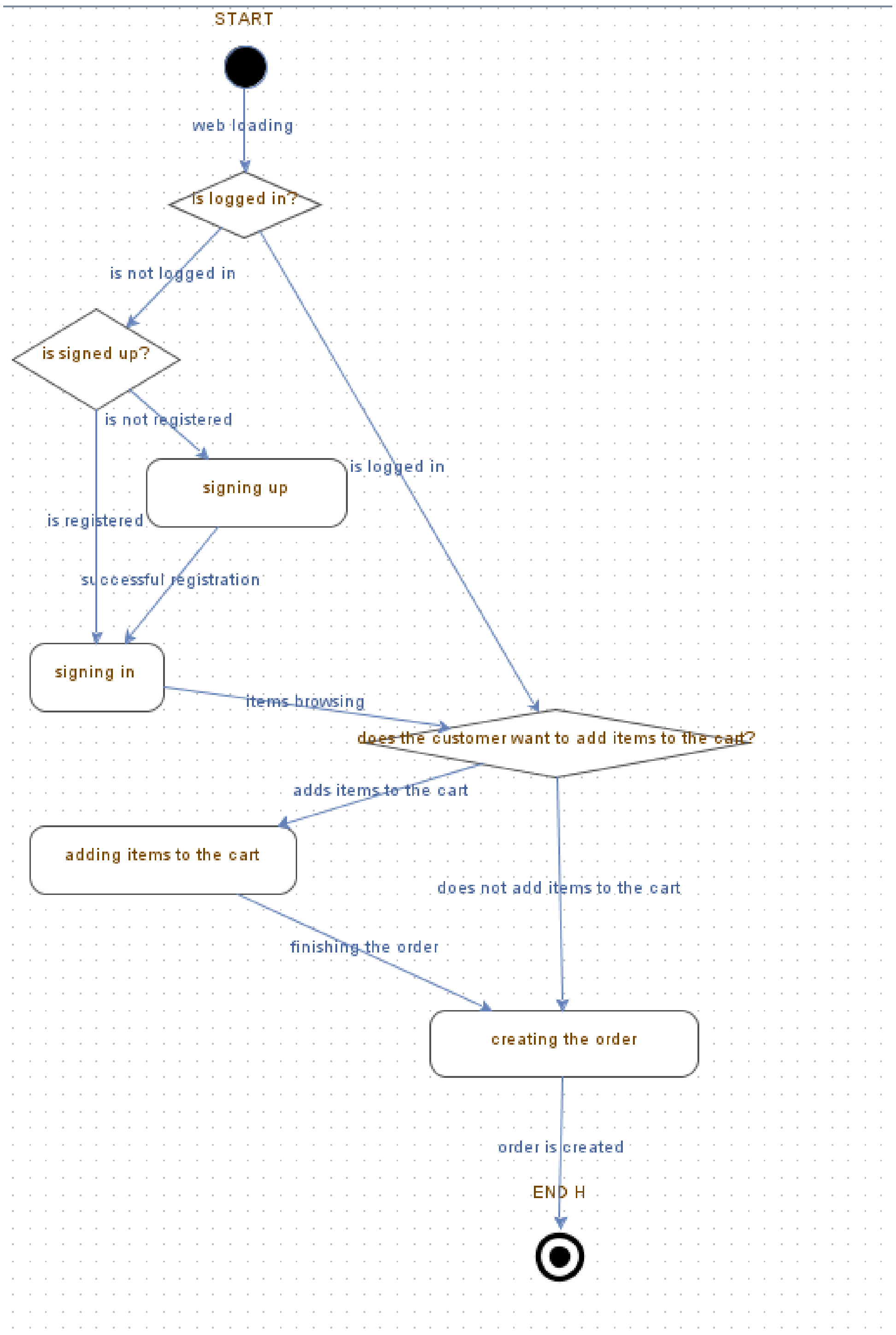


Figure 5.3: A motivational example 1.1 for e–shop modelled as a directed graph using Oxygen

