Improving Automated Software Testing while re-engineering legacy systems in the absence of documentation

A thesis submitted to the
College of Graduate and Postdoctoral Studies
in partial fulfillment of the requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Hamid Khodabandehloo

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis. Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan S7N 5C9
Canada

Or

Dean
College of Graduate and Postdoctoral Studies
116 Thorvaldson Building, 110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan S7N 5C9
Canada

Abstract

Legacy software systems are essential assets that contain an organizations' valuable business logic. Because of outdated technologies and methods used in these systems, they are challenging to maintain and expand. Therefore, organizations need to decide whether to redevelop or re-engineer the legacy system. Although in most cases, re-engineering is the safer and less expensive choice, it has risks such as failure to meet the expected quality and delays due to testing blockades. These risks are even more severe when the legacy system does not have adequate documentation. A comprehensive testing strategy, which includes automated tests and reliable test cases, can substantially reduce the risks. To mitigate the hazards associated with re-engineering, we have conducted three studies in this thesis to improve the testing process.

Our first study introduces a new testing model for the re-engineering process and investigates test automation solutions to detect defects in the early re-engineering stages. We implemented this model on the Cold Region Hydrological Model (CRHM) application and discovered bugs that would not likely have been found manually. Although this approach helped us discover great numbers of software defects, designing test cases is very time-consuming due to the lack of documentation, especially for large systems. Therefore, in our second study, we investigated an approach to generate test cases from user footprints automatically. To do this, we extended an existing tool to collect user actions and legacy system reactions, including database and file system changes. Then we analyzed the data based on the order of user actions and time of them and generated human-readable test cases. Our evaluation shows that this approach can detect more bugs than other existing tools. Moreover, the test cases generated using this approach contain detailed oracles that make them suitable for both black-box and white-box testing. Many scientific legacy systems such as CRHM are data-driven; they take large amounts of data as input and produce massive data after applying mathematical models. Applying test cases and finding bugs is more demanding when we are dealing with large amounts of data. Hence in our third study, we created a comparative visualization tool (ComVis) to compare a legacy system's output after each change. Visualization helps testers to find data issues resulting from newly introduced bugs. Twenty participants took part in a user study in which they were asked to find data issued using ComVis and embedded CRHM visualization tool. Our user study shows that ComVis can find 51% more data issues than embedded visualization tools in the legacy system can. Also, results from the NASA-TLX assessment and thematic analysis of open-ended questions about each task show users prefer to use ComVis over the built-in visualization tool. We believe our introduced approaches and developed systems will significantly reduce the risks associated with the re-engineering process.

Acknowledgments

My sincere gratitude and appreciation go to my supervisors Professor Chanchal Roy and Professor Kevin Schneider. This work would have been impossible without their continuous support, vision, and immense knowledge. Expressions of profound gratitude go to my committee members, Professor Manishankar Mondal, Professor Madison Klarkowski and Professor Mamun Abdullah, for their valuable comments, encouragement, and insights.

I would also wish to express my gratefulness to Professor Banani Roy for extended discussions, valuable feedback, and passionate participation, which significantly contributed to my study.

I express my heartiest gratitude to my dearest friends, Lawrence and Eileen Thoners. I learned the meaning of true friendship alongside them.

Thanks to all of the Software Research Lab members with whom I have had the opportunity to grow as a researcher. I am thankful to the Department of Computer Science at the University of Saskatchewan for their generous financial assistance through scholarships, which helped me focus more deeply on my thesis work.

Finally, I wish to thank my parents for their love and support throughout my life. Thank you both for giving me the strength and courage to reach my potentials and chase my dreams. I also would like to thank my siblings for their constant support and motivation all along the way. And last but not least, I would like to convey my greatest love and gratitude to my beloved wife, Maryam, who has stood by me through all my efforts. She has always been supportive, no matter what. I am thankful to have her in my life.

To Maryam,

my amazing wife,

whose love and passion has brightened my life

Contents

List of Tables

List of Abbreviations

AIF	Application Independent Functionalities
AUT	Application Under Test
CRHM	The Cold Regions Hydrological Model
GPS	Global Positioning Systems
GUI	Graphical User Interface
POM	Page Object Mode
SDLC	Software Development Life Cycle
STLC	Software Testing Life Cycle
TA	Thematic Analysis
TDD	Test-Driven Developmen

List of Figures

1 Introduction

This chapter presents a brief introduction to this thesis. Section 1.1 manifests the primary motivation of the thesis. In Section 1.2, we define the problems, and in Section 1.3 we describe our contributions towards addressing the problems. The accepted and under review publications related to the thesis are listed in Section 1.4. Finally, Section 1.5 outlines the remaining chapters of the thesis.

1.1 Motivation

The quality of a software product is one of the primary measures of the success of a product. Software developers, quality assurance engineers, project managers and other stakeholders work together to produce software systems that are easy to comprehend and use but have only minimum number of bugs. Many researchers have worked to understand what is the share of software testing in software production and maintenance. One study [45] estimated that between 25% to 50% of project resources had been dedicated to software testing to ensure the quality of the software product. The world Quality Report for 2015-2016 [21] announced that about 35% of IT project budget are allocated to quality assurance in the world. This number is 41% for North America. The expense and energy that the quality assurance process takes from software production teams lead them to automate this process. Test automation, alongside other approaches such as early testing, reduces the cost of software products.

However, much work has been done on the software test automation in the Software Development Life Cycle (SDLC)[44, 95, 19, 95], but only a limited amount of work has been devoted to trying to automate software testing during re-engineering or modernization of a legacy system [32, 38]. Re-engineering is a substitute for redevelopment to reduce costs [88], but it comes with risks such as performance loss, current staff rejection, delay due to architectural mismatches, delay due to testing bottlenecks, and failure to achieve quality goals [89]. From the five mentioned risks, the latter two are directly related to software testing, and early performance tests could predict performance loss. There are several ways to reduce the risks related to testing: better test planning, more inclusive user involvement in the acceptance test process, and most important, test automation, which supports regression testing in a way that could discover discrepancies between the old and new program versions and demonstrate functional sameness [89]. Moreover, several legacy systems' re-engineering experience confirms that testing uses 50% to 75% of the total resources [89], which is about 25 percentage points more than when a system is developed from scratch. Considering the risks associated with weak test strategies and the burden of testing in the re-engineering process, it seems test automation is a requirement of a successful and cost-effective legacy system re-engineering.

One of the most valuable software artifacts that evolve with software is its documentation. While re-engineering a legacy system, documents such as user manuals, diagrams, and natural language test cases can help development teams understand the legacy system's business and functionality faster and better.

When it comes to testing, test cases are very valuable as they can reduce testing time, increase accuracy, and improve test coverage. Some studies focus on test case reuse in the software re-engineering process [42, 43]. They discuss how to migrate test cases so they will be beneficial in the renovated systems as well as which test cases to migrate. However, when no test case exists for the legacy system, test cases should be generated based on the legacy system's functionality. Newly generated test cases should be tested on both old and new systems to ensure that the re-engineering process does not affect the new system's functionality. Generating test cases manually is a time-consuming and labor-intensive task. Many researchers have studied automatic test case generation. Some of them focus on the capture and reply technique [41, 51] which is appropriate for legacy systems. In this approach, user actions are captured, and test cases are generated based on the collected data. This approach's drawback is that the generated test cases do not include potent test oracles (expected results); hence, they cannot detect many of the bugs. Therefore, in our study, we are interested in developing an approach to generate test cases with detailed oracles.

Test automation and test case generation can improve the quality of software testing, as discussed. However, re-engineering data-driven legacy systems that work with large amounts of data raises more concerns, such as data validity after the re-engineering process. One of the significant matters that Sneed et al. [89] have found in their research investigating the re-engineering of 13 legacy systems was a demand for regression test tools to find the difference between the legacy and renovated system. To mitigate this problem, we introduce a data visualization approach that can help testers and developers find data discrepancies while re-engineering data-driven applications.

1.2 Research Problems

Automated testing, automatic test case generation, and visualization approaches are used vastly in the Software Development Life Cycle (SDLC) in order to produce high-quality software. However, the usage of these techniques in the re-engineering process is limited. In the following, we discuss three research problems that to improve quality of testing while re-engineering a legacy system.

- First, the Software Testing Life Cycle (SDLC) is tailored for developing software products from scratch. Development teams use testing models such as V-Model [8] to ensure the quality metrics have been met. Testing models have been improved over time, and different versions have been introduced for different purposes [57, 85, 58]. Similar to testing models, test automation techniques are more focused on new software product development, which does not address re-engineering issues such as result sameness checking. Thus, we notice a lack of study on introducing a model for automated testing while re-engineering to improve the quality of the final product.

