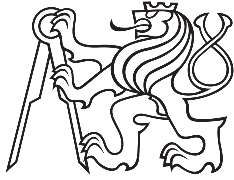
Master Thesis



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Computer Graphics Group

System for Evaluation of Model-based User Interface Testing Techniques Effectiveness

Bc. Zdeněk David

Supervisor: Ing. Feras Abdul Hadi Mustafa Daoud
Field of study: Open Informatics
Subfield: Human-Computer Interaction
May 2024

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **David Zden k** Personal ID number: **483781**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Graphics and Interaction**

Study program: **Open Informatics**

Specialisation: **Human-Computer Interaction**

II. Master’s thesis details

Master’s thesis title in English:

System for Evaluation of Model-based User Interface Testing Techniques Effectiveness

Master’s thesis title in Czech:

Systém pro evaluaci efektivity technik testování uživatelského rozhraní založených na modelu

Guidelines:

Design and implement a system for evaluating the effectiveness of user interface model-based testing techniques. The system will have the following components: a suitable system (or systems) to be tested for the case studies, for which source code will be available, automated tests as specified by the supervisor, a description of active, historical or artificial bugs in the software under test, and a component supporting the evaluation of the effectiveness of the automated tests. As part of the project validating the results, conduct three case studies using the functionality of the system and document these studies.

Bibliography / sources:

Utting, M., & Legeard, B. (2010). Practical model-based testing: a tools approach. Elsevier.
Ammann, P., & Offutt, J. (2016). Introduction to software testing. Cambridge University Press.
Kuhn, D. R., Kacker, R. N., & Lei, Y. (2013). Introduction to combinatorial testing. CRC press.

Name and workplace of master’s thesis supervisor:

Ing. Feras Abdul Hadi Mustafa Daoud Department of Computer Science FEE

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

Date of master’s thesis assignment: **07.02.2024** Deadline for master's thesis submission: **24.05.2024**

Assignment valid until: **21.09.2025**

Ing. Feras Abdul Hadi Mustafa Daoud
Supervisor’s signature

Head of department’s signature

prof. Mgr. Petr Páta, Ph.D.
Dean’s signature

III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work. The student must produce his 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

© VUT v Praze, Design: VUT v Praze, VIC

Acknowledgements

I would like to express my deepest appreciation to my supervisor Ing. Feras Abdul Hadi Mustafa Daoud and doc. Ing. Miroslav Bureš, Ph.D. for so much patience, insightful feedback, and unwavering support throughout the making of this thesis.

Equally, my heartfelt thanks go to my fiancée and my family for their endless love, support, and patience during my studies.

Declaration

I hereby declare I have written this master's thesis independently and quoted all the sources of information used in accordance with methodological instructions on ethical principles for writing an academic thesis. Moreover, I state that this thesis has neither been submitted nor accepted for any other degree.

In Prague, May 2024

. . . .

Bc. Zdeněk David

Abstract

This thesis describes the design and implementation of a system developed to assess the effectiveness of model-based user interface testing methodologies, focused on Combinatorial Interaction Testing - CIT. The purpose of this system is to improve software testing by enhancing defect detection rates, test coverage as well as efficiency compared to traditional testing methods. As software applications grow in complexity, traditional testing methods become less effective, necessitating innovative approaches to ensure reliability and quality.

In order to do this, we have created a methodology for conducting case studies, an environment for testing the cases and a system for evaluating the effectiveness of these tests. Three case studies were done utilizing CIT in order to evaluate its practical implementation. The experiments involved setting up controlled environments, generating test cases both manually and using CIT, executing these case studies and analyzing the results.

The findings indicate that CIT improves the effectiveness of defect detection and test coverage while reducing overall testing time. This means that model-based testing techniques, such as CIT in particular, offer a more efficient approach to software quality assurance, promoting their wider implementation in software testing processes.

Keywords: Model-based Testing, User interface testing, Combinatorial Interaction Testing, Software quality assurance, Test coverage, Defect detection, Automated testing

Supervisor: Ing. Feras Abdul Hadi Mustafa Daoud

Abstrakt

Tato práce představuje návrh a implementaci systému pro hodnocení účinnosti technik testování uživatelského rozhraní založených na modelech, konkrétně pomocí metody kombinatorického testování (angl. Combinatorial Interaction Testing - CIT). Cílem je zlepšit testování softwaru zvýšením míry odhalení chyb, pokrytí testů a efektivity ve srovnání s tradičními metodami. S rostoucí složitostí softwarových aplikací se tradiční metody testování stávají méně efektivními, což vyžaduje inovativní přístupy k zajištění spolehlivosti a kvality.

Za tímto účelem jsme vytvořili metodiku pro provádění případových studií, prostředí pro testování případů a systém pro vyhodnocování účinnosti těchto testů. Byly provedeny tři případové studie využívající CIT k vyhodnocení praktické implementace této metody. Experimenty zahrnovaly vytvoření kontrolovaného prostředí, vytvoření testovacích případů ručně i pomocí CIT, provedení těchto případových studií a analýzu výsledků.

Zjištění ukazují, že CIT zlepšuje detekci chyb a pokrytí testů a zároveň zkracuje celkovou dobu testování. To naznačuje, že techniky testování založené na modelech, zejména CIT, poskytují efektivnější přístup k zajištění kvality softwaru, což podporuje jejich širší implementaci do procesů testování softwaru.

Klíčová slova: Testování založené na modelu, testování uživatelského rozhraní, metody kombinatorického testování, zajištění kvality software, pokrytí testů, detekce defektů, automatizované testování

Překlad názvu: Systém pro evaluaci efektivity technik testování uživatelského rozhraní založených na modelu

Contents

1 Introduction **1**
1.1 Background and context of model-based user interface testing . . 1
1.2 Importance of evaluating the effectiveness of model-based testing techniques 1
1.3 Research objectives and scope of the thesis 2
1.4 Overview of the thesis structure . 2

2 Theoretical Framework of Software testing **3**
2.1 Introduction to Software Testing . 3
2.1.1 White box testing 3
2.1.2 Black box testing 3
2.1.3 Theoretical Models of Software Quality and Testing 3
2.2 Concepts of Testability, Factors of Testability in Software 5
2.2.1 Testability Factors 5
2.2.2 Requirements of Software Testability 6
2.2.3 Improving Software Testability 7
2.3 Test Coverage Criteria and Adequacy Models 7
2.3.1 Code coverage 8
2.3.2 Compatibility coverage 8
2.3.3 Product coverage 8
2.3.4 Requirements coverage 8
2.4 Principal Test Levels 9
2.4.1 Unit testing 9
2.4.2 Module testing 9
2.4.3 Integration testing 10
2.4.4 Functional and system testing 10
2.4.5 User Acceptance Testing 10
2.4.6 Beta testing 10
2.4.7 Regression testing 10
2.5 Combinatorial Interaction Testing 11
2.6 Test Planning and Management 12
2.6.1 Key components of a test plan 12
2.6.2 Designing test cases and managing test data 12
2.7 Evaluating Test Automation Return on Investment 13

3 Literature Review **15**
3.1 Definition and principles of model-based user interface testing . 15
3.2 Techniques and approaches for model-based testing in UI design . . 15
3.3 Evaluation criteria and metrics for assessing the effectiveness of testing techniques 16
3.4 Principles of Model-Based Testing 18
3.5 Previous studies and research on model-based testing effectiveness . . 19

4 Evaluating the effectiveness of model-based testing techniques **21**
4.1 Research design and approach for evaluating model-based UI testing effectiveness 21
4.1.1 Objective 21
4.1.2 Methodology 21
4.1.3 Evaluation Criteria 22
4.2 Selection of testing tools and frameworks 22
4.2.1 ACTS 3.2 22
4.2.2 Cypress 22
4.2.3 Git/Github 22
4.2.4 Google Sheets as a Test Management Framework 23
4.3 Design of experiments and case studies 23
4.3.1 Impact Analysis of Combinatorial Interaction Testing on Systematic Software Validation 23
4.3.2 Combinatorial Interaction Testing in Usability Studies 29
4.3.3 Effectiveness of Combinatorial Interaction Testing in Test Automation 35
4.3.4 Threats to Validity 40

5 Results **43**
5.1 Results of Specific Case Studies . 43
5.2 Summary of Key Results 45
5.3 Implications of Results 46
5.4 Limitations of the Study 47
5.5 Recommendations for Future Research 47

6 Conclusion **49**

Bibliography **51**

A Acronyms **57**

B Used Software **59**

Figures

2.1 Diagram of the V-Model in Software Testing 5

4.1 Flowchart of the experiment method 25
4.2 OpenCart - Subscription Plan Form 27
4.3 OpenCart - Attribute Form 28
4.4 Odoo - Dashboard 31
4.5 Odoo - Employee Form 32
4.6 Odoo - Product Form 33
4.7 Odoo - Sale Quotation Form . . . 34
4.8 JTrac - Login Form 36
4.9 JTrac - Creating New Item Form 36

Tables

2.1 Mapping Testing Levels to SDLC Stages and Code Accessibility 9
2.2 Test case example 13

4.1 Number of values of each parameter in Adding a Subscription Plan Form 28
4.2 Number of values of each parameter in Adding an Attribute Form 28
4.3 CIT Parameters and Values for Subscription Plans 29
4.4 Participant Observations Collection Sheet for Usability Study 30
4.5 CIT Parameters and Values for Usability Testing 35
4.6 Number of values of each parameter in Creating New Item Form 38

5.1 Comprehensive Comparison of Testing Metrics with and without CIT 43
5.2 Average Time to Design Test Scenarios 44
5.3 Average Task Completion Time . 44
5.4 Comparison of count of test cases, count of defects detected and time spent between 'without CIT' and '2-way' and 'Mixed' methods 46

Chapter 1

Introduction

1.1 Background and context of model-based user interface testing

The complexity of software applications is growing in the current digital era, driven by the need for more sophisticated features and smooth user experience (UX). This increase in complexity poses significant challenges to traditional software testing methods, making them very time-consuming. This leads to poor test coverage of many systems.

Model-Based Testing (MBT) has become an example of innovation in this field, reducing overall testing time. Additionally, by using various test selection criteria, one may use the same model to build a variety of test suites [1]. An industrial case study revealed that although MBT required more preparation, it was more methodical and effective, resulting in more even test coverage and revealing a greater number and severity of functional issues than manual testing, which was however faster to initiate [2].

Despite this, there is still a need for more research to understand the effectiveness of some newer Model-Based Testing techniques in practical, meaningful scenarios.

1.2 Importance of evaluating the effectiveness of model-based testing techniques

Since Model-Based Testing methods have the potential to improve software quality assurance, it is important to evaluate their effectiveness. Model-Based Testing methods provide an organized way to find errors, guarantee thorough coverage, and boost the effectiveness of the testing process by automating the creation and execution of test cases from abstract models. This kind of assessment is important for determining the benefits and drawbacks of Model-Based Testing in a practical context, which would help developers and testers choose the best approach for their work, and help in the creation of more dependable, user-focused software applications.

Current empirical research emphasizes how important it is to implement

Model-Based Testing to manage the complexity and improve the quality of web applications. Many empirical studies point to the importance of effective testing techniques in order to guarantee software dependability and adjust to quick modifications [3–5].

1.3 Research objectives and scope of the thesis

The main objective of this thesis is to design and implement a system for evaluating the effectiveness of chosen newer Model-Based Testing techniques. The purpose of the system is to assess how the application of a certain Model-Based Testing technique improves and speeds up the testing process in comparison to a baseline testing process in which no particular techniques are used while generating test cases.

1.4 Overview of the thesis structure

This thesis is organized into five main chapters.

The *Literature Review* chapter first reviews existing literature on model-based user interface testing, including the definition and principles of Model-Based Testing, then covers techniques and approaches for UI design, evaluation criteria and metrics for assessing testing effectiveness and briefly explores previous studies and research on the subject.