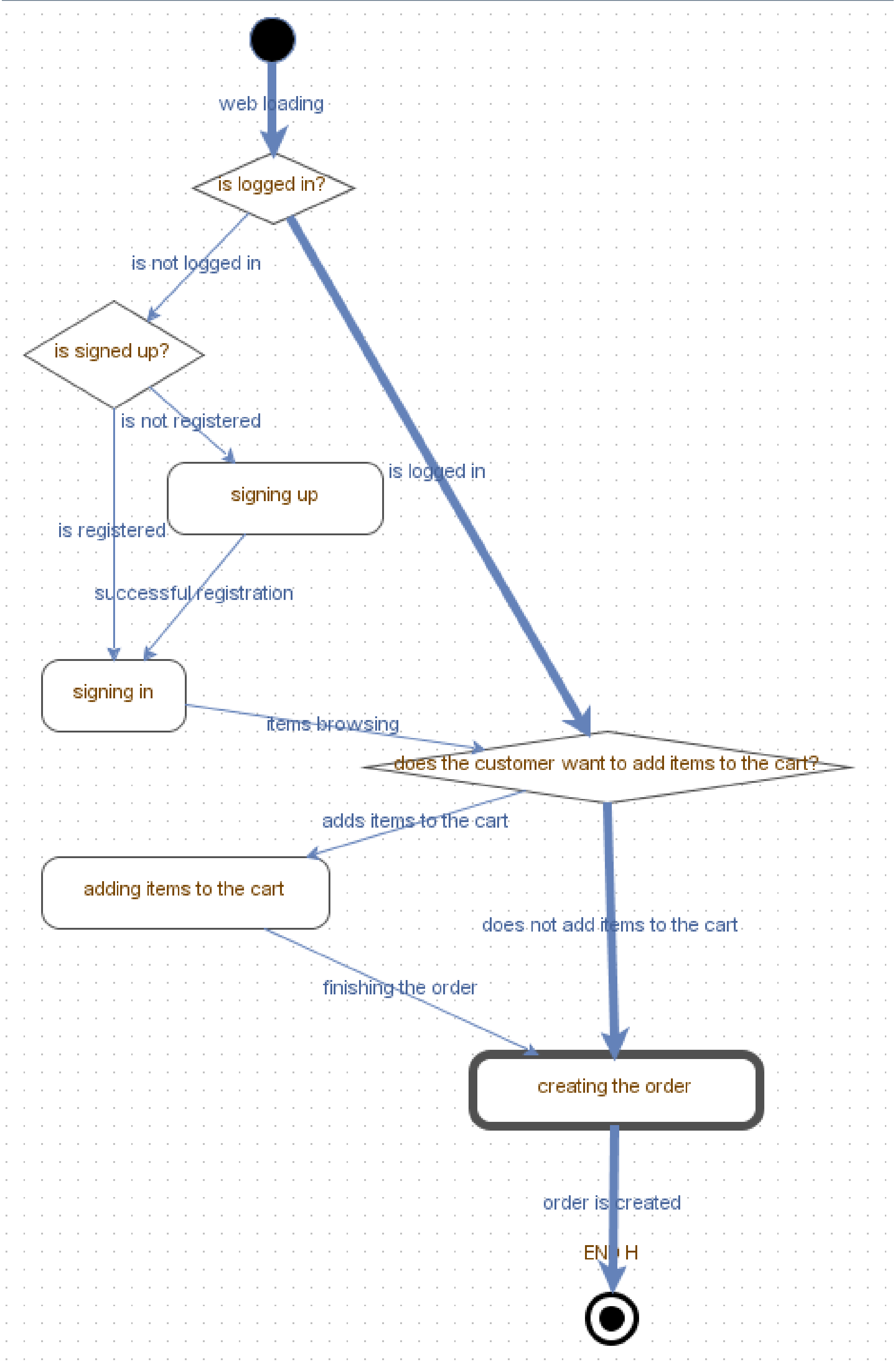


Figure 5.4: E–shop: path №1 generated by Oxygen

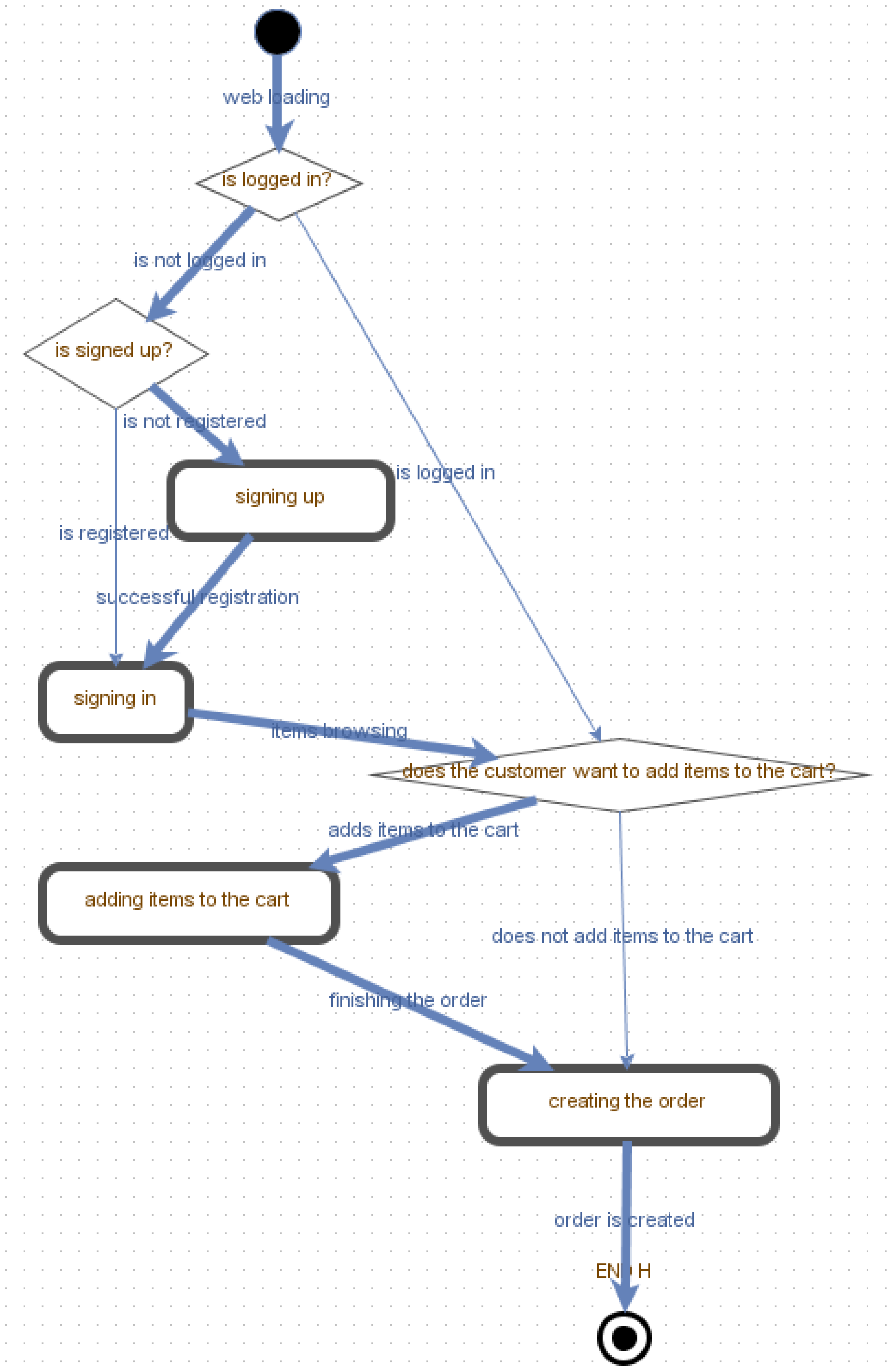


Figure 5.5: E–shop: path №2 generated by Oxygen

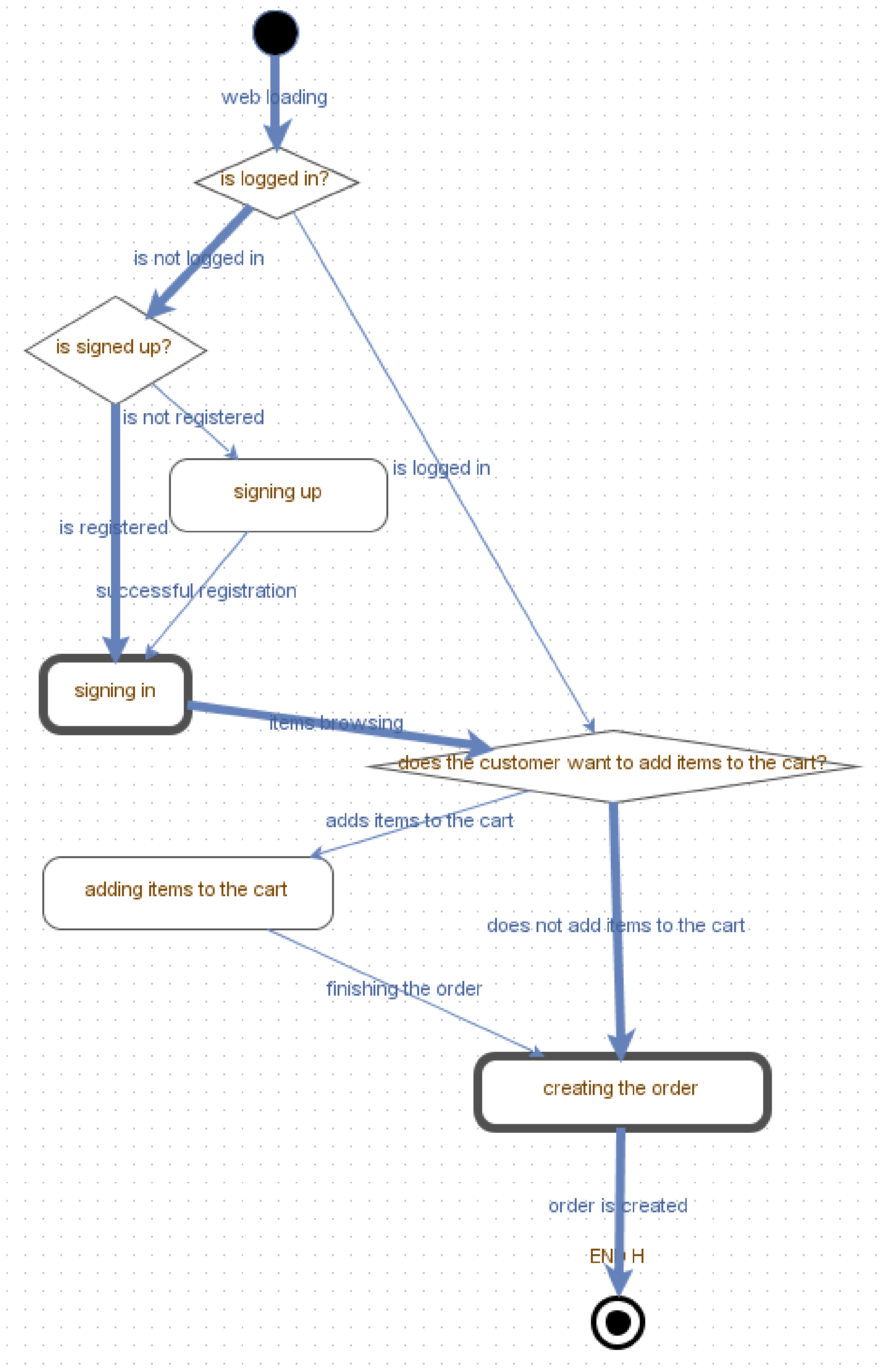


Figure 5.6: E–shop: path №3 generated by Oxygen

Part III

Development

Chapter 6

Development

The chapter details the proposed algorithm. Further, it describes implemented design patterns and provides ideas on how they may be useful for future application extensions. The chapter ends by detailing the strategies employed to guarantee application quality.

6.1 Algorithm Overview

Objective: Let $G = (V, E)$ be an unweighted DAG without parallel edges, where V is the set of nodes and E is the set of edges. Denote $\text{start} \in V$ as a starting node for each path, and $N_e \subseteq V$ as the set of ending nodes.

The objective is to find the set of paths $\mathcal{T}$ that covers all edges E in graph G. Each path $\pi \in \mathcal{T}$ should start at a starting node start and end at an ending node $e_i \in N_e$. All paths must satisfy the defined constraints.