- Second, the model that the previous research problem is designed for is helpful when the legacy system is not large. When we have an enterprise legacy system with no test cases, manually extracting test cases is a tedious task. Several previous studies have attempted to address this issue using different

techniques [28, 54, 55, 71]. However, none of them addressed the issue of test case oracles. Hence, we observe a shortage of techniques to generate test cases with detailed oracles for legacy systems.

- Third, the combination of test automation and test case generation is not sufficient for testing while re-engineering data-driven systems. As data-driven systems produce large amounts of data, to determine if the renovated system is working as well as the old system is necessary. Some research has been done to test data-driven applications [2, 48, 66], but none of them use the benefits of interactive visualization in finding data issues. Therefore, we find a lack of work in visualizing the difference between outputs of legacy and renovated systems to find defects.

1.3 Addressing Research Problems

The previous section outlined three research problems associated with test automation in the process of re-engineering a legacy system. In this section, we explain studies that we have carried out to address the mentioned issues.

1.3.1 Study 1: An Automated Testing Approach While Re-engineering Legacy Systems

This work introduces a new testing model tailored to the re-engineering process. This model includes steps to deconstruct the legacy system's functionality and reconstruction steps to generate test cases. A test automation framework is also offered to handle automating test processes in two levels: Unit tests and UI tests.

1.3.2 Study 2: Analyzing User Footprint Data for Generating GUI Test Case with Oracles

In this study, we collect and analyze users and AUT's footprint to generate test cases with oracles. User actions, AUT status change, database, and file system change data form a comprehensive dataset that is used in this analysis. The data analysis process identifies a set of candidate test cases using n-gram and time of the actions. Results show that the n-gram is accurate in finding test cases with high number of test steps. Mean accuracy for tests between 6 to 8 steps is 61%, which means this analysis identifies 61% of the actual test cases. On the other hand, time analysis is better in finding test cases with fewer steps. This analysis is able to find 42.8% of real test cases. Test cases generated in this study include natural language steps and oracles that can be used as a document for the system.

1.3.3 Study 3: ComVis: A Comparative Visualization Approach to Test Data-Driven Applications

To solve the data discrepancy issues between the old and renovated system, in this study, we introduce a comparative visualization tool (ComVis) to visualize differences between two data files (one from the legacy system and the other from the renovated system). ComVis is web-based, too, and can be used for any two sets of flat files. We conducted a user study to analyze the effectiveness of ComVis over embedded visualization in CRHM. Results from 20 participants showed that ComVis could find significantly more issues. We also performed a thematical analysis of user answers and identified the tool's positive and negative aspects.

1.4 Publications

- **Hamid Khodabandehloo**, Banani Roy, Manishankar Mondal, Chanchal Roy, Kevin Schneider. Analyzing An Automated Testing Approach While Modernizing Legacy Systems. 2021 IEEE 28th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2021.
- **Hamid Khodabandehloo**, Banani Roy. ComVis: A Comparative Visualization Approach to Test Data-Driven Applications. 30th International Symposium on Software Testing and Analysis (ISSTA). 2021. (Under review)
- **Hamid Khodabandehloo**, Banani Roy. Analyzing User Footprint Data for Generating GUI Test Case with Oracles. 30th International Symposium on Software Testing and Analysis (ISSTA). IEEE, 2021. (Under review).

1.5 Thesis Outline

This thesis is organized in six chapters. In Chapter 1, we discussed a brief introduction to the present thesis. Chapter 2 discusses the required background and technologies for this thesis. Chapters 3, 4, and 5 describe our first, second, and third study, respectively. Finally, Chapter 6 concludes the thesis with an overall summary and discusses future research opportunities.

2 Background

In this chapter, we provide a discussion of the background and required technologies of this thesis. In Section 2.1, we discuss different types of software tests, including functional and non-functional testing. In Section 2.2, we present an overview of software re-engineering and the reasons behind it. Next, Section 2.3 discusses the tools and technologies required for this thesis. Finally, we explain our subject system, the Cold Regions Hydrological Model (CRHM), in Section 2.4.

2.1 Software testing

Software testing is an iterative process in the software development life cycle (SDLC) to ensure that the current version matches expected requirements. Defect detection is the primary goal of software testing, and software testers make sure they find as many defects as possible. Other software testing goals can be measuring the quality of the product, providing confidence, and program analysis to prevent product failures [30]. Software testing can be categorized into two major types: functional and non-functional.

2.1.1 Functional testing

Functional testing focuses on the correctness of a system's functionality by testing the output of a module against its input. Functional testing is usually considered a black-box testing technique where the object's inner composition is not known to the tester. Functional testing is performed at different levels for different purposes. Next, we discuss different types of functional tests.

Unit test

Testing the smallest testable component of a software product is called unit testing. A unit test usually tests functions, methods, and classes to guarantee that the component works as expected. This type of test is typically performed in development time by software developers. In Test-Driven Development (TDD), developers create a test method first and then develop the primary method to pass the test. Listing 2.1 shows a simple example of a unit test in a C++ calculator program. In this example, the test method in line 11 tests the *add* method, and the test method in line 17 tests the *mul* method.

Integration test

In software products, a group of components integrates to make a subsystem. By unit testing, we make sure that each component is working fine, while in the Integration test, we guarantee that interaction between components works as expected. In the calculator example, a user can get a wrong answer for a calculation such as $a + b \times c$ due to user interface problems even though, the *Add* and *Mul* unit tests have passed.

System test

After ensuring that subsystems are working fine using the integration test, it is time to check if the system fulfills the system requirements. In the calculator example, if the system requirement is that calculate $(a+b)\times c$ typing $a+b\times c$ but $a+(b\times c)$ is being calculated, a bug has been introduced. These type of bugs are not detectable using unit test nor integration test and a system test is required to discover this bug.

```
#include "calc.hpp"
int add(int op1, int op2) {
    return op1 + op2;
}
int mul(int op1, int op2) {
    return op1 * op2;
}

#include <gtest/gtest.h>
#include "calc.hpp"
TEST(CalcTest, Add) {
    ASSERT_EQ(2, add(1, 1));
    ASSERT_EQ(1, add(-1, 2));
    ASSERT_EQ(-2, add(-1, -1));
    ASSERT_EQ(0, add(-1, 1));
}
TEST(CalcTest, Mul) {
    ASSERT_EQ(10, sub(2, 5));
    ASSERT_EQ(-10, sub(-2, 5));
    ASSERT_EQ(10, sub(-2, -5));
    ASSERT_EQ(0, sub(0, 5));
}
int main(int argc, char **argv) {
    testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

Listing 2.1: C++ Unit test example

User acceptance test

End-users of a software product should be satisfied with the product they are using and for which they probably paid. A user acceptance test is necessary to make sure the product meets the expectations of the end-users. In the calculator example, the user can have trouble using the product due to an unwieldy design. The user acceptance test is usually conducted by a real user or a user representative before handing the product to all the users.

Regression test

When a change in the form of a bug fix or new feature is made on the software, it can introduce new unwanted defects in other parts of the system. To guarantee the quality of the whole system in addition to testing the fixed bug or the new feature, we need to test other parts of the system as well. This test is called a regression test and has different levels depending on the change. In some cases, testing the whole system is required, while if the change is not very major, retesting a subset of test cases is sufficient.

All the mentioned tests require a test case. A test case is a software artifact that provides a step-by-step instruction to perform a test, giving expected results (test oracles) for each step and the entire test case. A software product has test cases for all its functionalities in an ideal situation, and testers test them either manually or automatically and report bugs in case of any discrepancy.

2.1.2 Non-functional testing

Contrary to functional testing that focuses on a software system's behavior, in non-functional testing, the main concentration is on how a system operates. For instance, a system's satisfactory performance is a non-functional requirement and is tested by a performance test. There are more than ten varieties of non-functional tests, such as document tests, penetration tests, and recovery tests. In this section, we elaborate on performance test.

Performance test

We call an application performant or well-performing when it lets users complete an assigned task without a delay, lag, or irritation [65]. The test ensuring that an application is performant is called a performance test. To discover performance-related issues, several measures and standards have been introduced. To measure performance, the following key performance indicators (KPIs) are considered [65]:

- Availability: the amount of time that a system is accessible by users.
- Response time: the amount of time that it takes to respond to a user request.
- Throughput: the number of application events such as database read and write in a time span.
- Utilization: the portion of a resource such as RAM theoretical capacity.

There is no single guideline to set limitations for the mentioned KPIs, and it depends on the application and the user who is operating it. GL Martin et al. [56] analyzed the decrease of response time to investigate user's productivity. The result of this study shows that response time greater than 15 seconds for a simple task is not tolerable by end-users and requires a redesign of the product. Response time of 4 to 15 seconds is tolerable, but it will hinder productivity and cause frustration. A time span of 2 to 4 seconds is bearable, but it is considered a long delay for users' tasks requiring remembering information. Many applications with a

response time of less than 2 seconds are considered as performant systems. Some applications, such as photo editing applications, need less than a second to satisfy end users. Finally, applications such as computer games need decisecond (less than 0.1 seconds) response time to make users accept the system.

2.1.3 Test Automation

Software testing is an expensive process and test automation can reduce testing costs by up to 80% and improve quality of the product [26]. Test automation is a development task that can be categorized to code level and user interface (UI) level categories.

Code level automation

Code-level test automation mostly refers to unit testing. Automated unit testing is performed using a framework that isolates each unit and tests it. One of the most frequently used unit testing frameworks is xUnit family. xUnit family share the same architectural design and is covered in many programming languages such as Java (JUnit), C (CUnit), C++ (CppUnit), and Python (pyUnit). There are several unit test frameworks for every language. Figure 2.1 shows the architecture of xUnit family.

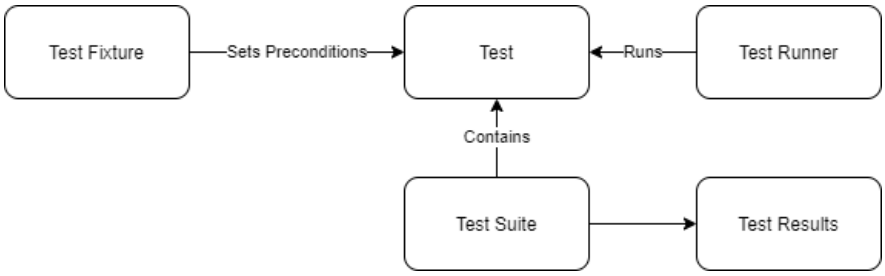


Figure 2.1: xUnit test framework architecture

In this design, each test is in a *test suit* and every test reads its preconditions and data from a class called *test fixture*; then an executable called *test runner* runs the test. Finally, the result of the test is sent to *test result* class.

UI level automation

Our intention in the UI level test automation is to perform integration, system test and user acceptance test through the product's user interface. There are several frameworks for each platform that let automation test developers automate test cases. For example, *Selenium*[1] is widely used in web-based application test automation, *Appium*[2] for mobile and *Winium*[3] and *Microsoft Coded UI* [4] for Windows applications. These

[1]https://www.selenium.dev/
[2]http://appium.io/
[3]https://github.com/2gis/Winium
[4]https://docs.microsoft.com/en-us/visualstudio/test/use-ui-automation-to-test-your-code

frameworks provide methods and classes that assist developers to mimic users' behavior. Listing 2.2 shows an example that implements a login test case on a web application.

For large projects, test automation teams usually use design patterns such as Page Object Model (POM) to make them easy to maintain and change. In this design pattern, every page in the software has a corresponding object that keeps actions and attributes of the page in itself.