The chapter on the *Theoretical Framework of Software Testing* provides an introduction to the basic concepts in software testing. It covers topics such as white box and black box testing, theoretical models of software quality, and the principles of Model-Based Testing and model checking. The chapter also covers test levels, specific testing techniques used in this thesis and test planning and management.

The *Evaluating the Effectiveness of Model-Based Testing Techniques* chapter outlines the research design and also methodology used to evaluate the effectiveness of Combinatorial Interaction Testing (CIT). The chapter covers the process of selecting testing tools and frameworks, design of the experiments and setup of three case studies.

The *Findings* chapter presents the results of the case studies, summarizing key findings and discussing their implications. Secondly, it also addresses the limitations of the study and provides recommendations for future research.

The chapter *Conclusion* provides a concise summary of the primary contributions of the thesis. It also evaluates the overall influence of the research on the area of software testing.

Chapter 2

Theoretical Framework of Software testing

2.1 Introduction to Software Testing

Software testing ensures the reliability, functionality and performance of software applications and is a crucial component of the software development lifecycle. In order to assess one or more qualities, software/system components are executed manually or preferably automatically to lower its cost dramatically, decrease human error, and simplify regression testing using tools. [6] Finding errors, gaps, or missing requirements in comparison to the real requirements is the main objective of software testing.

2.1.1 White box testing

White box testing *(also called structural testing, clear box testing and glass box testing)* is a software testing approach that verifies the internal structure and functionality of a system. This approach requires access to the system's source code and knowledge of the underlying programming language. White box testing is used for unit, integration and regression testing [7], which are covered later in principal testing levels in section 2.4.

2.1.2 Black box testing

Black box testing *(also known as functional testing and behavioral testing)* is a software testing approach where the functional requirements of the software are tested without knowledge of the internal code structure and implementation specifics. It is present in all of the principal testing levels (covered in section 2.4) apart from unit and module testing [8].

2.1.3 Theoretical Models of Software Quality and Testing

Theoretical models of software quality and testing give critical guidance for ensuring that software solutions fulfill the highest quality requirements. The ISO/IEC 25010 standard defines the models for both software product quality and software quality in practice, as well as practical instructions on how to

utilize the quality models [9], while the V-Model applies this concept to a realistic, step-by-step development and testing approach.

ISO/IEC 25010

The ISO/IEC 25010 standard, part of the Systems and Software Quality Requirements and Evaluation (SQuaRE) framework, specifies a comprehensive model for evaluating software quality. This model divides software quality into eight main characteristics, each with their own set of sub-characteristics. These characteristics ensure that software not only fulfills its intended functionality, but also adheres to quality parameters that improve user satisfaction, performance efficiency, compatibility and more [10, 11].

- Functional Suitability
  This is about the software's ability to provide functions that meet stated and implied needs under specific conditions, focusing on the completeness, correctness, and appropriateness of those functions.
- Performance Efficiency
  It evaluates the performance relative to the amount of resources used under stated conditions, looking at time behavior, resource utilization, and capacity.
- Compatibility
  This characteristic assesses the software's ability to co-exist and exchange information with other products, systems, or components in a shared environment, emphasizing co-existence and interoperability.
- Usability
  It covers the software's ease of use and attractiveness, including user interface aesthetics, learnability, and operability.
- Reliability
  This focuses on the software's capability to maintain a level of performance under stated conditions for a stated time period, assessing aspects like maturity, fault tolerance and recoverability.
- Security
  Security evaluates the software's ability to protect information and data, preserving confidentiality, integrity, and authenticity.
- Maintainability
  This characteristic, which focuses on modifiability, testability, and analysability, looks at the ease with which the software can be modified to correct defects, meet new requirements, or make future maintenance easier.
- Portability
  It assesses the ease with which the software can be transferred from one environment to another.

V-Model

The V-model, named for its characteristic V shape, is an extension of the waterfall model used in software (and hardware) development. This model emphasizes the value of validation at every level of the process by directly connecting each development phase with a matching testing stage [12].

This diagram in figure 2.1 illustrates the parallel relationship between each development stage and its associated testing phase in the V-Model process.

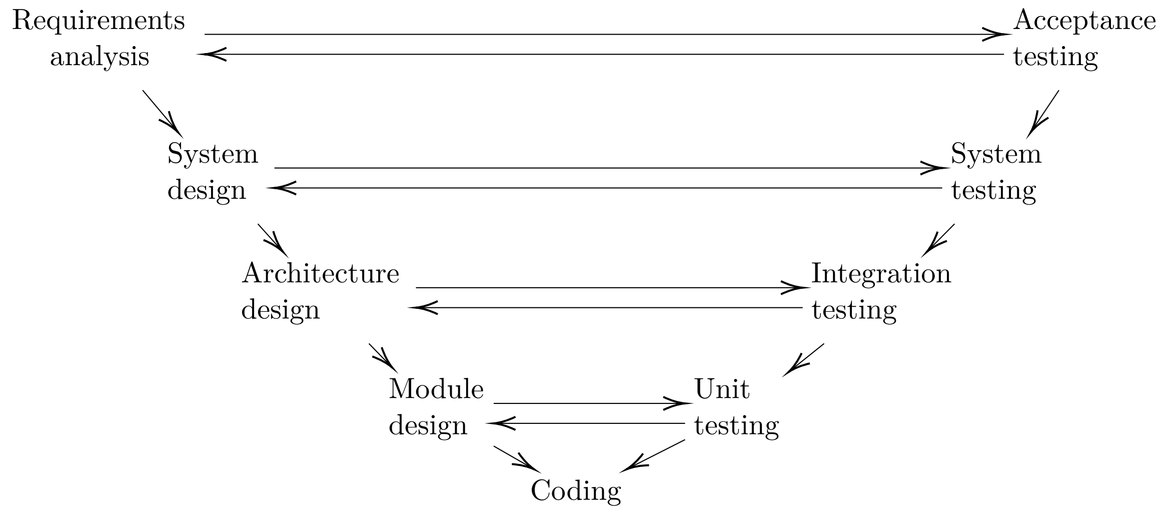


Figure 2.1: Diagram of the V-Model in Software Testing

2.2 Concepts of Testability, Factors of Testability in Software

Software testability is a measure of how effectively a software system or component can be tested. The ease and efficiency of conducting tests are determined by several influencing factors. Gaining an in-depth knowledge of testability and implementing measures to improve it is important in order to enhance the quality of software and minimize the expenses and labor associated with testing.

2.2.1 Testability Factors

There are several key factors that contribute to the testability of software. These factors can help in assessing and improving the ease with which software can be tested.

Observability

Observability refers to the ability to observe the user inputs into the system during testing. A system with high observability provides clear information

about its behavior, making it easier to identify defects. This includes logging, reporting and debugging capabilities that expose the system's internal operations to testers [13].

Controllability

Controllability is the degree to which the state of the software modules can be controlled by testers in isolation. High controllability means that testers can easily manipulate the software to achieve desired test conditions and inputs. This includes the ability to set up and modify the software's environment and configurations to facilitate testing [13].

Simplicity

Simplicity refers to the complexity of the software. Simpler software has fewer complexities, making it easier to test. This includes well-defined interfaces, modular design and high coding standarts. Simplified software reduces the effort needed for testers and devs to test the system and minimizes the potential for introducing errors during testing [14].

Stability

Stability is the extent to which the software undergoes changes. Software that is stable and has fewer changes is easier to test because the test cases remain valid for longer periods. Frequent changes in the software can lead to outdated test cases and require constant updates to the testing process [14].

Availability

Availability refers to the extent to which all necessary objects and entities for testing are readily available at every stage of development. This includes the availability of test environments, test data, and the software itself. High availability ensures that testers have continuous access to the resources needed for conducting tests without significant downtime or interruptions [13].

2.2.2 Requirements of Software Testability

Software testability requires the fulfillment of specific conditions and attributes to ensure testability. These requirements include:

- **Test support capabilities:** It is essential to ensure that every team member, test interface, and scenario has access to the entry point for testing drivers and root, and that it remains intact accordingly.
- **Module capabilities:** A modular architecture allows testing of individual components independently, making it easier to isolate and fix defects.

- **Defect disclosure capabilities:** The frequency of system errors should be reduced to prevent them from impeding software testing.

  The requirement document must be testable, accurate, correct, concise, unambiguous, not contradicting other requirements, prioritize-based, and domain-based to ensure easy implementation and avoid challenges in changing requirements.
- **Observation capabilities:** The software should have mechanisms to monitor user inputs, output, and factors influencing it, including static, dynamic, and functional analysis capabilities.

These requirements ensure that the software is prepared for thorough testing and can be evaluated effectively for quality assurance [13].

2.2.3 Improving Software Testability

Improving software testability involves adopting strategies that enhance the factors mentioned above. Some methods to improve testability include:

- **Refactoring Code:** Regularly refactoring the code to reduce complexity and improve readability can significantly enhance testability.
- **Enhancing Observability:** Implementing robust logging and monitoring mechanisms helps in tracking the software's behavior and diagnosing issues within the SUT.
- **Improving Documentation:** Maintaining up-to-date and comprehensive documentation aids testers in understanding the system and creating accurate test cases.
- **Integrating Tools for Testers:** Integrating automated testing tools and frameworks into the development process ensures that tests can be executed efficiently and frequently.
- **Designing for Testability:** Incorporating testability considerations into the software design phase, such as using modular architectures and well-defined interfaces, can make the software inherently more testable.

By focusing on these areas, organizations can enhance the testability of their software, leading to more effective testing processes and higher quality software products [14].

2.3 Test Coverage Criteria and Adequacy Models

We can employ various test coverage criteria to evaluate the completeness of a set of tests.

2.3.1 Code coverage

The amount of code that has been subjected to automated testing is measured by code coverage. This category generally falls under white box testing.

We may use methods such as *statement coverage, branch coverage* and *path coverage*, each of which focuses on a different aspect, to find untested parts of the codebase [15].

Statement coverage

Statement coverage is a testing criterion that validates the sufficiency of a given set of tests ensuring that each statement in an application is executed at least once [16].

Branch coverage

Branch coverage is a testing metric that assesses the proportion of code's executed decision points, or if-else statements, to make sure a wide variety of scenarios are tested [17].

Branch coverage shows the coverage of the basic logic behind the application's decision-making process by guaranteeing that each branch—true or false—of each decision point is run at least once [17].

Path coverage

Path coverage is a similar testing method, which aims to test every branch combination and possible execution path in the program. Although this approach aims for a comprehensive testing procedure, as software complexity increases, it may significantly increase the overall number of tests required [17].

2.3.2 Compatibility coverage

Compatibility coverage employs a wide range of testing techniques, including mobile, hardware, browser and network testing to ensure that the software works across different browsers reliably, operating systems, and devices [18].

2.3.3 Product coverage

Product coverage is the process of evaluating a software's performance from a product perspective, concentrating on important areas by developing checklists, defining criteria, and putting test automation into practice [18].

2.3.4 Requirements coverage

By tracking the implemented requirements and testing them through various tests, requirements coverage assesses whether a software solution satisfies all specified functionalities and client requirements [18].

2.4 Principal Test Levels

There is a testing level for every unique software development activity: Unit testing assesses how well the code is implemented, module testing compares the completed design of individual modules in isolation, integration testing looks at how well subsystems are integrated, system testing confirms the architectural design, acceptance testing makes sure the software satisfies requirements and regression testing confirms that no unintended changes were introduced with a software update [6].

We will also shortly explain beta testing since it can be considered a test level [8].

Table 2.1 methodically presents the assignment of each testing level to distinct stages of the software development life cycle (SDLC), together with the code access strategy (white box vs. black box). This demonstrates how testing approaches are strategically integrated throughout the development process.