Structure: The implemented algorithm aims to find the path set that meets the requirements specified in 5.2.3.

The algorithm starts by finding all paths that meet the constraints. This part is based on a recursive DFS algorithm. The key differences from the classic DFS implementaion are constraint checking via the `CanIncludeEdge` method, storing all valid paths, and allowing vertices to be revisited in different paths.

Once the set of all possible paths is found, it should be ensured it meets the criteria extensively defined in 5.2.3. The implemented algorithm iteratively selects the subset that provides the maximum unique coverage, adds it to the cover and updates the covered elements. This process continues until all elements in the universe are covered or no more useful subsets can be added. In the consecutive optimization pass, the algorithm removes redundant subsets by ensuring each subset in the final cover adds unique elements not already covered by previously included subsets.

The presented approach does not guarantee an optimal solution; however, it is favoured for its simplicity and reasonably good performance in test instances.

6.2 Application Architecture

Given the exploratory nature of the proposed application, which aims to validate the suitability of the chosen method, it is evident that several extensions may be needed in future. Therefore, the application architecture should be designed to implement these extensions easily.

6.2.1 Input Processing

The implemented application can process both input obtained from Oxygen in JSON format and the ones defined in plain text. Edge ID is not mandatory; it will be autogenerated when not specified. The text file should have the following format:

```
// VERTICES
START A B C D E F
// EDGES
START A 1
START B 2
START C 3
A D 4
B F 5
C E 6
D E 7
// CONSTRAINTS
5 FOLLOWS 3
```

Oxygen already supports several types of exports and may support more in future. Therefore, a `Strategy` design pattern was implemented. It allows an easy extension to support further input formats if such requirements arise. The core component of this design is the `GraphInputStrategy` common interface for all concrete strategies that read graph data from different formats, such as manual entries, JSON files, or potentially XML files in the future.

The following strategies are already a part of the presented solution:

- **ManualInputStrategy:** Implements the `GraphInputStrategy` for reading graph data from manually prepared text files. This strategy is particularly useful during development and allows to use the application without prior graph generation in Oxygen.

- **OxygenJsonInputStrategy:** Used for reading graph data from JSON files exported from Oxygen.

The `GraphReaderContext` references a `GraphInputStrategy`. This setup allows the application to change the reading strategy based on the `app.input.source` in the `application.properties` processed by Spring framework.

By isolating the input reading logic into specific strategy classes, the system remains open for extension but closed for modification. New input formats can be supported by introducing new strategy classes without changes in existing code, thus following SOLID's Open/Closed Principle [1].

6.2.2 Constraints

Applying constraints is considered the main objective of the thesis. Hence, I prioritize its quality, considering scenarios for potential extensions.

Consider, for example, a new requirement on applying constraints across multiple paths. This could be implemented by extending the existing `apply` method or introducing new methods in the `EdgeConstraint` interface to handle collections of paths.

EdgeConstraint Base Class

The `EdgeConstraint` class is an abstract base class implementing `IEdgeConstraint` for different edge constraints. It encapsulates `an expression` attribute that holds the original value of the constraint before parsing, which may be a human-readable string or a formal OCL definition suitable for parsing. Keeping the original expression in the constraint allows its dynamic changes in runtime in the future.

ConstraintParser

`ConstraintParser` interface separates parsing logic from constraint representation and application. It facilitates the addition of new parsing methods without modifying the existing code.

`ANTLRConstraintParser` is a placeholder for an ANTLR-based parser that may be implemented when new requirements for more complex parsing arise.

6.3 Testing

The application is extensively covered with unit and e2e tests using the JUnit 5 framework; some tests are parameterized with custom-defined input-providing methods.

6.3.1 Test Data Preparation

Testing path generation without constraints. E2E tests aim to verify the correctness of the generated paths. For this, the results generated by Oxygen are taken as the reference base. Since there can be multiple methods to create a set of paths, it is impractical to design tests that validate the precise

[1]SOLID is an acronym for five design principles intended to make object-oriented designs more maintainable, source: `https://www.digitalocean.com/community/conceptual-articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design`

presence of elements in the resulting paths. Therefore, I defined the criteria that must be met to consider the tests successful. They are the following:

- The Generated set of paths should cover all edges in the graph (TDL-1 test coverage).
- The number of generated paths is not more than the one generated by Oxygen.
- The number of edges in the paths is not more than the one generated by Oxygen (thus, revisiting edges is minimized).
- Each path starts in a starting vertex and ends in one of the 'Finish' vertices.

Listing 1 presents paths assertions used in the implemented JUnit tests.

Testing path generation with constraints. Although Oxygen was the trustworthy baseline for basic inputs, it cannot generate test paths with employed constraints. Therefore, I manually computed the expected outputs and prepared test files. For inputs with constraints, E2E tests assert the exact appearance of specific paths in the resulting set.

6.4 Evaluation

This section contains the evaluation of the proposed algorithm against the Oxygen, including the motivational example and several other graphs.