```
from selenium import webdriver
driver = webdriver.Chrome()

driver.get ("URL")
driver.find_element_by_id("ID").send_keys("username")
driver.find_element_by_id ("ID").send_keys("password")
driver.find_element_by_id("submit").click()

self.assertTrue(driver.find_element_by_link_text("Logout"),"Logout
     link")
```

Listing 2.2: Login test case implemented using Selenium

2.2 Software re-engineering

Alteration and reconstruction of a software product without changing the functionality of it is called software re-engineering [17]. Re-engineering improves the software and legacy systems to make it more efficient and effective. Software developed using old technologies and maintained by many different people over time is called legacy systems[76]. Maintenance and on-going evolution of legacy systems become expensive due to technological changes and human resources' unavailability to perform the changes. When a legacy system has a high business value, it can be a candidate for re-engineering [50]. Moreover, if the changeability and level of encapsulation used in the legacy system are low, re-engineering can be justified compared to the improvement and reconfiguration of the system [76].
Discussing legacy systems, we need to take software migration and modernization into account. These phrases, alongside software re-engineering, are used interchangeably in the literature, but they are not exactly the same. Next, we will discuss each of these terms.

Software Migration

Software migration is the process of moving a system from one platform or environment to another platform or environment. This task can be undertaken for various reasons: lower costs, better performance, and supportability of the new platform or environment. The main concern in software migration is replicating the system in a new platform or environment without change to the system's functionality or structure. Migrating to could platforms is an example of software migration in which the functionality of the system and structure of the system remains the same, and only environment-related code is changed.

Software Modernization

Unlike software migration and re-engineering, the modernization process involves improvements to current features of a legacy system. Therefore, functionalities may change, or new functionalities be added to the system. For example, in a mobile application's modernization process, the login module can be improved so users can use touch id to login. In many projects, migration and re-engineering are considered as prior steps to modernization.

2.3 Tools and Technologies

Google test

Google test[5], also known as *gtest* is a xUnit base unit test framework for C++ programming language. Google test is known for its reliability and maintainability. Features such as the ability to disable each test, adding custom assertions and XML test reports makes gtest a strong candidate when compared to other C++ unit test frameworks. Moreover, using gtest enables developers to use *Google test Mocking (gMock)*[6] which is used to develop mock classes to test APIs and Interfaces that are used in the code.

Microsoft Coded UI

Coded UI is a UI automation framework developed and maintained by Microsoft that can be installed as a part of a Visual Studio IDE. Coded UI helps developers automate functional tests of UI controls and verify that the whole application is working as expected. One of this framework's features is that the developer can manually perform the test, and *Coded UI Test Builder* records the actions and generates the code for the test automation. However, generated code may not be proper to use directly in complicated software tests even though it can reduce development time.

Winium

Winium, similar to Coded UI, is a framework to automate functional UI tests on Windows-based applications. Winium is a Selenium based framework that makes it more flexible compared to Coded UI and supports a wide variety of programming languages.

Windows Application Driver

Windows Application Driver, also known as *WinAppDriver* is a tool to support Winium test automation on Windows applications. This tool helps developers to explore UI elements of an application and find attributes of the UI elements. For example, Winium needs attributes such as name, Id, or Xpath of a button to click on

[5]https://github.com/google/googletest
[6]https://github.com/google/googletest/tree/master/googlemock

it. *WinAppDriver* provides this information when the user clicks on the element. To record a history of user actions, we have updated *WinAppDriver* to gather all the user actions on a CSV file. We call the changed application *WinAppRecorder*.

2.4 The Cold Regions Hydrological Model (CRHM) application

We used CRHM application [23] in this thesis as our subject system in all three studies. This Windows-based application is used by more than 100 hydrology experts to model and analyze water behavior in cold regions. For instance, this application is employed to predict water balance for some research basins in the prairies [25]. Researchers collect various data such as snow depth and density, precipitation, and air temperature and estimate a basin's streamflow using CRHM application [25]. CRHM is developed using Borland C++, and it is being re-engineered using Microsoft Visual Studio C++. Figure 2.2 shows a screenshot of the application.

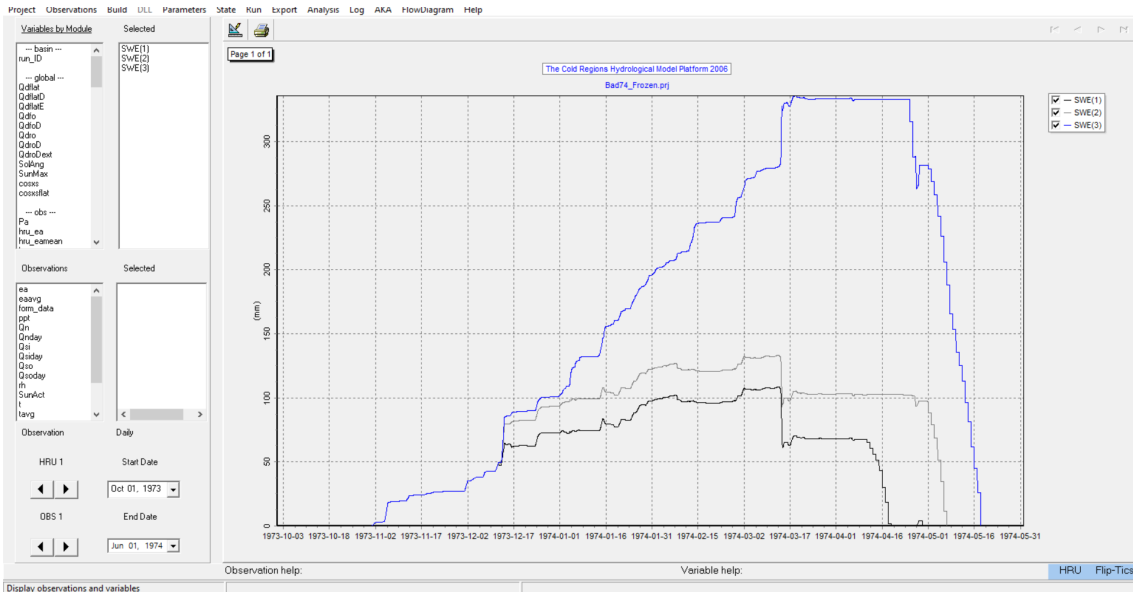


Figure 2.2: CRHM application.

3 An Automated Testing Approach While Re-engineering Legacy Systems

Testing is one of the most crucial disciplines in the Software Development Life Cycle (SDLC). Having a proper testing method results in a high-quality software system. Testing is extremely important when re-engineering a legacy system that is still being used by many users. When users use the renovated system, they expect exactly the same behavior as the old system. Therefore, testing an application which is being re-engineered is more challenging and should be more accurate. For this purpose, we designed a model to ensure that the development during re-engineering produces fewer bugs, and bugs will be identified and fixed in shorter periods of time. In this model, we test every method in the development time using Google test unit tests. Having unit tests contributes to the fact that our current changes do not affect the other parts of the code-base. On the other hand, we mimic user behavior using automated UI tests after each release on both the old and new versions and compare the results to make sure the renovated version is working accordingly. We applied this testing method on the Cold Regions Hydrological Model (CRHM) application, which is being migrated from Borland C++ to Microsoft Visual C++, and we were able to find some bugs which were impossible to detect manually. This process enhances reliability, accuracy, and speed of testing.