Test level	Stage of SDLC	Access to code
Unit testing	Development	White box
Module testing	Development	White box
Integration testing	Testing	White box & black box
Functional and system testing	Testing	Black box
User Acceptance Testing	Testing	Black box
Beta testing	Deployment	Black box
Regression testing	Testing & Maintenance	White box & black box

Table 2.1: Mapping Testing Levels to SDLC Stages and Code Accessibility

2.4.1 Unit testing

Unit testing evaluates a system's smallest components, such as functions, in isolation from the rest of the application. Test cases are intended to cover all potential pathways, including loops, forks, and particular lines of code, to ensure that functionality functions as anticipated. Concurrent unit testing with development promotes clean code and early discovery of errors, leading to a scalable and sustainable method [19].

2.4.2 Module testing

Module testing, like unit testing, requires a thorough understanding of the software's internals. The primary goal of module testing is to detect defects in the module's interface and data flow interactions, ensuring that the module meets its specifications and performs as intended. By testing modules in isolation, developers can identify and address specific areas of failure early in the development process. This phase broadens the scope of white box testing while maintaining a focus on the code's logic and structure [20].

2.4.3 Integration testing

Integration testing examines the interaction of several modules or components of an application. This testing phase attempts to detect defects in the interfaces and interactions of integrated components. The primary purpose is to identify any inconsistencies between merged units that may not be obvious during previous testing [21].

Unlike module or unit testing, which analyzes individual components in isolation, integration testing investigates the data flow and communication between modules to ensure they function properly.

2.4.4 Functional and system testing

Functional and system testing tries to ensure that the application meets high-level design criteria and specifications. Functional testing determines if the program executes its functionality specified in the requirements specification successfully, while system testing assesses the software's behavior in different environments to ensure compatibility and performance per user expectations. This testing phase focuses on outputs in response to inputs, without concern for the application's internal structure [8].

2.4.5 User Acceptance Testing

Acceptance testing is defined as formal testing in which the software is evaluated to see whether it fits the acceptance criteria and allows the customer to decide whether to accept the system. In this phase, it is verified that the software meets the customer's specifications and requirements. Acceptance tests are often set by the customer and carried out based on their expectations of the software's functionality [8].

2.4.6 Beta testing

Beta testing involves providing the yet unreleased system to selected people outside of the development team, often for free, with the condition that they report any identified issues back. This phase often results in the detection of unexpected faults, as beta testers may use the product in unforeseen ways and in diverse environments. The benefits of beta testing include a broad search for faults at a low cost because beta testers are often unpaid. However, a lack of rigorous testing and potentially low-quality mistake reports might make it difficult to assess feedback [8].

2.4.7 Regression testing

Testing to verify that recent modifications haven't added new defects to previously functional software is known as regression testing. The modifications include adding new features or requirements, bug fixes, performance optimizations, patch updates, UI modifications, configuration changes and

integrating third-party systems. Regression testing is a good candidate for automation since it is repetitive and acts as a frequent check to ensure the integrity of the product [22].

Regression test automation is frequently done with tools like testRigor[1], Sahi Pro[2] and Selenium[3], which offer support for various testing needs across multiple platforms and environments [23].

2.5 Combinatorial Interaction Testing

This section explores the technique present in all of the case studies later presented in chapter 4.3.

Combinatorial interaction testing (CIT) is a software testing approach that methodically analyzes combinations of input parameters to efficiently detect problems resulting from unexpected interactions among them. The fundamental concept underlying CIT is that it is unnecessary to test every conceivable combination of parameter values. Instead, by selecting strategic combinations that optimize the coverage of interactions, the number of test cases may be greatly reduced while still achieving a high level of test effectiveness [24].

The technique relies on the use of *covering arrays*, which are mathematical constructs that ensure each combination of parameters up to a certain length is tested at least once. This approach is rooted in the design of experiments (DoE) and has been adapted to address the unique challenges in software testing [25].

Advantages of Combinatorial Interaction Testing:

- **Reduced Test Suite Size:** By using algorithms that generate minimal covering arrays, CIT can significantly reduce the number of test configurations compared to exhaustive testing.
- **Improved Fault Detection:** Empirical studies suggest that most defects are caused by interactions between a small number of parameters. CIT targets these interactions directly, often leading to more effective fault detection.
- **Efficiency:** Due to the progress in algorithms for producing covering arrays, CIT has become a feasible and effective method for evaluating complex systems that have many parameters and configurations.

Practical Applications: CIT has been successfully applied in some domains, including software, hardware and system integration testing. The method is particularly valuable in environments where parameters and configurations are numerous and complex, making traditional testing approaches both challenging and cost-prohibitive.

[1] testRigor. Available at `https://testrigor.com/`
[2] Sahi Pro. Available at `https://www.sahipro.com/`
[3] Selenium. Available at `https://www.selenium.dev/`

2.6 Test Planning and Management

Test plan is a document describing the scope, approach, resources, and schedule of intended test activities. It identifies test items, the features to be tested, the testing tasks, who will do each task, and any risks requiring contingency planning. [26]

Test management is a complete process that involves the proactive monitoring of testing operations throughout software development. This consists of test planning, organization, coordination, and oversight to guarantee that high-quality, dependable software is delivered on schedule [27].

2.6.1 Key components of a test plan

A test plan describes the approach that will be used to verify that a software product satisfies its requirements, including design standards. This document contains a description of the testing scope, timetable, required resources, environments, tools, risk management strategies, defect management procedures, and exit criteria. It acts as a road map for the testing procedure, guiding teams through every important phase to guarantee thorough assessment and quality control of the program [28].

The scope, schedule, resources, environment, tools, and strategies for managing defects, risks, and exits are explained in the following list.

- Scope
  Specifies what will be tested and the features or areas that will be excluded from testing.
- Schedule
  Lays out the timelines for every stage of testing.
- Resource Allocation
  Describes how employees and equipment are distributed.
- Environment
  Specifies the hardware and software setup for testing.
- Tools
  Lists the software and tools required for testing.
- Defect Management
  Describes tracking and resolving of issues.
- Risk Management
  Identifies possible risks and suggests mitigation strategies.
- Exit Parameters
  Sets criteria for concluding the testing phase.

2.6.2 Designing test cases and managing test data

To assess whether an application's functionality is performing properly, test cases are created by defining the input, action, or event, as well as the intended

result. The primary purpose is to discover scenarios in which the application may not perform as planned. This involves:

- **Test Case Identification:** Identify scenarios based on needs and specifications.
- **Test Data Selection:** Choosing or generating data to utilize during testing to ensure that the application is tested properly.
- **Defining Expected Results:** Define the expected outcome of the test and compare it to the actual result.
- **Test Case Documentation:** Documenting the test case in sufficient detail for others to perform it.

The test case documentation is an important part of the testing process, and properly managing the test cases will improve and simplify the testing process. Previously, companies utilized spreadsheet applications like Microsoft Excel for test management. Although using a test management framework is more costly, automation frameworks can decrease software deployment times by 75% while increasing test coverage by 35% [29].

Usual structure of a documented test case will contain fields similar to these: *Test Case ID, Test Scenario, Test Steps, Prerequisites, Browser, Test Data, Expected/Intended Results, Actual Results* and *Test Status – Pass/Fail* [30]. Table 2.2 shows a simple test case based on a specific scenario.

Test Case ID	TS-75
Test Scenario	ValidLogin
Test Steps	Visit the login page Input 'admin' into the login field Input 'admin' into the password field Click the submit button
Prerequisites	A registered user with login and password
Browser	Brave Version 1.63.169 Chromium: 122.0.6261.111
Test Data	login: admin password: admin
Expected/Intended Results	User logs in, page redirects to dashboard
Actual Results	Successful login, page redirects
Test Status - Pass/Fail	T

Table 2.2: Test case example

2.7 Evaluating Test Automation Return on Investment

Return on Investment (ROI) is a statistic that quantifies the potential return on investment achievable by integrating an automation approach into your quality assurance processes. By calculating Test Automation's ROI, stakeholders may be more confident that the investment will be beneficial in the long term.

We can generally calculate ROI using the formula below:

$$\text{ROI} = \frac{\text{benefits} - \text{costs}}{\text{costs}} * 100\% \text{ [31]}$$

Of course, *benefits* and *costs* are very vague terms, but often, they will be defined similar to this:

- **Benefits:** This term includes *cost savings*(reduction in manual testing hours, decreased time to market and lower defect resolution costs post-release) and *quality improvements*(higher defect detection rates, improved test coverage and increased reliability).
- **Costs:** Will include things such as *initial investment*(tool licensing, hardware and training costs) and *operational costs*(maintenance, updates and support costs).

After nearly six months of using tools for automation, the average ROI is usually around 250% [32].

Chapter 3

Literature Review

3.1 Definition and principles of model-based user interface testing

Model-Based Testing automates the creation of black-box tests by using a model that represents the expected behavior of the system under test (SUT). Unlike traditional black-box testing, which involves manually crafting tests from requirements documentation, Model-Based Testing employs tools to generate tests from this behavior model automatically [1].

We cover the specific generation of the models in chapter 3.2.

3.2 Techniques and approaches for model-based testing in UI design

This part examines approaches to software application user interface (UI) testing. GUIs are using MBT more and more. In MBT, user interactions are described as sequences from which test cases are generated [1].

A. M. Memon has conducted extensive research [33–35] on Model-Based Testing for interactive systems and he proposes these techniques to represent the model:

- *State machine*
  A state machine describes the states, transitions and actions of a system depending on events or conditions. The system can only be in one state at a time and it transitions to other states based on events and conditions.
- *Workflow*
  Workflows can be used to model the processes and interactions within a system, which can help us state what must be tested within the system in order to meet business requirements.
- *Pre- and Postcondition*
  Pre- and post conditions are statements that describe the system state before and after one function or operation. Preconditions state

conditions that must be fulfilled before the operation is performed, while postconditions depict the expected outcome of an operation if it has been performed. They are used to ensure that a particular software application behaves in a particular manner in a given scenario.

- *Event sequence*
  An event sequence is a systematic collection of critical system events used in testing to uncover any potential problems or decrease in performance during regular operations or during intense stress tests. The goal of efficient event sequence creation is to completely cover multiple interactions while avoiding redundancy.
- *Probabilistic*
  Probabilistic models utilize stochasticity and probability to forecast distinct events and behaviors of a system. They are beneficial for simulating systems with inherent uncertainties and for conducting tests that include different probability possibilities.
- *Combinatorial*
  Combinatorial testing refers to a systematic approach of combining input factors to create test cases that cover all the possible combinations of parameter up to certain level. This approach is greatly beneficial when looking for faults caused by interactions with multiple inputs.
- *Hierarchical*
  In hierarchical models the objects are arranged within the framework of a cascade of levels: higher levels represent more general concepts while lower levels contain detailed implementation. In software testing, the system under test (SUT) are considered as being composed of subsystems, thus, through the hierarchy, the systems are gradually simplified so that testing can be performed on all or selected sub–systems as chosen by the testers.

It has been proposed to use MBT derived from research records, sketches, design files, prototypes, reports, and other documentation that design teams produce during a project as a step before human-based usability testing [36], since it is known to be expensive and time-consuming and works better on systems that have previously undergone extensive testing.

There is also an effort to come up with approaches that are able to model natural interactions and the ability to use new input methods, including multi-touch displays. For example, this study [37] proposes the Malai GUI specification language to overcome these restrictions and identify defects in interactive systems.

3.3 Evaluation criteria and metrics for assessing the effectiveness of testing techniques

The book "Software Testing Fundamentals: Methods and Metrics" [38] cites several key metrics relevant to our study:

- The Time Required to Run a Test

  The time required to complete planned tests is a key statistic in software testing and is used to estimate the length of a test effort. In addition to considering the possibility of repeated attempts to accomplish successful and reliable test execution, this estimation must account for test setup and cleanup times, either as part of the test duration or separately.

- The Cost of Testing

  Testing costs include tester salaries, used systems, software and tools, which together sum up to the price of running a test or a test suite.

  The author states that while it is easy to calculate using accurate project metrics, it is difficult to compare the cost of testing to the cost of *not* testing.