6.4.1 Algorithm Validation on a Real Application

This section describes the results obtained for the example from thesis motivation. The table 6.1 presents the mapping of edge-id to edge-name that was used for modelling:

Edge ID	Edge Name
1	Web loading
2	Is not logged in
3	Is logged in
4	Is registered
5	Is not registered
6	Items browsing
7	Successful registration
8	Adds items to the cart
9	Does not add items to the cart
10	Finishing the order
11	Order is created

Table 6.1: Edge IDs and their corresponding names

Oxygen

The test paths generated by Oxygen for the e-shop SUT from 5.1 **without applied constraint** are the following:

Test case №1: `[1, 3, 9, 10, 11]`

```
1. Web loading
3. Is logged in
9. Does not add items to the cart
10. Finishing the order
11. Order is created
```

Test case №2: `[1, 2, 5, 7, 6, 8, 10, 11]`

```
1. Web loading
2. Is not logged in
5. Is not registered
7. Successful registration
6. Items browsing
8. Adds items to the cart
10. Finishing the order
11. Order is created
```

Test case №3: `[1, 2, 4, 6, 9, 10, 11]`

```
1. Web loading
2. Is not logged in
4. Is registered
6. Items browsing
9. Does not add items to the cart
10. Finishing the order
11. Order is created
```

The presented test case set will be referred to as 'Test Suit Oxygen' later.

Implemented algorithm

For the same SUT, **with the applied constraint** `11 FOLLOWS 8`, which is the edge ID representation of `"Order is created" FOLLOWS "Adds items to the cart"`,

the implemented algorithm generates the following test paths:

Test Case №1 `[1, 2, 4, 6, 8, 10, 11]`

Test Case №2 `[1, 2, 5, 7, 6, 8, 10, 11]`

Test Case №3 `[1, 3, 8, 10, 11]`

The generated test paths are interpreted as the following test cases:

Test case №1: `[1, 2, 4, 6, 8, 10, 11]`

```
1. Web loading
2. Is not logged in
4. Is registered
6. Items browsing
8. Adds items to the cart
10. Finishing the order
11. Order is created
```

Test case №2: `[1, 2, 5, 7, 6, 8, 10, 11]`

```
1. Web loading
2. Is not logged in
5. Is not registered
7. Successful registration
6. Items browsing
8. Adds items to the cart
10. Finishing the order
11. Order is created
```

Test case №3: `[1, 3, 8, 10, 11]`

```
1. Web loading
3. Is logged in
8. Adds items to the cart
10. Finishing the order
11. Order is created
```

The presented test case set will be called ‘Test Suit Constraint‘ later.

Evaluation

The tests for validating generated test cases were prepared using the Cypress framework for publicly accessible e-shop `https://e-shop.webowky.cz/`. The example of the implemented test can be found on listing 2.

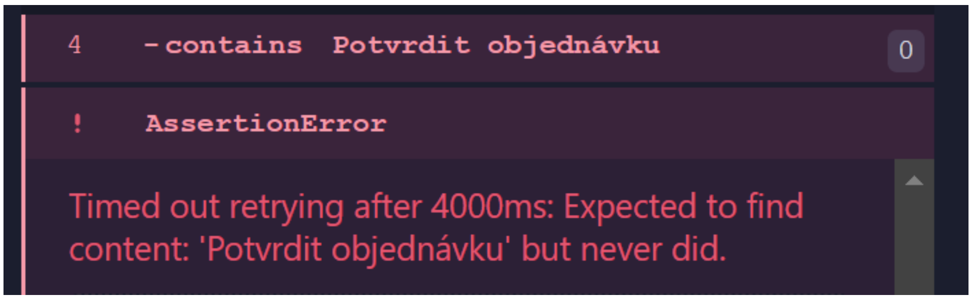
Two out of three tests have failed in the ‘Oxygen Test Suit‘: test №1 and test №3. The problem arises while attempting to locate the "Potvrdit Objednávku" (Create an order) button, as seen on the screenshot 6.1. It results in a false evaluation of the SUT as malfunctioning, even though it functions as intended.

There were no test failures in the second test suite; all three tests passed successfully.

6.4.2 Computed Results

The following parameters are used for evaluating the test cases generated by Oxygen and by the newly implemented algorithm:

- Amount of test cases, which corresponds to the number of generated test paths. The smaller amount means better performance as testers need to execute/prepare fewer tests.



```
4    -contains  Potvrdit objednávku                0
!    AssertionError
Timed out retrying after 4000ms: Expected to find
content: 'Potvrdit objednávku' but never did.
```

Figure 6.1: The screenshot of failed test for `https://e-shop.webowky.cz/` in Cypress

- Total amount of steps in the test cases. It corresponds to the total amount of edges in the generated test paths. The shorter test cases are beneficial for resource preservation.
- Amount of infeasible test cases based on defined constraints. Although Oxygen does not allow for defining any constraints, this metric helps evaluate the proposed algorithm.
- Amount of uncovered edges. Some edges may be left covered (therefore, TDL-1 will not be guaranteed) due to the application of the constraint preventing particular edge combinations from being present in a path.

The results obtained using TDL-1 coverage are presented in 6.2.

It can be seen that not only were the test scenarios sometimes shorter, but their number was also reduced, particularly for larger graphs. Additionally, the implemented algorithm effectively prevents the generation of infeasible test cases. However, this comes at a cost: some edges may not appear in the test cases due to the applied constraints. These results align with the assumptions made in the analysis section and are, therefore, expected.

```
assertAll(
    () -> assertTrue(doSubsetsCoverSet(cover,
    ↪  graph.getIdsOfGraphEdges()),
    "Computed paths should cover all edges in the graph"),
    () -> assertTrue(actualCoverSize <= expectedCoverSize,
                        String.format("Number of found paths
                        ↪  (%d) should not exceed the number
                        ↪  (%d) than Oxygen computed:
                        JSON=%s CSV=%s", actualCoverSize,
                        ↪  expectedCoverSize, manualFile,
                        ↪  resultsFile)),
    () -> assertTrue(actualTotalEdges <= expectedTotalEdges,
                        String.format("Total number of edges
                        ↪  in paths (%d) should not exceed
                        ↪  those (%d) found by Oxygen:
                        ↪  JSON=%s CSV=%s", actualTotalEdges,
                        ↪  expectedTotalEdges, manualFile,
                        ↪  resultsFile)),
    () -> cover.forEach(
        path -> {
                Integer firstEdge = path.isEmpty() ? null :
                ↪  path.getFirst();
                Integer lastEdge = path.isEmpty() ? null :
                ↪  path.getLast();
                assertTrue(firstEdge != null &&
                ↪  Objects.equals(
            graph.getEdgesById().get(firstEdge).sourceVertex,
            ↪  startVertex.getId()),
            "Path should start from the START vertex.");
                assertTrue(lastEdge != null &&
                ↪  graph.getFinishVertices().contains(
            graph.getEdgesById().get(lastEdge).targetVertex),
            ↪  "Path should end at a finish vertex.");
                })
);
```