3.1 Introduction

Software testing is an iterative task during the software production process. For generating a better result, it is better to start testing soon. Detecting and fixing bugs in the early steps of software production reduces software development and maintenance costs. There are several methods to develop and test a software product. According to Balaji et al. [8], it is better to use V-Model in case we need to validate every phase and start testing from the early stages of development. It has been suggested to use this model if the software requirements do not change many times [8]. As we are re-engineering a legacy system and requirements will not change, we decided to design a model based on V-Model concepts that are tailored regarding re-engineering concerns. The whole idea behind V-Model is that software testing is as important as software development [30]. V-Model schema is shown in Figure 3.1, the left branch shows development processes, and the right branch shows the testing activities. For each level of development on the left branch, the right branch defines a corresponding test level. So, for assuring that a component is working correctly, a component test (unit test) is designed. The integration test, system test, and user acceptance test are designed to guarantee technical system design, functional system design, and requirement definitions are designed and implemented as expected. W-Model [90] proposes the same idea, including more details. According to this model, testing activities start soon enough to test the requirements before starting any other phase, and in each phase, a respected test is designed. After developing and integrating each phase, tests will be executed,

and in case of the existence of any bugs, code will be fixed to make sure the designed test is working as required. The W-Mode provides a test method for all kinds of functional tests, including user acceptance tests, system tests, integration tests, and unit tests for the software production process, emphasizing early testing. Figure 3.2 shows details of W-model.

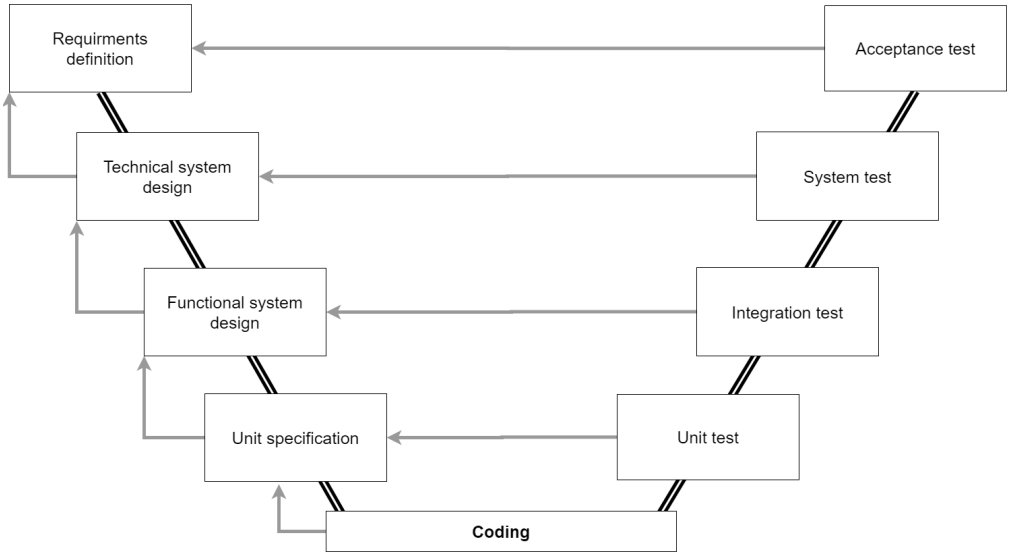


Figure 3.1: V-Mode

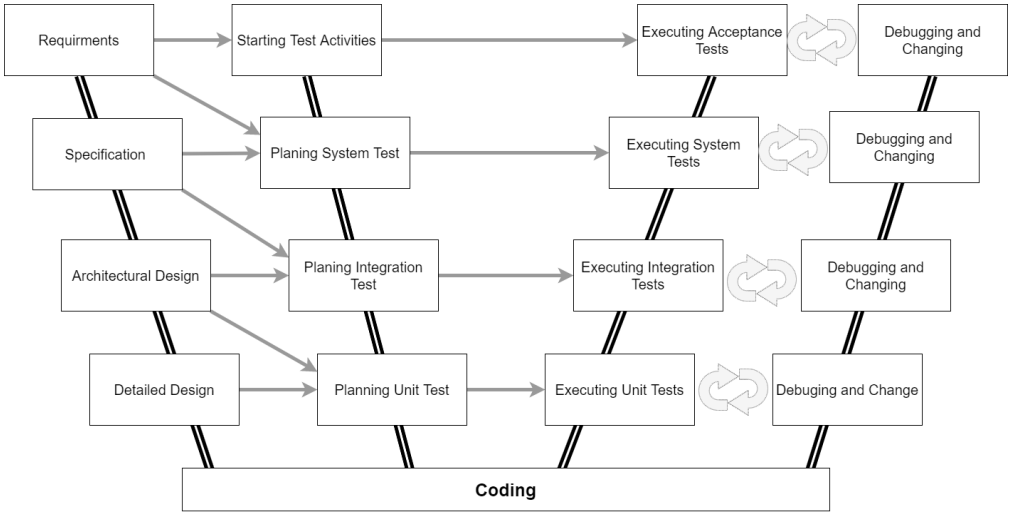


Figure 3.2: W-Model

Using models such as V-model and W model is very helpful in developing software products. However, when an existing legacy system with no development documentation is being re-engineered, more steps are required. Therefore, we propose a new model based on the V-model and W-Model when a legacy system is re-engineered. We call this new model reV-Model. A legacy system is a software that is developed and maintained by different people during its life-cycle [77]. After deciding that a legacy system needs to be re-engineered according to three factors: Business vale, Changeability, and Encapsulation [77], the whole

process of testing is followed as proposed in this model. We also implemented this process on the Cold Regions Hydrological Model (CRHM) application and observed the results. We automated testing this application in two ways. First, we developed automated unit tests using Google Test framework. Second, we automated user acceptance, system, and integration tests using Microsoft Coded UI [36] (using C#), which leads to early bug detection. The process for all kinds of tests will be explained in Section 3.4.

Next, in this chapter, we discuss the motivation behind this work in Section 3.2. Section 3.3 discusses our new proposed model. Section 3.4 describes the implementation of the approach. Engineering challenges are described in 3.5. Next, Section 3.6 elaborates related work. Section 3.7 discuss the limitations of this work. Finally, we discuss the conclusion and future work in Section 3.8.

3.2 Motivation

In this section, the main motivations behind this work will be portrayed. The primary goal of this project is to provide test methods while re-engineering an existing software to ensure that end-users do not suffer from many bugs, so the transition from the old version to the new version will be easier. End-users are already using the Borland version, and if the Microsoft version does not work as well as the product they already are using, they will go back to the old version, and the whole migration will be in jeopardy. Moreover, users need to trust the new system. Having newly developed software with fewer bugs will result in faith in the product. Using the introduced W-model (Figure 3.3), we can make sure that every feature, user story, and functionality is tested and migrated to the new version.

Another challenge of re-engineering a legacy system is that users expect the same functionality with the same performance after re-engineering. In some complicated cases, the old CRHM application takes 2 or 3 days to analyze the input and show the results. Users expect the same performance if not better. For this purpose, we use the same test cases and inputs for both old and new applications and compare their running time. Therefore, if the new application works slower, we can detect it and fix it before handing it to the end-user.

The next goal is to automate tests to find bugs sooner and spend less time on repetitive manual tests. For this purpose, we are using the Google unit test framework for developing the unit tests and Microsoft Coded UI for automated user acceptance tests.

Our final goal is to have continuous delivery and continuous integration for the migration to ensure that every part of the system has been tested and delivered to the end-user on time.

3.3 V-Model for re-engineeing (reV-Model)

Our proposed model is shown in Figure 3.3. This diagram has two main branches: Deconstruction and Reconstruction. We explore the entire system in the deconstruction branch and identify features, user stories, and user functionalities. The legacy system consists of several features, and each feature contains several