- Test sample units

  Although there isn't a single, accepted method for calculating a test's size, it is nevertheless useful to recognize and quantify the various types of tests to assess the effectiveness of testing. The techniques for estimating and tracking tests — which are defined by parameters like importance/priority, quantity and type, are covered in the following list.

  - Importance/priority

    Prioritizing tests using ranking criteria, making sure that the most important tests are identified and run first to reduce the greatest risks related to the functioning and dependability of the product.

  - Quantity

    The overall number of tests planned for a project is important because it establishes the basis for organizing and allocating resources. Comprehensive test coverage is also important because it guarantees that all important software features are covered in detail.

  - Type

    To provide a thorough testing effort, there are different types of tests, such as path tests, data tests, module tests, user scenarios, installation tests, environment tests, and configuration tests. Each test type covers a different aspect of the product.

- Bug sample units

  While there is no standard in place for measuring bugs, according to the author, there are several sample units we can measure regarding bugs: Severity, quantity, type, duration, distribution and cost to find and fix [38]. They are briefly explained in the following list.

  - Severity

    Since there is no industry-wide standard for severity, many systems for assessing severity are common to evaluate the effect of a defect.

  - Quantity

This refers to the total number of defects, errors or issues identified in the software during various phases of the testing process.

- Type

  Bugs cover a broad range of issues, from misunderstood interfaces to catastrophic failures, and are classified according to local rules as well as specific criteria like reproducibility and fixability.
- Distribution

  The system modules with the highest number of issues, or the most serious bugs, should be the focus of our efforts.
- Cost to find and fix

  This metric, which can be derived from other metrics, serves as another source of information that we can use. It can usually be expressed in units like currency per issue or person-time per issue.

- Test Coverage

  Test coverage is a metric that evaluates the amount of software that is tested to guarantee thorough testing. Test coverage is the part that was actually tested out of a set of items that could be tested. This could be expressed by the following formula:

  $$\text{test coverage} = \frac{\text{test conducted}}{\text{total tests}} * 100\% \text{ [38]}$$

- Test Effectiveness

  Test effectiveness evaluates how well tests identify errors; this can be done by focusing on the most beneficial tests with the least amount of time and money, without having to cover every possible scenario. It is determined by calculating the proportion of all bugs that the tests detect, with an emphasis on identifying the most important ones. These crucial tests offer a significant return on testing investment and are particularly helpful for long-term monitoring and problem-solving in real systems. It can be calculated using the simplified formula below:

  $$\text{test effectiveness} = \frac{\text{bugs found in test}}{\text{total bugs found}} * 100\% \text{ [38]}$$

As the book is quite old, it does not mention *Automation Coverage (also known as Test Automation statistic)* as some newer literature [39, 40]. This metric simply shows what proportion of manual test cases are automated and can be calculated as follows:

$$\text{automation coverage} = \frac{\text{total no. of test cases automated}}{\text{total no. of test cases}} * 100\% \text{ [39]}$$

3.4 Principles of Model-Based Testing

In MBT, the behavior of the program being tested during runtime is compared to predictions generated by a model. A model is a description of the behavior of a system. In this context, these models can be created from a domain model, an environment model, a behavior model or from abstract tests to generate input data, test cases or test scripts [1].

To generate input data, the model provides information about the domains of the input values. We might then use techniques such as Combinatorial Interaction Testing (CIT)(also known as *t-way* or *t-wise* testing where t indicates interaction strenght [41]) to cleverly generate a minimum number of tests that go through each possible combination of input values.

Models describing use patterns or data value frequency that characterize the SUT's expected environment can be used to generate test cases. It is difficult to tell if a test case succeeds or fails since these models can generate SUT call sequences, but cannot anticipate results, so often we can only tell if the system crashes or not [1].

Then there's the option of developing executable test cases containing oracle data[1] including predicted outputs or comparisons of the actual outputs to determine correctness. For the model to do this, it must include the SUT's behavior regarding its inputs and outputs. Because it covers input selection, operation sequencing, and outcome verification, it is harder to implement, but it's the only option that can automate the entire test design process [1].

Another principle is to use an abstract test description (e.g. UML sequence diagram) and turn it into a detailed, executable test script.

3.5 Previous studies and research on model-based testing effectiveness

The Practical Model-Based Testing book [1] covers these experiences of real companies that have incorporated MBT in their applications:

1. Model-Based Testing at IBM

   IBM used a model-based test generator called GOTCHA-TCBeans for two case studies. In the first case study, a 17% reduction in cost was achieved by using MBT, which took 10 person-months and discovered 2 more issues in addition to the 18 revealed by a manually designed test suite (postmortem analysis showed 15/18 earlier defects would have been also detected) [43].

   In the second case study, which evaluated a Java garbage collector, MBT discovered four more defects and raised statement coverage to 83% while requiring about half the time of more conventional techniques [43].

2. Model-Based Testing at Microsoft

   Microsoft went through at least three generations of MBT tools developed and implemented in-house [44, 45].

   Their newest tool, Spec Explorer, was used at Microsoft to test an interaction protocol between Windows operating system components, the implementation code coverage increased from 60% to 70% and 10 times more errors were found than with standard manual testing. Additionally, twice as many design flaws as implementation defects were found

[1] a test oracle or simply oracle determines whether a test has passed or failed [42]

throughout this approach, demonstrating the effectiveness of Model-Based Testing in early mistake discovery and design validation [45].

3. Model-Based Testing in the Smart Card Industry

   Because smart card software demands a high level of validation and requires a lot of testing to assure conformity to standards, as much as half of the development effort is reserved for testing. As a result, incorporating Model-Based Testing into the current testing procedure is straightforward [1].

   For example, GSM 11.11 standard case study resulted in a substantial 30 percent decrease in workload regarding test design time, while providing broad coverage of around 85% [46].

4. Model-Based Testing in the Automotive Industry

   A conducted evaluation of MBT of an infotainment network has shown that automated suites are roughly as good as manual testing at detecting programming errors, but they are far better at detecting requirements-related errors. Additionally, only an 11% gain in error detection rates was achieved by automatically generating the test suite six times as big, suggesting that there is a limit to the benefits of larger test suites. [47].

Another study named "Implementation of Model Based Testing for Testing Kawn Subscriptions Manager Application" [48] indicates that the MBT-generated test cases are sufficient in regard to the mutation score. However, while MBT was effective in creating test cases that looked at the behavior of the application, it was discovered that the technique was unable to identify errors caused by mutations that had little to no effect on the behavior of the application. According to the research, MBT can be a useful tool for testing important features, but other techniques are required to discover errors that might be caused by little or nonexistent changes to the application code.

One of the MBT methods we evaluated was Combinatorial Interaction Testing (CIT). There is a noticeable study gap about the practical efficiency of CIT in identifying real issues in industrial settings, even though several studies [41, 49–52] have examined the effectiveness of CIT using simulations and mutation testing. Discussions focused on industry applications [53–55] and a few papers [53, 54, 56, 57] offer information about the time and test efficiency of CIT in identifying actual system defects. However, there are fewer documented comparisons between CIT and ad hoc testing methods.

Chapter 4

Evaluating the effectiveness of model-based testing techniques

4.1 Research design and approach for evaluating model-based UI testing effectiveness

This section outlines the research design and approach specifically focused on evaluating the effectiveness of Combinatorial Interaction Testing (CIT) within model-based UI testing frameworks. The approach was designed to systematically assess how CIT can enhance UI testing by improving defect detection rates, test coverage, and efficiency in test execution.

4.1.1 Objective

The primary objective was to assess the effectiveness of CIT in the context of model-based UI testing, specifically in relation to traditional testing approaches in terms of coverage, efficiency, and defect detection.

4.1.2 Methodology

The methodology incorporated specific elements to effectively evaluate CIT:

1. **Selection of CIT Tools:** We chose a tool that is capable of generating tests for 2-way through 6-way interactions. This tool was also chosen, because it generates fewer test sets while maintaining the same level of coverage.
2. **Experimental Design:** Controlled experiments were designed to directly compare the outcomes of CIT with those of traditional testing approaches. The experiments were designed to guarantee an accurate assessment of the efficiency and the ability to discover defects within the SUT.
3. **Case Study Implementation:** Three case studies were conducted using CIT in *close to* real-world scenarios to validate its practical effectiveness and applicability across various software applications and environments.

4. **Data Collection and Analysis:** We systematically collected quantitative data to measure the impact of CIT on test coverage, defect detection rates, and testing time.

4.1.3 Evaluation Criteria

CIT was evaluated based on the following metrics:

- **Defect Detection:** Evaluating the effectiveness of CIT in identifying defects, especially those resulting from complex interactions between different UI elements.
- **Efficiency:** Evaluation of the effectiveness of CIT in terms of minimizing the time and resources required for detailed test case development and execution.

4.2 Selection of testing tools and frameworks

In this section, we will cover all the tools and frameworks used in the case studies and explain why they were used.

4.2.1 ACTS 3.2

ACTS 3.2[1] was used to generate test cases, focusing on achieving 2-way and mixed coverage. This tool helped to create test cases systematically, ensuring comprehensive parameter interaction coverage. The 2-way coverage approach ensured that all possible pairs of parameter values were tested, whereas the mixed coverage criteria permitted the inclusion of more complex test scenarios.

4.2.2 Cypress

To automate some portion of the test cases, we have used Cypress[2]. Cypress is a test automation tool specifically designed for modern web applications. It works directly within the browser, allowing for real-time interaction with the application under test. This direct browser execution provides immediate feedback and simplifies the testing process by automatically handling tasks like waiting for elements.

4.2.3 Git/Github

We used Git[3] for version control to manage our test automation scripts, allowing for tracking of changes. We utilized GitHub[4] for code sharing, enabling

[1] ACTS 3.2. Available at `https://csrc.nist.gov/projects/automated-combinatorial-testing-for-software`

[2] Cypress. Available at `https://www.cypress.io/`

[3] Git. Available at `https://git-scm.com/`

[4] Github. Available at `https://github.com/`

us to store and share our test automation scripts with team members[5].

4.2.4 Google Sheets as a Test Management Framework

Google Sheets[6] is a web-based spreadsheet application that enables users to create, edit, and collaborate on spreadsheets online. It's part of the free, web-based Google Docs Editors suite offered by Google within its Google Drive service.

In section 2.6.2, we discussed that using a test management framework leads to decreased software deployment times and increased test coverage, however, these are often not free and if they are, they do not provide any significant features over Google Sheets. We chose this software for those reasons, as well as the convenience of not having to learn another framework.

We used it to track case study data involving time spent to create test scenarios, program and record the automated tests and execute the tests, effectiveness to detect defects, and to monitor the list of defects and test cases. In study covered in section 4.3.2 we also used it to track the participant observations and the test scenarios.

4.3 Design of experiments and case studies

In this section, we will explore the methods, setups and the results of the three case studies we conducted.

4.3.1 Impact Analysis of Combinatorial Interaction Testing on Systematic Software Validation

The primary objective of this thesis is to evaluate the impact of Combinatorial Interaction Testing (CIT) on software quality assurance, with a specific focus on its effectiveness in detecting defects in the OpenCart system. This study seeks to establish empirical data regarding the effectiveness of CIT in a controlled setting by introducing artificial defects and conducting targeted module testing.

Experiment Method

The experiment method for this study is outlined in Figure 4.1. The process begins with the preparation of the testing environment, followed by the setup of the test platform.

The process of developing test cases is divided into two separate paths: manual test case creation and CIT test case creation. During the process of Manual Test Case Creation, scenarios are determined by analyzing the functional requirements. Test cases are written and reviewed manually.

[5]The 'Effectiveness of Combinatorial Interaction Testing in Test Automation' study was done in collaboration with Ing. Feras Abdul Hadi Mustafa Daoud, doc. Ing. Miroslav Bureš, Ph.D. and Bc. Petr Syrovátka.