Listing 1: Implemented assertions for validating generated paths

```
it('Test Case 3: Logged in, adding items to the cart, and
↪  finishing the order', function () {
    cy.visit('https://e-shop.webowky.cz/');
    cy.wait(2000);

    cy.contains('Přihlásit se').click();
    cy.contains('Už jste zaregistrován?').click();

    cy.get('#txt_login_email').
        type('john.doe_test@example.com');
    cy.get('#txt_login_password').
        type('securePassword123');
    cy.get('#btn_login').click();

    cy.get('img.logo.img-responsive').click();
    cy.wait(2000);

    cy.get('.ajax_add_to_cart_button').
    first().click();
    cy.contains('Produkt byl úspěšně přidán do nákupního
    ↪  košíku').
    should('be.visible');

    cy.reload();
    cy.get('a[title="Zobrazit můj nákupní košík"]').click();

    fillAddressForm();

    simulateObjednatButton();
});
```

Listing 2: A test implemented in the Cypress based on the test case generated by the Constraint algorithm

Graph ID	**Algorithm**	**Edges/ Nodes**	**Test Cases/ Total Steps**	**Constraint**	**Infeasible**	**Uncovered Edges**
1	Constraint	13/9	5/19	–	0	0
	Oxygen	13/9	5/19	–	0	0
2	Constraint	10/8	4/16	–	0	0
	Oxygen	10/8	4/16	–	0	0
3	Constraint	23/16	13/43	–	0	0
	Oxygen	23/16	14/46	–	0	0
4	Constraint	23/16	11/34	–	0	0
	Oxygen	23/16	12/37	–	0	0
5	Constraint	12/10	3/20	–	0	0
	Oxygen	23/16	12/37	–	0	0
6	Constraint	13/9	5/19	–	0	0
	Oxygen	13/9	5/19	–	0	0
7	Constraint	25/13	12/52	–	0	0
	Oxygen	25/13	13/54	–	0	0
5F	Constraint	23/16	2/15	12 FOLLOWS 7 8 FOLLOWS 7	0	3
	Oxygen	23/16	12/37	12 FOLLOWS 7 8 FOLLOWS 7	1	0
6E	Constraint	13/9	5/19	4 EXCLUDES 10	0	0
	Oxygen	13/9	5/19	4 EXCLUDES 10	1	0
6M	Constraint	13/9	5/19	4 EXCLUDES 10 10 FOLLOWS 2	0	0
	Oxygen	13/9	5/19	4 EXCLUDES 10 10 FOLLOWS 2	1	0
7E	Constraint	25/13	12/52	3 EXCLUDES 24 13 EXCLUDES 25	0	0
	Oxygen	25/13	13/54	3 EXCLUDES 24 13 EXCLUDES 25	1	0
7M	Constraint	25/13	10/45	9 FOLLOWS 2 21 FOLLOWS 13 6 EXCLUDES 25 5 EXCLUDES 25	0	2
	Oxygen	25/13	13/54	9 FOLLOWS 2 21 FOLLOWS 13 6 EXCLUDES 25 5 EXCLUDES 25	6	0

Table 6.2: Comparison of the results obtained from the Constraint Algorithm and Oxygen Algorithm

Part IV

Conclusion

Chapter 7
Conclusion

This thesis has explored several aspects of test scenario generation for data-centric applications. Specifically, it focused on extending MBT by reflecting the data requirements of SUT using constraints. Consequently, a new Constraint, `"FOLLOWS"`, has been introduced as an extension to Negative-Constrained Path-based testing.

In this thesis, I addressed the limitations of the Oxygen tool, specifically its inability to reflect data requirements into the automatic generation of test cases. To address this limitation, I designed and implemented an algorithm capable of generating scenarios with defined constraints. It allows the model to be accurately abstracted, which in turn restricts the generation of infeasible test cases. Afterwards, the algorithm was validated on Webowky[1] e-shop and several abstract graphs, demonstrating its effectiveness in generating realistic test cases. The comparison with the Oxygen tool output highlighted the presented approach's advantages.

The thesis also researches existing ways of constraint representation. Although a basic approach was chosen, there is space for future extension to OCL notation. To support such enhancements, the proposed architecture is designed to be easily extensible, specifically allowing for the smooth integration of additional constraints, input formats, and parsing techniques.

This research contributes to enhancing testing processes by addressing the need for more realistic test scenarios. By preventing scenarios where a test cannot continue because the defined constraints were not met due to inconsistent data, the implemented algorithm improves the quality and reliability of test scenarios.

In summary, this thesis has successfully met the initial objectives, namely:

- Designing and implementing an algorithm that considers data constraints in path-based test case generation.
- Validating the algorithm on a real application.
- Evaluation of the algorithm's capabilities over the Oxygen in generating valid test scenarios.