user stories. Each user story consists of several user functionalities. For example, the CRHM application has features such as *Project* and *Construct*. *Project* feature contains user stories such as *Open project, Run Project, and Export project*. Every one of these user stories includes several user functionalities such as click on buttons and typing in text boxes. Reconstruction branches happen simultaneously with the structural changes of the code. In the re-engineering process, legacy code transfers to the new code; meanwhile, code structure may change, but features and user stories will not change. In this process, unit tests will be designed based on the new structure of the code. These unit tests are developed and executed after each release to make sure the new changes are not introducing bugs. Integration, system, and user acceptance tests are also designed taking code changes, user stories, and features into consideration. The result of this process is a set of human-readable test cases for each user story. A sample of test cases generated using this model is shown in Figure 3.4. Each test case has an Id, title, pre-condition, steps, and expected results. The title of the test case is extracted from the user story's name. For instance, for *Open project* user story, we have several *Open project test cases*. Steps are user actions taken from user functionalities. We also provide pre-condition, which indicates the state of the application before starting the test. Expected results describe the state of the application after running the test. Any discrepancy between the expected result and the actual result would be a bug.

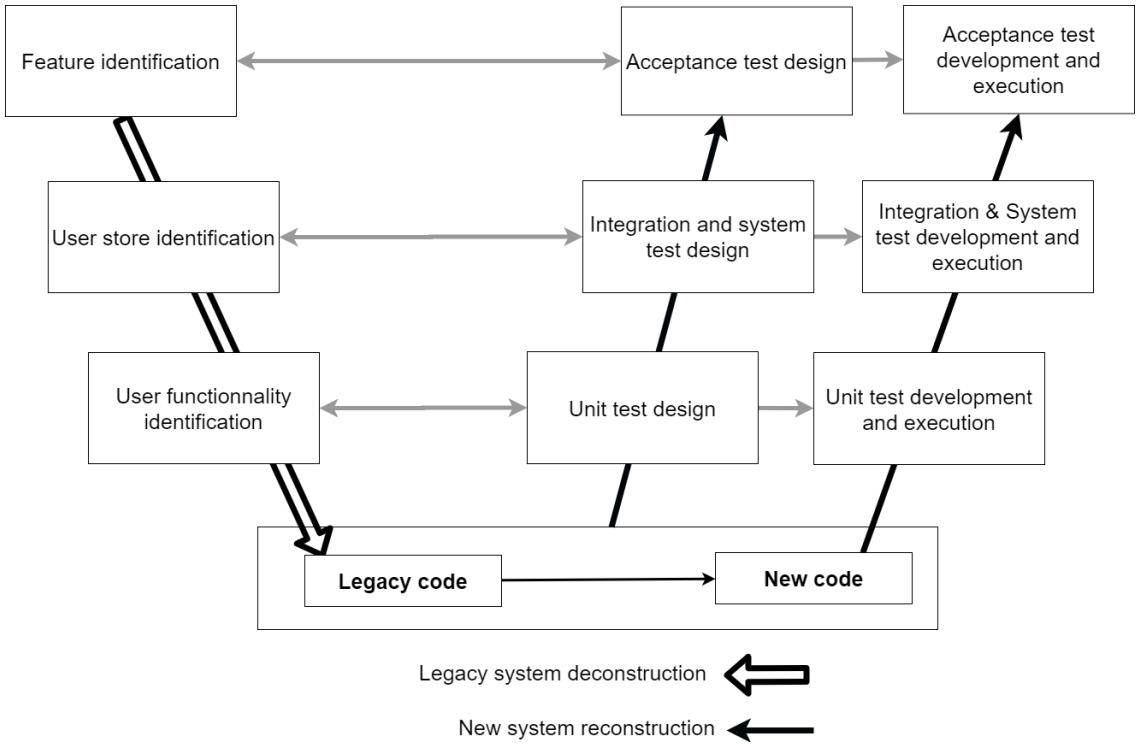


Figure 3.3: W-Model for re-engineering a legacy system.

Test Case Id	Title	Pre-Condition	Steps	Expected Result
CRHM-1-1	Open a project	1- The app should be opebed 2- Project File (prj) should be available 3- Observation file (obj) should be available in the same folder as prj file	1- Click on the project menu 2- Click on the open item 3- Select the prj file	Following list boxes should be filled: Variable by module Selected Observations Selected Start date and End date should be changed HRU1 and OBS1 should be enabled
CRHM-1-2	Open a project when there is no observation file	1- The app should be opebed 2- Project File (prj) should be available 3- Observation file (obj) should not be available	1- Click on the project menu 2- Click on the open item 3- Select the prj file	User should see this message: Cannot file observation file: "File Address"
CRHM-1-3	Save a Project	A project should be opened before	1- Click on the project menu and click on the save	Changes should be saved

Figure 3.4: Test Cases Sample.

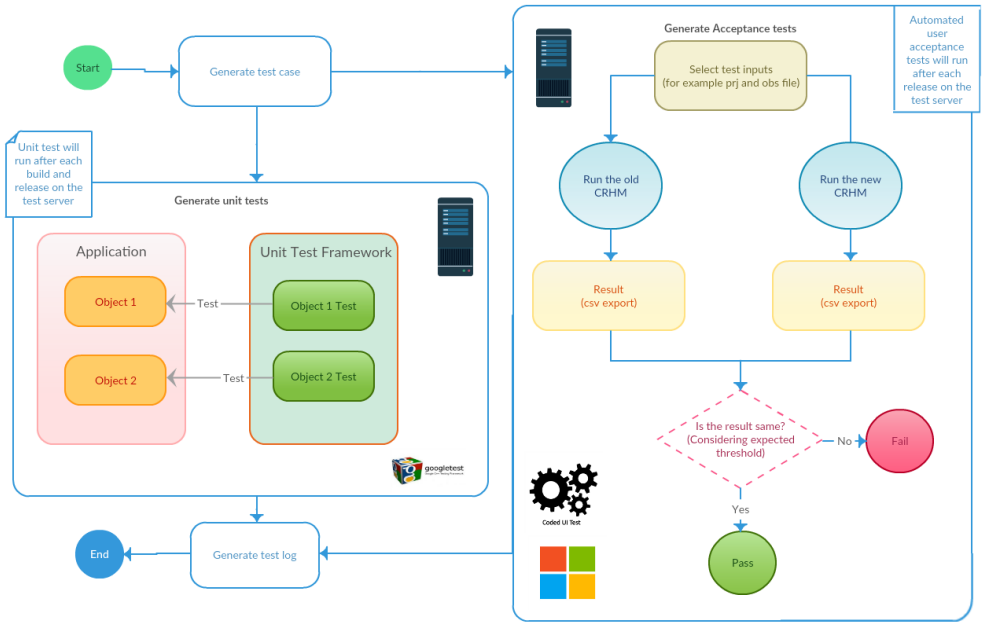


Figure 3.5: Automation test framework while re-engineering.

3.4 Implementation

In this section, the implementation of the automation test framework will be discussed. The logic behind the automation test framework is illustrated in Figure 3.5. The test case generated following the new W-Model is used as an entry point for the test. A sample test case is shown in Figure 3.4. For every automated test (unit or UI test), a test case has been designed. Having these test cases, we start developing tests. First, the unit test frameworks will be described. Second, automated UI tests will be discussed. In Sections 3.4.3 and 3.4.4, works that has been done to evaluate the performance of the system is presented. Next, we explain how the result of the tests is reported and the approach for selecting test cases for automation is explained. Finally, we discuss the limitation of this framework.

3.4.1 Unit Test Automation

Unit tests are developed and executed during the development of an application. Unit testing aims to isolate each part of the program and ensure that the individual parts are working correctly. In this project, we

used the Google test for creating our unit test framework. We have created a separate project for the unit test framework. This approach has several benefits, such as less overload on the system in the production environment and easier debugging, change, and maintenance of the framework. Figure 3.5 shows the unit test framework's schema as a part of the test framework. We have created a file for each class, and all the tests related to each class are developed in the related file. We tried to follow the same directory path for each file, so finding and editing tests become easier. All the tests are run after each build. In case a change affects another part of the code, the related unit test, which is affected, will fail. By increasing the coverage of the tests we can make sure that our new changes are not breaking other parts of the system. One of the challenges of unit tests is providing proper input for each test. In conventional unit tests, development approaches, inputs, and expected results for each test are given inside the code. This leads to a code change for different input values, which is more human work and can cause human errors. We used a CSV file to store inputs and expected value for each test case in this project. In this way, we can change the input values independent from the code. An example of a unit test written in C++ is shown in Figure 3.6. In this example, input and expected output of the *GetDateTimeInString* method is fetched from the CSV file. After running the method, the test will pass if the result is the same as the expected result. This process has been done for selected methods. (How methods are selected is discussed in Section 3.4.5)

3.4.2 UI Test Automation

After executing unit tests, it is always necessary to perform integration, system, and user acceptance tests to ensure that the system works correctly. In some cases, especially in software with many possible inputs and outputs, it is not easy to check everything manually. In such cases, we need to make the process automatic to perform it faster and more accurately. The input of the CRHM application is mostly data that has been collected hourly and contains thousands of entries. The output files are large as well, and their manual check is time-consuming and inaccurate. As the system under test is re-engineered, it is expected that