[6]Google, Google Sheets. Available at https://www.google.com/sheets/about/

During the CIT test case creation process, parameters and their possible values are established, constraints and interaction levels are specified in the advanced combinatorial testing system (ACTS), and the test cases are then automatically created using the generated output from ACTS.

After the test cases have been created, they are executed in the prepared testing environment. This means that we are deploying both manually created and CIT-generated test cases on the platform and executing them to gather results. The results of the test executions are documented, which includes if a defect was detected and the time spent on each test case.

The final phase of the experiment method involves analyzing these results. This analysis evaluates the effectiveness of the manual versus CIT methods, particularly focusing on time effectiveness and defect detection rate. Subsequently, we gather and analyze the results, which then serve as the foundation to evaluate the relative effectiveness of the two testing methods.

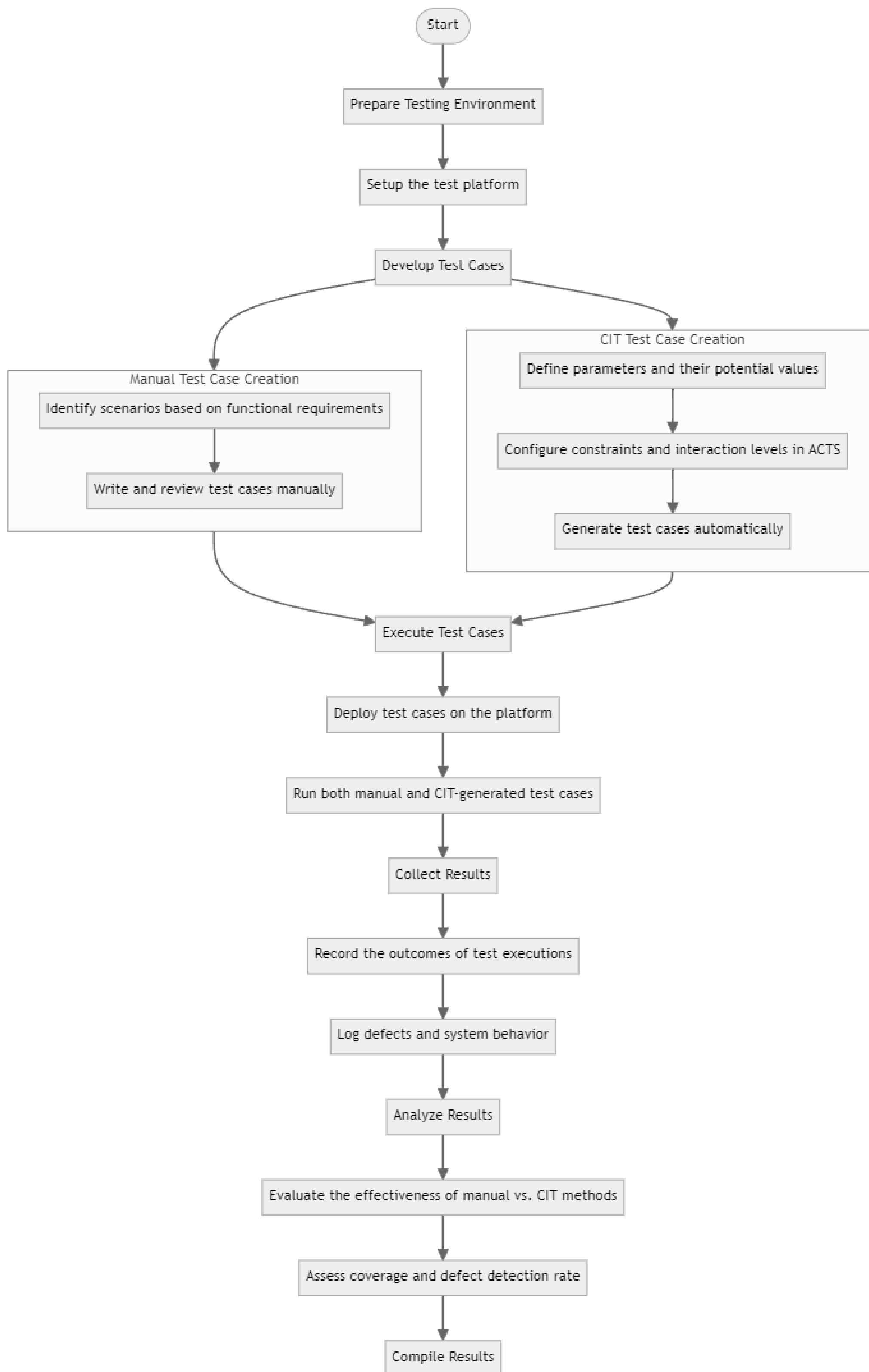


Figure 4.1: Flowchart of the experiment method

Experiment setup

1. **System under test**

   The OpenCart[7] system, hosted locally, served as the testing platform for this study. OpenCart is an open source shopping cart solution. The tests have been carried out in a controlled black-box environment, replicating real-world usage conditions without having access to the core structures of the software. The experiment was conducted using the stable version v4.0.2.3.

   Our testing efforts were primarily directed towards two main forms in the SUT: the Subscription Plan Form, which allows the administrator to create a new subscription plan in the system, and the Attribute Form, used for creating a new attribute within the system. In the Subscription Plan Form, we identified 9 fields that are considered relevant parameters to the CIT problem. Similarly, in the Attribute Form, there are 3 fields that are relevant to the CIT problem. The experiment aims to investigate the proper functionality of these forms within the OpenCart environment.

   Figures 4.2 and 4.3 depict the graphic user interface (GUI) of OpenCart. The former displays the subscription plan form, while the latter showcases the form for defining a new attribute.

2. **Defects**

   The OpenCart system was exposed to deliberate introduction of artificial defects in order to evaluate the form validation. These defects were designed to simulate typical errors that may arise in real-world applications. The defects included scenarios such as allowing negative numbers in duration fields and inadequate handling of extreme values, which assessed the system's ability to withstand faulty inputs.

3. **Test Case Design**

   The study required generating two separate sets of test cases to assess the effectiveness of Combinatorial Interaction Testing (CIT) in comparison to conventional test case design methodologies. The initial batch included conventional test cases that were created without employing CIT methodologies. The purpose of these tests was to assess the fundamental functionality and error-handling capabilities of the system using simple scenarios. Every test case was designed to assess particular functionalities or failure spots, replicating user interactions and verifying system responses against anticipated results.

   The second set used CIT. By utilizing ACTS to construct combinatorial test cases, we successfully developed a comprehensive suite that effectively examined various parameter interactions. This expanded the scope and complexity of the testing process and attempted to promote productivity by minimizing the repetition commonly found in manually generated test suites.

   To generate the CIT test cases for the OpenCart system using the ACTS tool, two specific suites were defined: one for the creation of

[7]OpenCart. Available at `https://www.opencart.com/`

Figure 4.2: OpenCart - Subscription Plan Form

subscription plans and another for adding an attribute. For the suites 'Adding a Subscription Plan' and 'Adding an Attribute' all relevant parameters such as input fields, command buttons, and user roles were identified and listed.

Subscription Plan Form - In this form, which is used to create a new subscription plan, 9 parameters with 26 different values were used for the CIT test case generation. There were 15 test cases in the original test set, of which 13 were replaced by 25 test cases generated by CIT.

Attribute Form - For this form, the original test set comprised of 7 test cases, 6 of which were replaced by 10 CIT generated test cases. For the CIT test case generation, 3 parameters with 9 total values were identified.

The number of parameters and the number of their respective values for the Adding a Subscription Plan Form and the Adding an Attribute

Figure 4.3: OpenCart - Attribute Form

Parameter Name	Number of values
Subscription Plan Name	2
Trial Duration	4
Trial Cycle	2
Trial Frequency	5
Trial Status	2
Duration	2
Cycle	2
Frequency	5
Status	2
Total	26

Table 4.1: Number of values of each parameter in Adding a Subscription Plan Form

Parameter Name	Number of values
Attribute Name	2
Attribute Group	5
Sort Order	2
Total	9

Table 4.2: Number of values of each parameter in Adding an Attribute Form

Form are presented in Table 4.2 and Table 4.1 respectively.

Using the tool ACTS, test cases were generated that covered all meaningful combinations of identified parameters, focusing on pairwise interactions.

4. Test Automation Scripts

Regarding the test cases without CIT, for the Subscription Plans suite, out of a total of 15 manually created test cases, 13 were then

automated, translating to an automation rate of approximately 87%. In the Attributes suite, 7 out of the original 7 manual test cases were automated, achieving a 100% automation rate.

Furthermore, Combinatorial Interaction Testing (CIT) added an extra level of automation. The CIT approach, via the ACTS tool, produced a multitude of test cases by considering each possible combination of input factors, as specified by the constraints defined inside the tool. All of these test cases were then covered by automated tests.

The configuration for CIT was defined within two XML files: `Subscription Plans.xml` and `Attributes.xml`. The files organized the test parameters and their corresponding values, making it easier to generate comprehensive test cases. The CIT setup included several parameters, such as 'SubscriptionPlanName', 'TrialDuration', and 'AttributeName', each with several possible values to include a wide range of test situations.

The test cases were generated using 2-way uniform strength combinatorial array technique, guaranteeing the testing of all potential interactions. The following table 4.3 is a concise overview of the parameters and their associated values utilized in the CIT configuration for the 'Subscription Plans' suite:

Parameter	**Values**
SubscriptionPlanName	Invalid, Valid
TrialDuration	Valid number, Zero, Negative number, Non-numeric input
TrialCycle	Valid, Invalid
TrialFrequency	Day, Week, Semi Month, Month, Year
TrialStatus	Checked, Unchecked
Duration	Valid number, Continuous
Cycle	Valid, Invalid
Frequency	Day, Week, Semi Month, Month, Year
Status	Checked, Unchecked

Table 4.3: CIT Parameters and Values for Subscription Plans

Each parameter was exhaustively combined with others using the IPOG algorithm, as specified in the ACTS's configuration, to ensure thorough coverage of all possible parameter interactions.

5. **Time Recording and Defects Detection**

   We have manually recorded the amount of time spent on several tasks related to developing, automating, and assessing test cases. These tasks include the time spent on designing the test cases, executing them, detecting defects, and automating the 'manual' test set. The data was kept in a Google Sheet.

4.3.2 Combinatorial Interaction Testing in Usability Studies

The main aim of this study is to investigate the effectiveness of Combinatorial Interaction Testing (CIT) in detecting usability problems in software appli-

cations when compared to conventional usability testing approaches. The objective of this research is to determine if CIT can effectively identify a wider array of problems by methodically examining different combinations of user interactions, system setups, and interface elements and decrease the testing process time.

Experiment method

1. **Participants**
   A total of 10 individuals were recruited for the study, who were divided into two groups of five. The participants were chosen to cover a spectrum of user experiences, ranging from beginners to experts, in order to ensure that the findings of the study would be relevant to a broad user base.
2. **Group Allocation**
   - Group 1 (CIT Group): Underwent testing using scenarios generated through Combinatorial Interaction Testing.
   - Group 2 (Traditional Group): Was tested using traditional usability testing methods developed based on judgment and typical user feedback mechanisms.
3. **Data Collection Methods**
   The data collection for this usability study was conducted using structured scenarios to evaluate the effectiveness of various tasks in the system.
   For each scenario, time taken to design the tasks were recorded. Also, observations were made on the ease of use, any difficulties encountered and overall user satisfaction. The data was systematically captured using a structured sheet that started empty and was filled out during the process, with columns for observations and each participant's data, as shown in the table 4.4:

Observations	Participant 1	Participant 2	...	Participant 10
			...	
			...	
			...	
			...	