The primary contributions of this work include:

[1] `https://e-shop.webowky.cz/`

- Identification of Infeasibility Factors - the situations leading to the generation of infeasible test cases were identified and carefully described.
- Implementation of the algorithm that allows to prevent particular parts of the system from appearing in the test cases.

7.1 Future Research Directions

While the proposed algorithm addresses many of the limitations of existing MBT tools, there are still places for future extensions.

The algorithm's efficiency heavily relies on the initial model's completeness and the accuracy of constraints. Therefore, introducing additional constraints and refining their definitions using OCL can provide better results for more complex systems. Moreover, allowing constraints to be applied across multiple paths can improve the algorithm's versatility. Additionally, there may be proposals to keep the desired code coverage even when constraints are applied. Extending the algorithm by the ability to handle cycles and parallel edges within the model will allow for more realistic scenarios. Proposing techniques for dynamically managing constraints will enable the model to adapt to changes in real-time.

The algorithm may be optimized to work with large data. Investigating ways for such enhancement may involve using mathematical optimization.

By focusing on the presented ideas, the implemented algorithm can be further enhanced, making it a more powerful tool for generating accurate test cases.

Appendices

Appendix A

Bibliography

1. UTTING, Mark; LEGEARD, Bruno. *Practical Model-Based Testing: A Tools Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006. ISBN 0123725011.
2. MYERS, Glenford J.; SANDLER, Corey; BADGETT, Tom. *The Art of Software Testing*. 3rd ed. John Wiley & Sons, 2011.
3. INTERNATIONAL SOFTWARE TESTING QUALIFICATIONS BOARD. *ISTQB Glossary of Testing Terms*. 2024. Available also from: `https://glossary.istqb.org/en_US/home`. Accessed: 2024-04-13.
4. AMMANN, Paul; OFFUTT, Jeff. *Introduction to Software Testing*. Cambridge: Cambridge University Press, 2016. ISBN 9781107172012. Available from DOI: `DOI:10.1017/9781316771273`.
5. FEWSTER, Mark; GRAHAM, Dorothy. *Software Test Automation*. Addison-Wesley Longman Publishing Co., Inc., 1999.
6. KANER, Cem. *Exploratory Software Testing*. Addison-Wesley Professional, 2013.
7. BEIZER, Boris. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. Wiley, 1995.
8. MYERS, Glenford J.; SANDLER, Corey; BADGETT, Tom. *The Art of Software Testing*. 3rd. Wiley Publishing, 2011. ISBN 1118031962.
9. HEADSPIN. *The Testing Pyramid Simplified for One and All* [`https://www.headspin.io/blog/the-testing-pyramid-simplified-for-one-and-all`]. 2021. Accessed: 2024-04-13.
10. KOSMATOV, Nikolai. Constraint-based techniques for software testing. In: *Artificial intelligence applications for improved software engineering development: New prospects*. IGI Global, 2010, pp. 218–232.
11. PELESKA, Jan; HUANG, Wen-ling. Complete model-based equivalence class testing. *International Journal on Software Tools for Technology Transfer*. 2016, vol. 18, no. 3, pp. 265–283.
12. AHMAD, Farooq; QAISAR, Zahid Hussain. Scenario Based Functional Regression Testing Using Petri Net Models. In: *2013 12th International Conference on Machine Learning and Applications*. 2013, vol. 2, pp. 572–577. Available from DOI: `10.1109/ICMLA.2013.179`.

13. BRIAND, Lionel C.; LABICHE, Yvan. A UML-based approach to system testing. *Software and Systems Modeling*. 2003, vol. 1, no. 1, pp. 10–42.
14. CHEN, P. P. The Entity-Relationship Model: Toward a Unified View of Data. *ACM Transactions on Database Systems*. 1976, vol. 1, no. 1.
15. BUREŠ, Miroslav. Model-based Software Test Automation. *habilitation theses*. 2018.
16. EDUCBA. *UML Activity Diagram: Components and Symbols*. 2023. Available also from: `https://www.educba.com/uml-activity-diagram/`. Accessed: 2024-04-13.
17. ROTHERMEL, Gregg; HARROLD, Mary Jean. Selecting tests and identifying test coverage requirements for modified software. *Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*. 1994.
18. LEE, Jihyun; KANG, Sungwon; JUNG, Pilsu. Test coverage criteria for software product line testing: Systematic literature review. *Information and Software Technology*. 2020, vol. 122, pp. 106272. ISSN 0950-5849. Available from DOI: `https://doi.org/10.1016/j.infsof.2020.106272`.
19. TRETMANS, J. Model based testing with labelled transition systems. In: *Formal Methods and Testing: An Outcome of the FORTEST Network*. Springer, 2008, pp. 3–38.
20. KOELBL, A.; JACOBY, R.; JAIN, Himanshu; PIXLEY, C. Solver technology for system-level to RTL equivalence checking. In: 2009, pp. 196–201. Available from DOI: `10.1109/DATE.2009.5090657`.
21. KAZEMI, Ali; ROSTAMPOUR, Ali; HAGHIGHI, H.; ABBASI, Sahel. A conceptual cohesion metric for service oriented systems. *Journal of Web Engineering*. 2014, vol. 13, pp. 302–332.
22. JORGENSEN, Paul C. *Software Testing: a Craftsman's Approach*. Fourth. Auerbach Publications, 2013.
23. FOWLER, Martin. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. 3rd ed. Addison-Wesley Professional, 2004.
24. ROSENBERG, Doug; STEPHENS, Matt. *Use Case Driven Object Modeling with UML: A Practical Approach*. Addison-Wesley Professional, 1999.
25. SOMMERVILLE, Ian. *Software Engineering*. 9th ed. Addison-Wesley, 2011.
26. *OMG Unified Modeling Language (OMG UML), Version 2.5.1*. 2017. Available also from: `https://www.omg.org/spec/UML/2.5.1`.
27. WARMER, Jos; KLEPPE, Anneke. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Professional, 2003.