```
TEST(DateTimeToString, DateTimeAsDouble) {
    StandardConverterUtility scu = StandardConverterUtility();

    FileOperation f = FileOperation();
    //Reads input from file

    std::string input = f.getInput("DateTimeAsDouble");
    //Reads expected result from a file
    std::string output = f.getOutput("DateTimeAsDouble");
    COleDateTime dateTest =(COleDateTime)atof(input.c_str());
    EXPECT_EQ(scu.GetDateTimeInString(dateTest), output);}
```

Figure 3.6: Unit test sample.

```
public void RunAndCompare()
{

1   TestConfig tf = new TestConfig("config.xml");

2   ApplicationUnderTest.Launch(@"C:\Program Files (x86)\CRHM\CRHM.exe");
3   this.UIMap.RunAndExport(tf.ProjectFile, tf.BorlandOutputFile);

4   WindowsOperations.BringToFront("Dialog");
5   this.UIMap.OpenAndRunProject(tf.ProjectFile, tf.VsOutputFile);

6   FileOperations fo = new FileOperations();

7   Assert.IsTrue(fo.CompareTwoObsFiles
        (tf.BorlandOutputFile, tf.VsOutputFile, tf.ComparisonReportFile));

}
```

Figure 3.7: Sample of automated user acceptance test.

the output of both old and new applications will be identical. Hence, we feed the same input file to both applications and compare the results. We used Microsoft Coded UI for automating the test. We automated all the required steps to take the output file on both old and renovated applications. Then, we developed a method to compare the output files. An example of this code is shown in Figure 3.7. In this example, first, we open the old application and run the *RunAndExport* method. This method imports the required input files to run the model and export the output. All the information regarding input and output files are read from *config.xml* file. The same thing with the same input happens for the new application, and at the end, *CompareTwoObsFiles* compares the output. If both files are similar, the test will pass. This process helped us to find bugs related to the calculation of the model. We also realized that sometimes we have tiny differences that are negligible. Therefore we added a threshold to ignore those differences. This process is shown in Figure 3.5. The figure's caption contains the details of the process.

3.4.3 Performance Test

The performance loss is one of the significant risks of re-engineering. As a part of the user acceptance test, we tracked the time of the test executions. We compared execution time for each run on both applications. To ensure the only influential factor is the code, not other factors such as operating system and hardware, tests for both applications are done in the same environment. Considering these tests, we did not observe any meaningful difference, and this implies that the new application is working with the same performance as the old application. Performing these tests will help us make sure we are not changing anything that makes the performance of the system unsatisfactory.

3.4.4 Test Result Report

After each test execution, we need to explore the test results and determine which tests failed and passed. Apart from the pass/fail result, we are interested in test execution time as well. In the case of a test failure, we need to investigate the reason and report a bug. Therefore, we are using Microsoft Visual Studio test explorer to see which tests pass and which tests fail. For unit tests, expected results, real results, and elapsed time are given in case of failure. By investigating the difference between the real result and the expected result, we can determine the issue's root cause and report it as a bug. An operating system command line (Windows or Linux) can be used as an alternative for Visual Studio test explorer. For UI tests, a message describing why the test failed, elapsed time, and stack trace are produced in case of failure. For example, when control is hidden for any reason, and the code cannot click on it, the message shown to the user is:"Cannot perform 'Click' on the hidden control". Moreover, we log all the differences between the output of the old and the new application on each run so testers and developers can detect the deference manually to find the cause of the issue.

3.4.5 How to Decide What To Automate

Automating tests is a time-consuming process and requires substantial upfront investment [82]. It is not feasible or necessary to develop automated tests for every part of the system. Consequently, we need factors to prioritize test cases. For unit tests, we decided to select methods called the most in the system because any defect on these methods will affect other parts of the system. The other factor for unit tests is how likely it is that the method will be changed or refactored in the future. If a method is changed many times, we increase the priority as every change may introduce a new bug. Developers and testers evaluated both these factors and did not develop a systematic way to prioritize them. For user acceptance tests, we prioritize based on the importance of the test case and whether we have enough unit tests for that part or not. As we extracted the system's features and stories, we can assume which ones are used more frequently, so we try to automate those parts first. The second priority is for parts of the system that we do not have unit tests. We intend to make sure that parts without unit tests are working correctly by performing UI tests.

3.5 Engineering Challenges

During test development, we encountered several challenges. In this section, we are going to elaborate on these challenges. The first one from unit testing is that we have huge methods migrated from the legacy system which do not follow design principles. For example *RunClick* method of CRHM application is a massive method with many variables and possible execution paths, and developing unit tests with acceptable coverage is very difficult. Therefore, we need to re-design this method and break it down into several methods. After performing this refactoring, we will be able to develop test cases with acceptable coverage. In order to

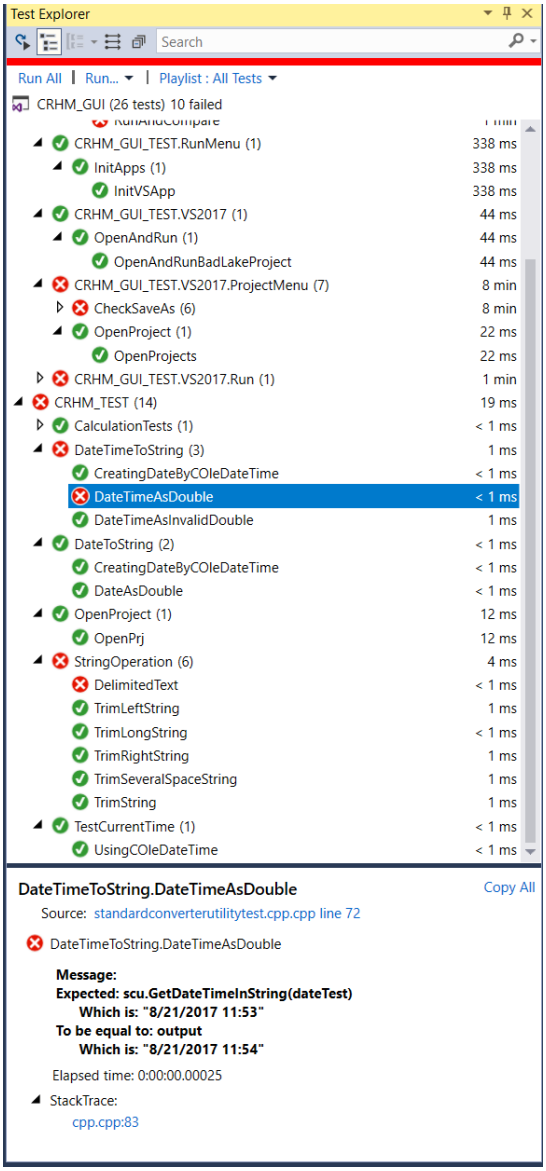


Figure 3.8: Test results sample.

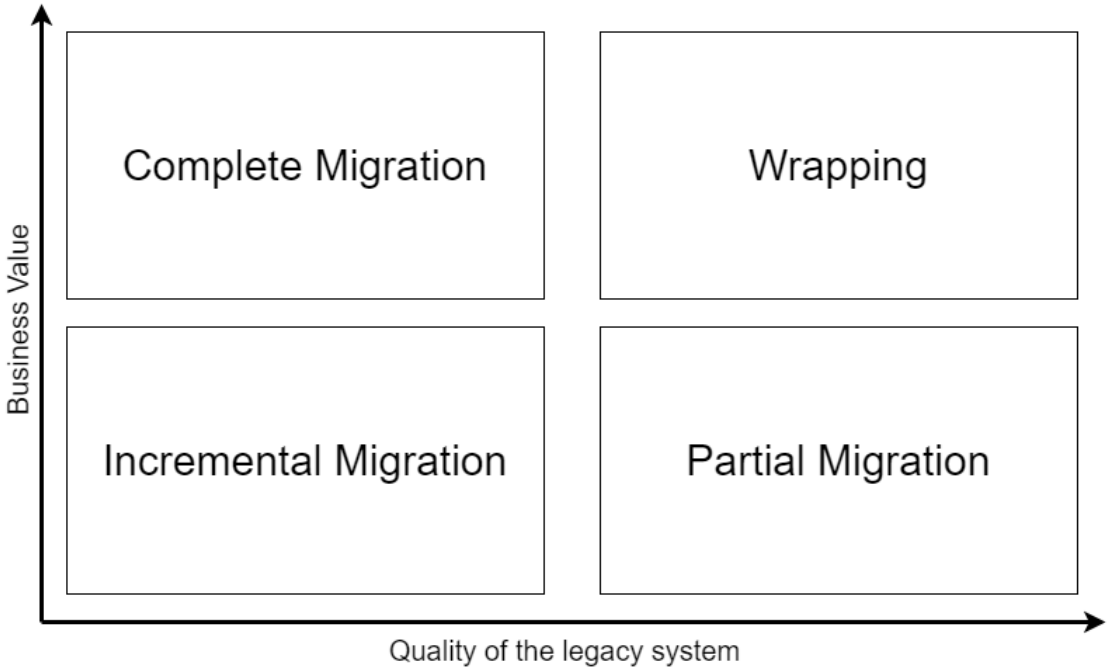


Figure 3.9: Sample of automated user acceptance test

compensate for this issue, we formulated a short-term solution for this kind of method. We have developed UI tests that call those methods from the UI instead of developing a unit test to ensure these methods are working as expected. Regarding UI test challenges, performing some actions on the windows applications using Coded UI is complicated and, in some cases, impossible. For example, values of labels verification. We are trying other automation tools such as Winium [37] to overcome this issue. Considering these challenges, we were able to run the automated tests successfully and find bugs that we could not find manually.

3.6 Related Work

A lot of work has been done regarding software re-engineering, and different kinds of strategies have been introduced. Althani and Khaddaj [3] conducted a study which reviewed legacy system migration systematically. One of their significant findings is that none of the earlier works in the literature had concentrated on quality assurance at every step of the migration and re-engineering processes. Therefore, they proposed a new assessment system of legacy systems based on the legacy system's business value and quality to help teams decide how to proceed with their development process. Their assessment system is shown in Figure 3.9. Based on this assessment, if the legacy system's business value and quality are high, software teams can wrap the product in a new interface, and there is no need for re-engineering. Although they assess the quality of the existing legacy system, they do not suggest any method to improve the quality of the re-engineering process. Among the many research done on software re-engineering, only a few concentrated on software testing as a part of the work. For example, Rosenberg et al [79]. considered testing techniques and methods similar to

those used in from scratch system development, and emphasized documentation as a general requirement for testing [17]. Ducasse et al. [22] proposed a traced-based logic testing. They used execution trace as their test basis. They had a satisfactory result; however, they did not perform any user acceptance tests. Our work is different from previous studies because we try to put the user first by extracting features and user stories as our test basis. The other factor that makes our work valuable is to employ automated software testing to save time and make the tests more accurate.

3.7 Limitations

Although the proposed testing model and the test automation architecture assist testers and developers to test a legacy system while re-engineering, some tasks need to be performed manually. In this section, we explain these limitations and our plan to address them.

- First, in the proposed model, the test case generation process is a manual task. We propose a data analytics approach to generate test cases from user's footprint to mitigate this issue. This approach is covered in the next chapter of this thesis. This approach helps to determine the importance and priority of the test cases as well.
- Second, the method explained in 3.4.2 gives a pass or fail result, and for more details, testers should work on the out put data and find the data issues manually, which is a challenging task. Therefore, we propose a comparative visualization tool (ComVis) in the fifth Chapter of this thesis. This tool help testers explore data from both old and renovated versions and find differences introduced with bugs.

3.8 Summary