Table 4.4: Participant Observations Collection Sheet for Usability Study

Experiment setup

1. **System under test**
   The Odoo[8] demo system, accessed online, served as the testing platform for this study. Odoo is an open source ERP and CRM bundle. The tests were conducted in a controlled black-box environment that

[8] Odoo. Available at `https://www.odoo.com/`. Demo available at `https://master.odoo.com/saas_master/demo/`

replicated real-world usage settings. This approach guarantees that the usability evaluation accurately mirrors the experience of the end-user. The experiment was done using the latest stable version that was available during the investigation.

We focused our testing efforts on the 'Employees', 'Product Inventory' and 'Sales Quotations' modules. The system dashboard is shown in figure 4.4 and the forms can be seen in figures 4.5, 4.6 and 4.7.

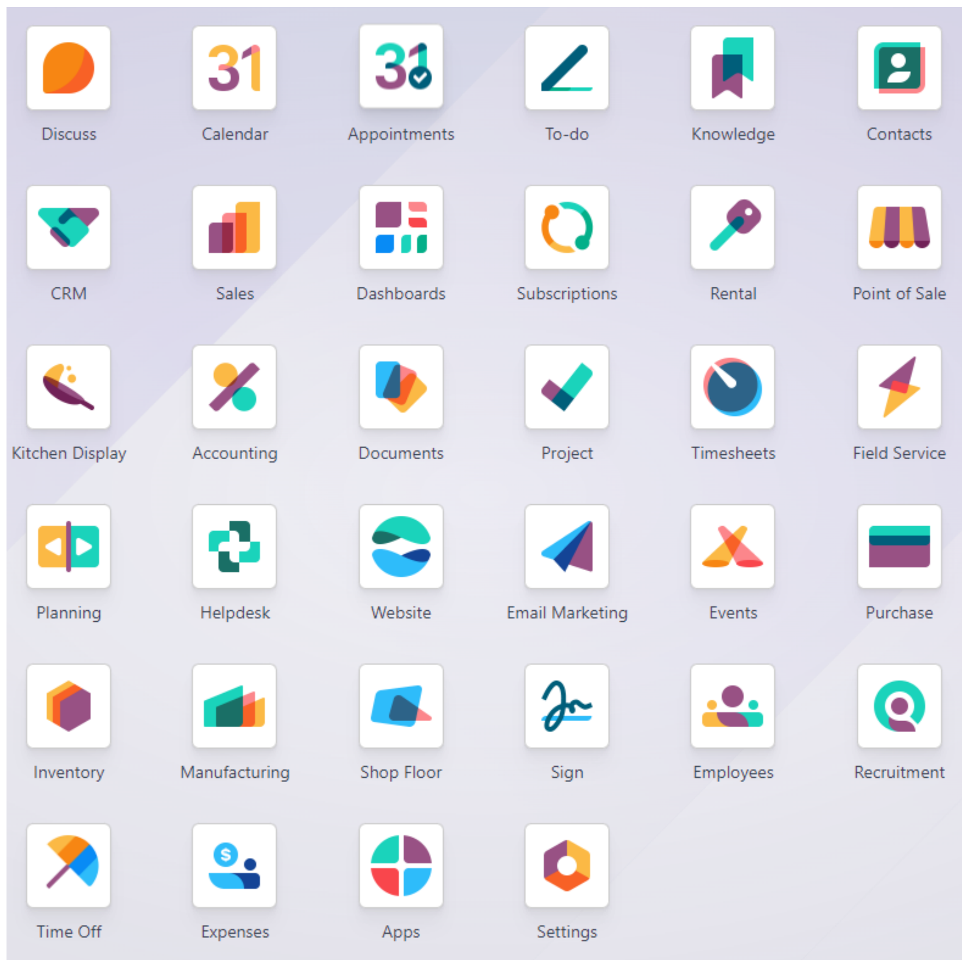


Figure 4.4: Odoo - Dashboard

Employee's Name
Job Position
Tags
Work Mobile
Work Phone +1 (650) 555-0111
Work Email
Company Demo Company
Department
Job Position
Manager
Coach ?
Resume | Work Information | Private Information | HR Settings
RESUME
Create a new entry
SKILLS
You can add skills from our library to the employee profile.
If skills are missing, they can be created by an HR officer.
Pick a skill from the list
Send message Log note Activities Follow

Figure 4.5: Odoo - Employee Form

Figure 4.6: Odoo - Product Form

Figure 4.7: Odoo - Sale Quotation Form

2. **Test scenario design**
   The study utilized two separate sets of test scenarios to assess the effectiveness of conventional usability testing approaches in comparison to Combinatorial Interaction Testing (CIT).
   We first created the traditional test scenarios - one of them can be seen in the example below:
   - **Scenario 2: Product Inventory Addition**
     - **Objective:** Evaluate the process of adding a new product to the inventory in the Inventory module.
     - **Steps:**
       a. Go to the Inventory module.
       b. Select 'Products' and then 'Products' again from the drop-down menu.
       c. Click on 'New' to add a new product.
       d. Enter product details such as name, sales price, cost, and internal reference.
       e. Save the product and check for its presence in the inventory list.

- **Time to design scenario:** 20 minutes.
- **Browser:** Chrome.

Then, we created the CIT test scenarios. Table 4.5 shows the CIT parameters and values used for generation. ACTS tool was utilized to construct test scenarios by utilizing the parameters and values supplied in an XML file. These test scenarios covered all possible combinations of the stated parameters.

Parameter	Values
Module	Employees, Sales Quotations, Product Inventory
Task Type	Create, Edit, Delete
Browser	Chrome, Firefox, Safari, Edge

Table 4.5: CIT Parameters and Values for Usability Testing

4.3.3 Effectiveness of Combinatorial Interaction Testing in Test Automation

The primary objective of this study is to evaluate the impact of Combinatorial Interaction Testing (CIT) on improving test automation procedures, with a specific focus on its ability to detect usability problems within the Jtrac system. This study aims to quantitatively analyze the effectiveness of CIT by systematically generating and executing scenarios in a controlled testing environment. The study investigates the possible improvement in defect detection and overall system evaluation by using CIT-generated scenarios, as compared to traditional testing methodologies.

Experiment method

In this study, the methodology mirrored the experiment design outlined in study described in section 4.3.1, where two distinct testing approaches, manual and CIT, were compared.

First, the testing environment was set up to facilitate both types of testing. Test cases were developed through both manual processes, where scenarios were directly derived from functional requirements, and through CIT, where test case scenarios were automatically generated using predefined parameters in an ACTS tool. Following this, the test cases were executed across the platform. Results from these tests were collected and analyzed to evaluate the effectiveness of each method.

The primary distinction is that we also evaluated the mixed strength test set generation allowing for different parameter groups to be created and covered with different strengths as opposed to 2-Way test set generation.

Experiment setup

1. **System under test**

JTrac[9] is a web application that is open-source and specifically designed for the purpose of tracking issues. JTrac allows for customizable custom fields and includes all the standard features of an issue-tracking system, such as support for file attachments and integration with email. The experiment was carried out on the stable version 2.3.1, which was released in May 2023, and subsequently on version 2.3.2, which was released in January 2024.

We focused our testing efforts on two principal forms in the SUT: the Issue tracking form, where the user can create a new issue in the system, in which 8 fields were identified as relevant parameters to the CIT problem, and the Login form used for entering the system, containing 2 fields relevant to the CIT problem. In the experiment, the correct functionality of these forms within the JTrac environment is investigated.

Figures 4.9 and 4.8 illustrate the user interface of JTrac, showcasing the login form and the form of creating a new item, respectively.

Figure 4.8: JTrac - Login Form

Figure 4.9: JTrac - Creating New Item Form

[9] JTrac. Available at `http://jtrac.info/`

2. **Defects**

When creating test cases, two specific types of defects were considered: historical and artificial. The historical defects were derived from the publicly accessible bugs page[10] of JTrac, which documents real-world issues that have been encountered in the past. These defects provide as a foundation for comprehending the system's behavior under well-known fault scenarios.

In order to counterbalance the inherent stability of the system, which naturally had fewer noticeable defects, extra simulated defects were deliberately inserted. These artificial defects were specifically designed to be more easily detectable, guaranteeing that the testing process could thoroughly assess the effectiveness of the test cases in discovering flaws across different scenarios. This methodology not only achieves balance in the test environment, but also replicates a broader range of possible operational obstacles, thereby improving the dependability of the testing process.

3. **Test cases**

We conducted three distinct test sets and executed each set once. The initial set of test cases was generated without utilizing Combinatorial Interaction Testing (CIT). This set consisted of a total of 136 test cases. The second set was generated by substituting certain test cases with newly created ones using CIT, specifically employing a 2-way uniform strength combinatorial array technique. We created the third set using a distinct CIT technique known as a mixed-strength combinatorial array.

The test cases were separated into suites - *login page* (corresponds to Login Form later in the text), *dashboard*, *create new issue* (corresponds to Creating New Item Form later in the text), *issue record*, we then decided to focus on the *login page* and *create new issue* as CIT could be used to generate most of the test cases for the suites.

Login Form - This is just a simple login form, only 2 parameters with 4 different values were used for the CIT test case generation.

The initial test set contained seven test cases, which were reduced to 6 through CIT. This reduction involved replacing 5 of the original test cases with 4 new ones, both in the 2-way uniform strength combinatorial array and the mixed-strength combinatorial array.

Creating New Item Form - In the second part of the experiment, we used a form for creating a new item (an issue in this case) with 8 identified parameters and 20 values in total used for CIT test case generation. Their numbers for individual parameters are presented in Table 4.6.

After implementing CIT with 2-way uniform strength, the original test set consisting of 28 test cases was replaced with a reduced set of 12 test cases. The number of test cases generated was the same for the third test set with mixed strength.

[10] JTrac Issues. Available at `https://sourceforge.net/p/j-trac/bugs/`

Parameter Name	Number of values
Summary	2
Detail	2
Severity	6
Name	2
AssignTo	2
NotifyByEmail	2
Attachment	2
SendEmailNotifications	2
Total	20

Table 4.6: Number of values of each parameter in Creating New Item Form

4. **Test Automation Scripts**

As stated before, we used Cypress to create the test automation scripts. Automating all of the test cases would be too time-consuming, so we chose to automate only a portion of the test cases - approximately 38%. We specifically selected test cases from both suites to create the best possible approximation. We believe the approximation to be correct since the automation scripts are fairly repetitive for specific test cases within suites. Next, we will cover one of the test cases from the *create new issue* suite.

This specific test case aimed to validate the JTrac 'Create' functionality with the validity of specific fields set up according to the test plan. This information is also preserved in the *describe* function.

```
describe('Summary-valid_Detail-valid_Severity-
    Suggestion_name-valid_AssignTo-valid_NotifyByEmail-
    admin_Attachment-not_attached_sendEmailNotifications-
    false', () => {
  it('Logs in, navigates, fills out a form, and submits
    it', () => {
      // Step 1: Login
      cy.visit('http://localhost:8888/app/login');
      cy.get('#loginName1').type('admin');
      cy.get('#password3').type('admin');
      cy.get('input[type="submit"]').click();
      cy.url().should('eq', 'http://localhost:8888/app/');


      // Step 2: Navigation
      cy.get('table.jtrac a').first().click();
      cy.url().should('include', '/app/item/form');

      // Step 3: Form Submission
      cy.get('[id^="summary"]').type('bug');
      cy.get('textarea[name="detail"]').type('Random text
```

```
     ' + Math.random().toString(36).substring(7));
      cy.get('select[name="fields:fields:0:field:border:
    field"]').select('5');
      cy.get('input[name="fields:fields:1:field:field"]').
    type('name1');
      cy.get('select[name="hideAssignedTo:border:
    assignedTo"]').select('1');
      cy.get('input[name="hideNotifyList:itemUsers"]').
    check();
      cy.get('input[name="sendNotifications"]').uncheck();

      cy.get('input[type="submit"]').click();

      // Step 4: Verification
      cy.url().should('match', /http:\/\/localhost:8888\/
    app\/item\/TEST-\d+\//);
  });
});
```

The test begins by navigating to the application's login page, which is hosted locally. It uses the *cy.visit* command to load the page and then enters the credentials of a valid user into the appropriate fields. This step is critical for ensuring that subsequent actions take place within an authenticated session.