28. QUERALT, Anna; ARTALE, Alessandro; CALVANESE, Diego; TENIENTE, Ernest. OCL-lte: A dcidable (Yet Expressive) fragment of OCL. *CEUR Workshop Proceedings*. 2012, vol. 846.

29. CABOT, Jordi; GOGOLLA, Martin. Verification and Validation of UML and OCL Models in Practice. *Software and Systems Modeling*. 2012, vol. 11, no. 4, pp. 573–580.

30. BARUZZO, Andrea. *A unified framework for automated UML model analysis*. 2008. Available also from: `https://www.dimi.uniud.it/assets/dottorato/phd.thesis.baruzzo.pdf`. PhD thesis. University of Udine.

31. BUREŠ, Miroslav; KLÍMA, Matěj. Negative Constrained Path-based Testing. *System Testing IntelLigent Lab (STILL), Dept. of Computer Science, FEE, CTU in Prague*. 2023.

32. KANSOMKEAT, Supaporn; THIKET, Phachayanee; OFFUTT, Jeff. Generating test cases from UML activity diagrams using the Condition-Classification Tree Method. In: *2010 2nd International Conference on Software Technology and Engineering*. 2010, vol. 1, pp. V1-62-V1–66. Available from DOI: `10.1109/ICSTE.2010.5608913`.

33. NGO, Minh Ngoc; TAN, Hee Beng Kuan. Heuristics-based infeasible path detection for dynamic test data generation. *Information and Software Technology*. 2008, vol. 50, no. 7, pp. 641–655. ISSN 0950-5849. Available from DOI: `https://doi.org/10.1016/j.infsof.2007.06.006`.

34. GONG, Huiquan; ZHANG, Yuwei; XING, Ying; JIA, Wei. Detecting Interprocedural Infeasible Paths via Symbolic Propagation and Dataflow Analysis. In: 2019, pp. 282–285. Available from DOI: `10.1109/ICSESS47205.2019.9040767`.

35. BURES, Miroslav; CERNY, Tomas. Static Testing Using Different Types of CRUD Matrices. In: KIM, Kuinam; JOUKOV, Nikolai (eds.). *Information Science and Applications 2017*. Singapore: Springer Singapore, 2017, pp. 594–602.

36. BURES, Miroslav; RECHTBERGER, Vaclav. Dynamic Data Consistency Tests Using a CRUD Matrix as an Underlying Model. In: *Dynamic Data Consistency Tests Using a CRUD Matrix as an Underlying Model*. 2020, pp. 72–79. Available from DOI: `10.1145/3393822.3432333`.

37. ZHANG, Xiaoyan; LIU, Weihua; MA, Zengbin. Test Case Generation for Object-Oriented Software Based on Genetic Algorithms. In: *IEEE International Conference on Quality, Reliability, Risk, Maintenance, and Safety Engineering*. 2008.

38. ZHENG, Qing; YAO, Xin; MU, Yi; ZHANG, Qiang. An Improved Genetic Algorithm for Test Case Generation Based on Path Analysis. In: *International Conference on Computer Engineering and Technology*. 2010.

39. YAO, Xin; ZHENG, Qing; MU, Yi. A Survey of Genetic Algorithms in Test Generation for Structural Testing. *Journal of Software Engineering and Applications*. 2014.

40. ÇALIŞKAN, Cenk. New Algorithms for the Minimum Flow Problem. In: 2016.

41. FRÖHLICH, Peter; LINK, Johannes. Automated Test Case Generation from Dynamic Models. In: BERTINO, Elisa (ed.). *ECOOP 2000 — Object-Oriented Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 472–491. ISBN 978-3-540-45102-0.

42. ORACLE. *Java SE Development Kit 22 Documentation*. 2023. Available also from: `https://www.oracle.com/java/technologies/javase/22-relnotes.html`. Accessed: 2024-05-12.

43. SMART, Y.; FERGUSON, J. *Maven: The Complete Reference*. O'Reilly Media, 2021.

44. TEAM, JUnit. *JUnit 5 User Guide*. 2023. Available also from: `https://junit.org/junit5/docs/current/user-guide/`. Accessed: 2024-05-12.

45. QUEZADA SARMIENTO, Pablo; GUAMAN, Daniel; BARBA GUAMÁN, Luis Rodrigo; ENCISO, Liliana; CABRERA, Paola. SonarQube as a tool to identify software metrics and technical debt in the source code through static analysis. In: 2017.

46. SPRING. *Spring Boot Documentation*. 2023. Available also from: `https://docs.spring.io/spring-boot/docs/current/reference/html/`. Accessed: 2024-05-12.

Appendix B
List of Abbreviations

CCTM	Condition–Classification Tree Method
DAG	Directed Acyclic Graph
DCyT	Data Cycle Test
DFA	Data Flow Analysis
DFS	Depth–first search
MBT	Model-Based Testing
NCPT	Negative Constrained Path–based Testing
OCL	Object Constraint Language
STILL	System Testing IntelLigent Lab
STT	State Transition Testing
SUT	System Under Test