Using a model for testing a legacy system under re-engineering makes the testing process more disciplined. This model helps to find bugs earlier and fix them before release. This is even more important when the old system is under heavy usage, and users do not expect defects. In summary, we performed the following steps to ensure the renovated system is working as expected:

- Extracting test cases from user stories extracted from the legacy system considering structural changes.
- Developing automated unit tests and UI tests for important functions of the system.
- Developing a user acceptance test which considers the legacy system as ground truth.
- Measuring the renovated system's performance continuously to make sure the re-engineering process is not causing performance loss.

We also mentioned limitations of this work. The next two chapters focus on tackling the limitations.

4 Analyzing User Footprint Data for Generating GUI Test Case with Oracles

Test case development is an essential step in the Software Testing Life Cycle (STLC), contributing to a robust test plan either for developing new software or re-engineering a legacy system. Developing test cases for already developed systems that do not have any test cases is a time-consuming task and usually inaccurate. This issue is more relevant when a legacy system is being re-engineered because the lack of a rich test plan can jeopardize the success of the entire process. Therefore, in this chapter, we propose a new approach to generate test cases by collecting and analyzing user actions such as mouse click and keyboard events and Application Under Test (AUT) reactions such as database and file system changes. Test cases generated using this approach contain detailed oracles that assist testers to detect more bugs. Our analysis shows that the sequence and the time difference between the performed actions are beneficial factors to generate test cases from the collected data. The empirical study shows that this approach can generate test cases with oracles that can detect bugs that none of the state-of-the-art techniques can find.

4.1 Introduction

Testing a software product is essential after every set of changes before a release. Software development teams usually perform functional tests to guarantee that the new changes are not inducing new defects [103]. These tests are usually done by interacting with the Graphical User Interface(GUI) to ensure that the whole system works as expected, and new bugs are reported[75]. In this process, system test cases play an important role in testing the system properly. Test cases are considered as one of the most important artifacts of any software system. In the software industry, people in different roles come and go, and it is important to meticulously document all the steps of a test without relying only on team members' memory. System test cases are steps written in natural language that describe how to perform the test[91]. A high-quality test case should indicate the exact expected result for each test step to ensure that the system is behaving as expected (Test oracles). Designing test cases with oracles is more expensive. In return, these kinds of test cases detect bugs earlier, and a larger number of faults are detected with a smaller number of test cases [62]. Figure 3.4 shows a test case written by a professional tester and Figure 4.2 presents the same test case with oracles. It is obvious that the latter test case is capable of finding more bugs. Moreover, test cases with oracles contribute to better test automation as assertion guidelines are provided in the test case as oracles. Writing test cases similar to Figure 4.2 is a challenging task because a tester who regularly writes black box system test cases is not aware of all of the mentioned details, especially the ones mentioned in the expected results of step 6. On the other hand, developers might not remember all the details, or several developers should be included to add such an accurate expected result. So it is complicated to develop such a test case even

with the collaboration of developers and testers. The details added to the test cases in Figure 4.2 transform the black-box test case to a gray box test case. This kind of test case not only can be used by testers to test the system either manually or automatically but also it can be used by new team members as valuable documentation of the system. This chapter's primary purpose is to provide gray box test cases, which will enable testing the software more accurately both manually and automatically.

Even though test case development can save time and money for teams by early and more bug detection, many software teams postpone test case development. They usually start writing test cases when the product is already in use, and many bugs have been reported. At this point, developers and testers feel that they need test cases to prevent bugs from reoccurring. This can happen for several reasons, such as a limited budget at the project's primary phases. Testing software, either a recently developed software or a legacy system that does not have any test cases, is costly, and fewer bugs are likely to be detected without test cases.

Despite the fact that there have been many advances in GUI testing, such as different techniques for GUI test automation such as Selenium for web, Appium for mobile applications, and Winium for desktop applications, the number of tools that contribute to automatic test case generation is minimal. The result is not satisfying, and this is even worse for test cases with test oracles[75].

In this paper, we propose a novel solution to generate test cases with oracles by collecting and analyzing user actions and Application Under Test's (AUT) data, including database and file system changes. This approach contributes to the production of high-quality test cases and generates test oracles, which lead to earlier and more bug detection. We developed a tool to collect required data from users and AUT, analyzed the data, and generated test cases with oracles.

Our proposed technique outperforms existing approaches in several ways. First, test oracles help testers to be more accurate and find more bugs. Second, as the generated test cases using this approach include a natural language description, it can be a valuable document for testers and developers to understand the system faster and easier. Next, generated test cases are based on real user actions. Therefore all the test cases point to a real user usage of the system. Lastly, collecting user actions helps us understand how often the user uses different parts of a system that can lead to determining the priority of test cases.

The main contributions of our work are:

1. A tool to collect user actions, database and file system changes for desktop applications.
2. An approach to generate system GUI test cases with test oracles.
3. Insights that help to have an abstract overview of user behaviour on the AUT.

This chapter is organized in 6 sections. Section 4.2 describes the applications under test that we have used to elaborate and evaluate our approach. In Section 4.3 our approach is explained. Section 4.3.2 provides information on data analysis, the most significant features of the data, and details about users' behavior. In Section 4.4 presents the results and evaluates the approach. Finally, Section 4.6 outlines the conclusion and implications for future research.

Title: Return a Book		**Priority:** High
Steps		
1	Open the application.	
2	Login with a valid user.	
3	On the main menu click on the "Return a Book".	
4	Select a user from "Users" dropdown.	
5	Select a book from "Books to return" grid view.	
6	Click on the "Return" button.	
Expected Results:		
The book should be removed from user's "Books to return" list and be available for other users to hold.		

Figure 4.1: Test case written by a professional tester.

Title: Return a Book		**Priority:** High
Steps:		**Expected Results:**
1	Open the application.	
2	Login with a valid user.	"Main" window should be opened.
3	On the main menu click on the "Return a Book".	"Return" window should be opened.
4	Select a user from "Users" dropdown.	
5	Select a book from "Books to return" grid view.	
6	Click on the "Return" button.	• In the Holds table "ReturnDate" column should be updated to current time and "Returned" column should be updated to True. • "Held" and "Taken" columns in the Books table should be updated to false.
Expected Results:		
The book should be removed from user's "Books to return" list and be available for other users to hold.		

Figure 4.2: Test case with oracle.

4.2 Application Under Test (AUT)

To evaluate our approach, we need applications with a user interface, storing data in a relational database, and working with the file system. Several applications are used in the literature as AUT. For instance, Universal Password Manager (UPM) [87], a password manager, and Buddi [12], a personal financing application, is used by Mariani et al. [54]. Rachota [47], a personal task manager application, is used by Arlt et al. [6]. Each of these applications is either works with a database or files system. None of them uses both a file system and a relational database. To explain our approach better and provide unified examples, we developed a windows-based *book reservation application* that uses both file system and database. As we are developers of the book reservation application and are aware of all the features and test cases, we used data collected from this application to test and improve our approach before applying it to other applications. The applications used in our evaluation section are the book reservation application, three mentioned applications used in the literature (UPM, Rachota, and Buddi) and CRHM windows application.