After successful authentication, the script navigates to the issue creation form within the application. This is accomplished by choosing the first link in a table identified by a class attribute. The navigation step is validated by asserting that the current URL contains a path indicating the form's location, ensuring that the test runs in the proper context.

The test consists primarily of filling out a form with predefined data. This includes providing a summary, a detailed description with a dynamically generated string to ensure uniqueness, choosing a severity level, specifying additional information such as the assignee and notification preferences and disabling email notifications for form submission.

The final step is to submit the form and verify its success by checking the URL pattern, which should match a specific format indicating a unique identifier for each submitted item. This verification ensures that the form submission process not only runs smoothly but also produces the desired result, which is the creation of a new issue within the application.

5. **Time Recording and Defects Detection**

We have manually kept track of time spent on several categories when creating, automating and evaluating the test cases: *Time spent on each test case (min), Time spent test execution, Defect detected (T/F)*. For simplicity, we also have the cumulative categories of *Time spent test execution on suite (min) and Time spent creating test cases (min)*. The

data were stored in a Google Sheet.

We have designed and automated at least 25% of the tests for each test set, with the remaining test set data being artificially estimated using the average of those tracked times as a rough guide.

Regarding defect detection, we executed the test cases and registered the values into the table. Note that some of the test cases were automated and some were not.

4.3.4 Threats to Validity

This subsection discusses the potential threats to the validity of the findings from three studies on Combinatorial Interaction Testing (CIT) in usability and test automation. We identify and elaborate on several categories of threats that could potentially invalidate the results. We will point out the consequences for a different system under test (SUT) that could be used in a different study or an industrial setting.

Limited Diversity in Testers

In all three studies, testing was conducted by a limited number of testers (two studies done by a single tester, the *Effectiveness of Combinatorial Interaction Testing in Test Automation* study done in collaboration by a team of three testers). This restricts the diversity in testing approaches and perspectives, which might lead to overlooking specific defects that a more varied group of testers could otherwise identify.

Experience of Testers

Only testers without previous testing experience were used in two of the studies. As for the usability study, the person conducting the study also had no previous experience in conducting usability studies. This lack of experience might negatively influence the effectiveness of defect detection and the generalization of the study results, as beginner testers might not effectively detect issues a more experienced tester would.

Confirmation Bias

There is an inherent confirmation bias in the studies due to the predefined setup and methodology, particularly in the selection and creation of test cases. This bias has the potential to influence the outcome by showing preference towards specific sorts of defects or system behaviors, which in turn distorts the evaluation of CIT effectiveness.

Artificial Defects

In two of the studies, we deliberately inserted artificial defects into the SUT to balance the inherent stability as well as low defect presence. Although this

approach helps in evaluating the test case effectiveness, it may not provide a realistic representation of real-world scenarios, where defects are not known in advance and could be more complex or subtle or just different altogether from the ones we inserted.

Homogeneity of Test Environments

The experiments were carried out in controlled scenarios, which may not adequately reflect the different operational environments in which the software would actually operate. The study's external validity may be compromised by this constraint, as the findings may not apply to various hardware or software setups.

Reproducibility of Results

The reproducibility of the experiments may be compromised by the particular settings and versions of the software used, such as JTrac versions 2.3.1 and 2.3.2. Variations in software behavior between different versions or configurations may result in different outcomes when attempting to repeat the studies.

Dependence on CIT Techniques

The effectiveness of the testing heavily relies on the specific CIT techniques employed (e.g., 2-way uniform strength, mixed-strength arrays). Differences in these CIT techniques may lead to large variations in the results, which could raise doubts about the reliability of the conclusions made regarding the overall effectiveness of CIT. The variability highlights the significance of consistently applying the methods in order to accurately evaluate the actual effect of CIT on software testing results.

Chapter 5

Results

This chapter summarizes the key findings from the evaluation of Model-Based Testing techniques as discussed in chapter 4. Setting up the case studies allowed for the collection of the necessary empirical data and offered valuable insights into these testing approaches.

5.1 Results of Specific Case Studies

Impact Analysis of Combinatorial Interaction Testing on Systematic Software Validation

We assessed the influence of Combinatorial Interaction Testing (CIT) on software testing efficiency by comparing conventional testing approaches with those that integrate CIT.

The results, summarized in the table 5.1, provide a detailed comparison of defect detection improvements facilitated by CIT.

Metric	Without CIT	With CIT
Total Test Cases		
Subscription Plans	15	27
Attribute	7	11
Total Time Spent (hours)		
Total	10:51:00	10:27:15
Defects Detected		
Subscription Plans	7	7
Attributes	5	6
Total Defects Detected	12	13
Defect Detection Effectiveness		
Average Time per Defect Detected	54.25 minutes	48.25 minutes

Table 5.1: Comprehensive Comparison of Testing Metrics with and without CIT

The results indicate that the utilization of CIT has led to an increase in the overall number of test cases and a small decrease in the total testing time. More importantly, it has also led to uncovering one additional defect in the

Adding an Attribute form. However, the total number of test cases with CIT increased for both the Adding a Subscription Plan and Adding an Attribute form by 80% and roughly 57% respectively.

Combinatorial Interaction Testing in Usability Studies

The effectiveness of Combinatorial Interaction Testing (CIT) was compared with traditional usability testing methods. We specifically targeted *Time to Design Test Scenarios* and *Time to Complete Tasks* in this study.

Time to Design Test Scenarios. The time taken to design test scenarios was measured. Traditional scenarios took an average of 20 minutes to design, while CIT scenarios required an initial setup time of 30 minutes, followed by specification of the generated test scenarios. This is summarized in Table 5.2.

Scenario Type	Setup Time (minutes)	Average Design Time (minutes)
Traditional	N/A	20
CIT	30	6:15

Table 5.2: Average Time to Design Test Scenarios

While the initial setup for CIT scenarios was longer, the following generation of subsequent test scenarios had reduced effort required for scenario creation and overall was much faster (by 320%[1]).

Time to Complete Tasks. We recorded the time taken by participants to complete the tasks for both groups. Table 5.3 provides the average time taken by participants in each group to complete the tasks.

The CIT group completed the tasks slightly faster on average compared to the traditional group, indicating a potential reduction in the overall testing process time when using the CIT approach. This could also be caused by the CIT group having more tasks, leading to the participants getting more experience with the system during the interviews.

Group	Average Time (minutes)
CIT Group	25:57
Traditional Group	26:27

Table 5.3: Average Task Completion Time

Effectiveness of Combinatorial Interaction Testing in Test Automation

We will assess the effectiveness of testing conducted without the knowledge of Combinatorial Interaction Testing (CIT) in comparison to testing conducted with the utilization of CIT.

[1] not accounting for setup time

Table 5.4 displays the results of this study. In the table, we have differentiated between the Login and Issue create forms to provide more comprehensive data, as they exhibit notable differences. Table 5.4 categorizes the arrays as '2-way' if they have a uniform strength and as 'Mixed' if they have varying strengths. These descriptions will be used as references when discussing the results later on. 'Without CIT' refers to the initial test cases that were created without using the CIT methodology. This study examines the effectiveness of CIT in creating and implementing test cases, with a particular focus on the Login and Issue create forms.

The results of our study demonstrate significant improvements in test efficiency and defect identification when using CIT. The use of CIT instead of traditional methods significantly reduced the time required to generate test cases for the Login form. Both the 2-way and Mixed CIT methods resulted in a time reduction of 42.8% compared to the non-CIT approach. Test cases that remained unchanged maintained their original time duration. The execution time for substituted test cases in the Login form was effectively halved using both CIT methodologies. The generation of test cases for the Issue create form showed a significant decrease of 57.14% when using the CIT method, both with the 2-way and Mixed CIT methods.

Overall, the Login form showed a 25% decrease in total time with both CIT approaches. The Issue create form, on the other hand, experienced a more significant reduction, with a decrease of 76.26% for the 2-way CIT method and 79.86% for the Mixed CIT method.

Defect detection saw an increase in the number of identified defects when CIT was applied, with the count rising from 13 without CIT to 16 with the implementation of both CIT methods. The defect detection rate has improved by 18.75% with the use of CIT. The application of CIT resulted in a notable reduction in the number of test cases for both forms, leading to an enhanced efficiency of the testing process. The results underline the advantages of CIT in improving software testing by reducing the time required for generating and executing test cases and enhancing defect detection rates.

The mean duration required to detect a single defect reduced from 1.4 hours in the first test set to 0.4 hours utilizing the CIT 2-way technique, and further enhanced to 0.35 hours with the CIT Mixed technique. The 2-way and Mixed CIT techniques had a significant reduction in detection time, with a drop of 71.43% and 75% respectively.

5.2 Summary of Key Results

- **Increased Defect Detection:** The studies generally indicated an increase in the detection of defects when employing CIT. This increase is likely attributed to the systematic exploration of parameter interactions that these techniques facilitate.
- **Variable Reduction in Testing Time:** The reduction in testing time due to the adoption of CIT varied. One study reported a significant

	Without CIT	2-way (CIT)	Mixed (CIT)
Total Test Cases			
Login Form	7	4	4
Issue create form	28	12	12
Total Time Spent (hours)			
Total	17.7	6.15	5.65
Defects Detected			
Login Form	4	5	5
Issue create form	9	11	11
Total Defects Detected	13	16	16
Defect Detection Effectiveness			
Average Hours per Defect Detected	1.4	0.4	0.35

Table 5.4: Comparison of count of test cases, count of defects detected and time spent between 'without CIT' and '2-way' and 'Mixed' methods

reduction, while another observed a minimal impact on testing duration. We think that this highlights the dependency on specific test conditions and configurations.

- **Cost-Effectiveness Considerations:** Initial findings suggest that using CIT can be cost-effective in the long run due to decreased manual effort, fewer requirements for extensive test case maintenance and due to the fact that we recorded a decrease in time needed for defect detection. However, the upfront cost and resource investment are notable and should be considered.
- **Positive Impact on Test Coverage:** Model-Based Testing techniques, specifically Combinatorial Interaction Testing, consistently improved test coverage. However, the degree of improvement varied across the three case studies.
- **Time Implementation Considerations:** The technique we implemented takes additional time to setup compared to traditional testing methods, which was however outweighed by the time saved later in the process.

5.3 Implications of Results

These findings imply several practical considerations for the adoption and implementation of Model-Based Testing techniques in software development:

- **Context-Specific Integration:** The impact of Model-Based Testing techniques can vary significantly depending on the specific software and testing environment. Software development teams should evaluate these techniques within the context of their specific conditions to maximize benefits.

- **Selective Tool Adoption:** The choice of Model-Based Testing tools should be aligned with the system under test and the particular needs of the project. Not all tools yield the same level of benefit, as observed in the varied outcomes of the studies.
- **Training and Skill Development:** Given the complexity and specialized nature of Model-Based Testing, adequate training and skill development are necessary for teams to leverage these techniques effectively.

5.4 Limitations of the Study

The study acknowledges certain limitations:

- **Controlled Test Environments:** The experiments were performed in controlled settings, which may not adequately represent complex and unexpected nature of software development (e.g. autonomous driving systems and large-scale distributed systems like cloud computing platforms).
- **Presence of Artificial Defects:** Artificial defects were used in two of the case studies to evaluate the effectiveness, however, this may skew the data in favour of the tested technique.
- **Limited Generalizability:** The results are based on specific case studies and may not be universally applicable across different types of software systems or industry sectors.

5.5 Recommendations for Future Research

Future research is needed on the use of Model-Based Testing in more realistic and less controlled environments to validate the results discussed. It would be helpful to conduct long-term analyses to identify the long-term outcomes and costs of these testing methods.

Chapter 6 Conclusion

This study examined the effectiveness of model-based methodologies for evaluating user interfaces, targeted especially on Combinatorial Interaction Testing (CIT). The main goal was to evaluate how CIT may increase defect detection, improve test coverage and optimize the effectiveness of test case development and execution in comparison to conventional testing approaches.

We specifically worked on development of a system that enables evaluation of these techniques. There were three basic case studies conducted in order to collect empirical evidence on the utility of CIT. These studies covered a range of software applications and settings to evaluate the capabilities of CIT.

The first case study was performed on a shopping cart system and followed a methodology focused on comparing manual test case creation in comparison to CIT test case creation. The second study examined utilizing CIT in generation of test scenarios in a usability study performed on an ERP system. The third study followed the same methodology as the first study, but was conducted in an issue-tracking system.

Case studies indicate that CIT increases the probability of defect detection and increases test coverage while at the same time reducing the overall testing time. Unlike the conventional approaches to testing, CIT applies a systematic way of testing different combinations of input parameters to ensure thorough testing, that may uncover defects that would otherwise have been missed. It is easy to see that CIT has great potential as a tool for software testing, especially for those complex applications with numerous configurations and interactions among them.

Still, this research has numerous limitations; however, the results and ideas presented in the article are certainly promising. The controlled experimental conditions are often not comprehensive, and they are not equal to actual software testing conditions. Further research should also try to replicate these findings in a broader and less controlled environments in order to confirm or deny the usefulness and reliability of Combinatorial Interaction Testing in real world scenarios.

Bibliography

[1] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.

[2] C. Schulze, D. Ganesan, M. Lindvall, R. Cleaveland, and D. Goldman, "Assessing model-based testing: an empirical study conducted in industry," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 135–144. [Online]. Available: https://doi.org/10.1145/2591062.2591180

[3] R. Marinescu, C. Seceleanu, H. Guen, and P. Pettersson, *A Research Overview of Tool-Supported Model-based Testing of Requirements-based Designs*, 12 2015, vol. 98, pp. 89–140.

[4] J. R. Monsma, "Model-based testing of web applications," Master's thesis, Radboud University Nijmegen, 2015.

[5] L. Ye, "Model-based testing approach for web applications," Master's thesis, University of Tampere, 2007.

[6] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge: Cambridge University Press, 2016.

[7] L. Williams, "White-box testing," https://students.cs.byu.edu/~cs340ta/spring2019/readings/WhiteBox.pdf, 2006, accessed: 2024-04-03.

[8] ——, "Testing overview and black-box testing techniques," https://students.cs.byu.edu/~cs340ta/fall2018/readings/BlackBox.pdf, 2006, accessed: 2024-04-03.

[9] J. Britton, "What Is ISO 25010?" https://www.perforce.com/blog/qac/what-is-iso-25010, accessed: 2024-18-03.

[10] ISO/IEC 25010, *ISO/IEC 25010:2011, Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models*, Std., 2011.

[11] Codacy, "An Exploration of the ISO/IEC 25010 Software Quality Model," https://blog.codacy.com/iso-25010-software-quality-model, accessed: 2024-18-03.

[12] R. S. Yadav, "Improvement in the v-model," *International Journal of Scientific & Engineering Research*, vol. 3, no. 2, pp. 1–6, 2012.

[13] Sep 2023. [Online]. Available: https://testsigma.com/blog/testability/#Requirements_of_Software_Testability

[14] GeeksforGeeks, "Software testability," Feb 2023. [Online]. Available: https://www.geeksforgeeks.org/software-testability/

[15] CodiumAI, "Code Coverage," https://www.codium.ai/glossary/code-coverage/, accessed: 2024-09-03.

[16] ——, "Statement Coverage," https://www.codium.ai/glossary/statement-coverage/, accessed: 2024-09-03.

[17] ——, "Branch Coverage," https://www.codium.ai/glossary/branch-coverage/, accessed: 2024-09-03.

[18] SIMFORM, "Test Coverage," https://www.simform.com/blog/test-coverage/, accessed: 2024-10-03.

[19] S. Bose, "Best Practices for Unit Testing," https://www.browserstack.com/guide/unit-testing-best-practices, accessed: 2024-15-03.

[20] K. Devi, "What is Component Testing? (with Examples)," https://www.browserstack.com/guide/what-is-component-testing, accessed: 2024-15-03.

[21] Kitakabee, "What is Integration Testing," https://www.browserstack.com/guide/integration-testing, accessed: 2024-15-03.

[22] Katalon, "What is Regression Testing? Definition, Tools and Examples," https://katalon.com/resources-center/blog/regression-testing, accessed: 2024-15-03.

[23] S. T. Help, "Top 10 Most Popular Regression Testing Tools In 2024," https://www.softwaretestinghelp.com/regression-testing-tools/, accessed: 2024-15-03.

[24] Dec 2023. [Online]. Available: https://testsigma.com/blog/combinatorial-testing/

[25] D. R. Kuhn, R. N. Kacker, and Y. Lei, *Introduction to combinatorial testing*. CRC press, 2013.

[26] "Ieee standard glossary of software engineering terminology," *IEEE Std 610.12-1990*, pp. 1–84, 1990.

[27] BrowserStack, "What is Test Management?" https://www.browserstack.com/test-management/what-is-test-management, accessed: 2024-10-03.

[28] ——, "What is Test Plan?" https://www.browserstack.com/test-management/features/test-run-management/what-is-test-plan, accessed: 2024-18-03.

[29] IBM, "What is test management?" https://www.ibm.com/topics/test-management/, accessed: 2024-10-03.

[30] S. Bose, "How to write Test Cases in Software Testing? (with Format & Example)," https://www.browserstack.com/guide/how-to-write-test-cases, accessed: 2024-10-03.

[31] Oct 2023. [Online]. Available: https://testsigma.com/blog/roi-test-automation/

[32] [Online]. Available: https://www.headspin.io/blog/step-by-step-guide-to-calculate-roi-of-test-automation-for-digital-testing

[33] A. M. Memon, "An event-flow model of gui-based applications for testing," *Software testing, verification and reliability*, vol. 17, no. 3, pp. 137–157, 2007.

[34] A. M. Memon and B. N. Nguyen, "Advances in automated model-based system testing of software applications with a gui front-end," in *Advances in Computers*. Elsevier, 2010, vol. 80, pp. 121–162.

[35] S. Arlt, I. Banerjee, C. Bertolini, A. Memon, and M. Schäf, "Grey-box gui testing: Efficient generation of event sequences," 05 2012.

[36] J. Bowen and S. Reeves, "Ui-design driven model-based testing," 2009.

[37] V. Lelli, A. Blouin, B. Baudry, and F. Coulon, "On model-based testing advanced guis," in *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2015, pp. 1–10.

[38] M. Hutcheson, *Software Testing Fundamentals: Methods and Metrics*, 01 2003.

[39] P. B. Nirpal and K. Kale, "A brief overview of software testing metrics," *International Journal on Computer Science and Engineering*, vol. 3, no. 1, pp. 204–2011, 2011.

[40] Y. Singh, A. Kaur, and B. Suri, "An empirical study of product metrics in software testing," *Innovative techniques in instruction technology, e-learning, e-assessment, and education*, pp. 64–72, 2008.

[41] M. Bures and B. S. Ahmed, "On the effectiveness of combinatorial interaction testing: A case study," in *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 2017, pp. 69–76.

[42] W. Howden, “Theoretical and empirical studies of program testing,” *IEEE Transactions on Software Engineering*, vol. SE-4, no. 4, pp. 293–298, 1978.

[43] E. Farchi, A. Hartman, and S. S. Pinter, “Using a model-based test generator to test for standard conformance,” *IBM Systems Journal*, vol. 41, no. 1, pp. 89–110, 2002.

[44] K. Stobie, “Model based testing in practice at microsoft,” *Electr. Notes Theor. Comput. Sci.*, vol. 111, pp. 5–12, 01 2005.

[45] M. Veanes, C. Campbell, W. Schulte, and N. Tillmann, “Online testing with model programs,” vol. 30, 09 2005, pp. 273–282.

[46] E. Bernard, B. Legeard, X. Luck, and F. Peureux, “Generation of test sequences from formal specifications: Gsm 11-11 standard case study,” *Software: Practice and Experience*, vol. 34, no. 10, pp. 915–948, 2004. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.597

[47] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner, “One evaluation of model-based testing and its automation,” in *Proceedings of the 27th international conference on Software engineering - ICSE ’05*, ser. ICSE ’05. ACM Press, 2005. [Online]. Available: http://dx.doi.org/10.1145/1062455.1062529

[48] A. Sinaga and M. Hutapea, “Implementation of model based testing for testing kawn subscriptions manager application,” *Jurnal Komputer dan Informatika*, vol. 11, pp. 174–184, 10 2023.

[49] H. Shu, H. Lv, K. Liu, K. Yuan, and X. Tang, “Test scenarios construction based on combinatorial testing strategy for automated vehicles,” *IEEE Access*, vol. 9, pp. 115 019–115 029, 2021.

[50] F. G. d. O. Neto, F. Dobslaw, and R. Feldt, “Using mutation testing to measure behavioural test diversity,” in *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2020, pp. 254–263.

[51] M. Betka and S. Wagner, “Extreme mutation testing in practice: An industrial case study,” in *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*, 2021, pp. 113–116.

[52] A. Parsai and S. Demeyer, “Comparing mutation coverage against branch coverage in an industrial setting,” *International Journal on Software Tools for Technology Transfer*, vol. 22, no. 4, pp. 365–388, Aug. 2020.

[53] J. D. Hagar, T. L. Wissink, D. R. Kuhn, and R. N. Kacker, “Introducing combinatorial testing in a large organization,” *Computer*, vol. 48, no. 4, pp. 64–72, 2015.

[54] D. R. Kuhn, R. Bryce, F. Duan, L. S. Ghandehari, Y. Lei, and R. N. Kacker, "Combinatorial testing: Theory and practice," *Advances in computers*, vol. 99, pp. 1–66, 2015.

[55] J. Petke, M. B. Cohen, M. Harman, and S. Yoo, "Practical combinatorial interaction testing: Empirical findings on efficiency and early fault detection," *IEEE Transactions on Software Engineering*, vol. 41, no. 9, pp. 901–924, 2015.

[56] L. Hu, W. E. Wong, D. R. Kuhn, and R. N. Kacker, "How does combinatorial testing perform in the real world: an empirical study," *Empirical Software Engineering*, vol. 25, pp. 2661–2693, 2020.

[57] X. Li, R. Gao, W. E. Wong, C. Yang, and D. Li, "Applying combinatorial testing in industrial settings," in *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2016, pp. 53–60.

Appendix A

Acronyms

ACTS advanced combinatorial testing system. 22, 24, 26, 28, 29, 35

CIT Combinatorial Interaction Testing. 2, 11, 19–24, 26–30, 34, 35, 37, 40, 41, 43–46, 49

CRM Customer Relationship Management. 30

ERP Enterprise Resource Planning. 30, 49

GUI graphic user interface. 15, 26

MBT Model-Based Testing. 1, 2, 15, 16, 18–20, 43, 46, 47

ROI Return on Investment. 13, 14

SDLC software development life cycle. 9

SUT system under test. 7, 15, 16, 19, 21, 26, 36, 40, 47

UML Unified Modeling Language. 19

UX user experience. 1

Appendix B
Used Software

The following software was used in the writing of this thesis in compliance with the *Methodological guideline No. 5/2023*[1]:

- ChatGPT (OpenAI)[2] for input on text style and for rephrasing suggestions
- Grammarly[3] as a grammar and spell checker
- QuillBot[4] as a grammar checker and for rephrasing suggestions

[1] https://www.cvut.cz/sites/default/files/content/d1dc93cd-5894-4521-b799-c7e715d3c59e/en/20231003-methodological-guideline-no-52023.pdf
[2] https://chat.openai.com
[3] https://www.grammarly.com
[4] https://quillbot.com/