```
1  public bool ReturnABook(int bookId, string username)
2  {
3  using (AUTContext ctx = new AUTContext())
4  {
5  this.Book = Book.GetBookById(ctx, bookId);
6  this.User = User.GetUserByUserName(ctx, username);
7  Hold hold = getHold(ctx, bookId, username);
8  try
9  {
10 //Update the return date of the book
11 hold.updateReturnDate(DateTime.Now);
12 hold.Returned = true;
13
14 //Book is in the library and ready to be holded again
15 hold.Book.Held = false;
16 hold.Book.Taken = false;
17 ctx.SaveChanges();
18 return true;
19 }
20 catch (Exception e)
21 {
22 ExceptionLogger.LogError(e.ToString());
23 return false;
24 }}}
```

Listing 4.1: C# code for returning a book. This Method is used when a book is returned by user. Four columns are changed in two tables.

We also designed a set of *ideal test cases* with oracles for book reservation application to be able to compare our results with them. Then we asked a professional software tester to review the test cases to make sure that those are reliable and accurate.

The functionality of the book reservation application is to manage book lending of a library. This windows application contains several features such as sign up, log in, create, update, delete, hold, pick up, and return a book. Users are also able to export the history of books and users to a file. A user can log in after signing up and perform all the actions through a graphical user interface. All the data of the application is stored in a relational database.

Listing 4.1 shows the corresponding code that is executed in the sixth step of the test case in Figure 4.2, which is one of ten test cases that has been designed for this application. This code changes four columns of two tables in the database.

4.3 Proposed Approach

The approach starts with collecting data from three sources:

1. User actions, such as click and typing.
2. Database changes such as insert, update, and delete on tables.
3. File system changes such as create, change, and delete files.

Next, data has been assessed to make sure the collected data reflects users' behavior. Issues such as missing or duplicate data have been detected in the assessment phase. All the gathered data are merged after performing data cleaning, such as removing duplicate rows and handling missing values. Next, data has been analyzed, and important features to identify the test cases are recognized. Finally, test cases have been generated, considering the features. An abstract architectural overview of the approach is shown in Figure 4.3. This section is organized in four subsections, which will describe every step in detail.

4.3.1 Data Collection

We asked two users to use the book reservation application and the other four applications for collecting real data. We provided the users with the required information and privileges to work with the software. Then, we recorded all the actions and their consequences on the database and file system.

User Action

For user actions, we collect eight characteristics of action: Time of the action, control type (button, menu item, and textbox), control name, action (right-click and left click), window name, window lineage (sequence of windows that lead to the current window), description and source (User, database or file system). A

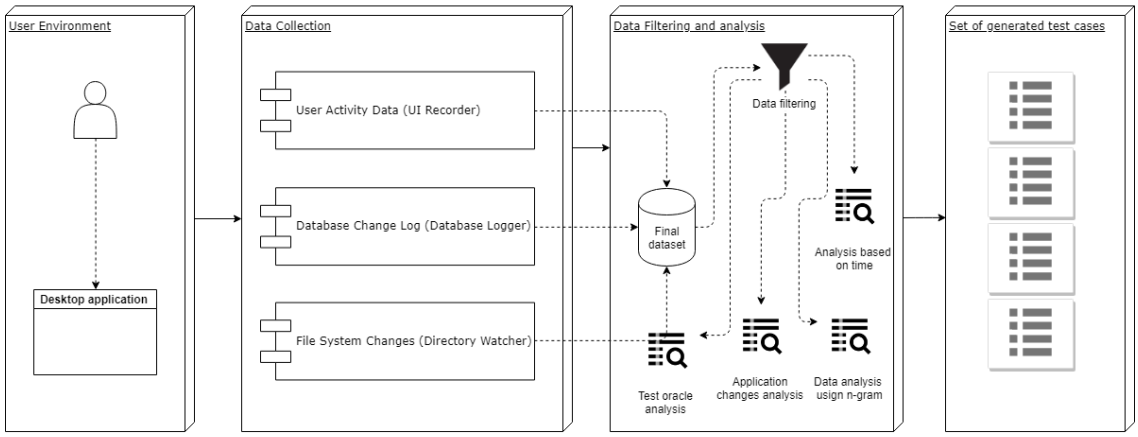


Figure 4.3: Architecture of the approach.

sample of the data collected from user is shown in Table 4.1. This data is collected when user has performed a left click on the *save button* in the *book window* at the mentioned *time*. The description column is a human readable sentence generated from other columns that is used as the step text. We refer to a user action as a pair $<t,a>$ where t is the time of the action and a is the action. In the previous example, t is the time of the right click action and a is the left click on the save button on the book window.

In order to record user actions we extended *WinAppDriver* tool. This tool helps test developers inspect elements of a windows application and get information such as XPath and an element's name. Using the source code, we are able to fetch all characteristics of a user action. We have changed this tool to employ it for our use. Using this application, all the windows elements and actions are recorded. We added four main features to *WinAppDriver*. First, as we only need the AUT's actions, not the entire operating system, we have restricted the tool to collect the actions in the AUT. Second, we collect all the recorded information in a *csv* file with mentioned details in Table 4.1. Third, we added a database connection feature, so if an application has a relational database, we can connect to it and record database changes. Finally, we added a file system change recorder, which records changes in a target directory. Users can select an AUT to record on the modified tool, and by start recording all the user actions, database, and file system changes will be saved in separated csv files. These files are used as a source of data for generating test cases.

Changes in the application

Performing an action in the application changes state of the application. For example, by clicking on the login button on the login page main window will open, or if the username or password is not valid, a message will be shown. We refer to these changes as a pair $<t,w>$ where t is the time of the change and w is the change. This type of change is detectable by comparing two consequent user actions. Therefore, two user actions $<t,a_i>$ and $<t+1,a_j>$ can introduce a change in the application $<t,w>$. This status change is an indication of a test oracle. In the previous example, opening the main window or seeing an error message can be considered the login action's expected result.

Time	Control Type	Control Name	Action	Window	Window Lineage	Description	Source
2019-09-14 13:40:10.127	Button	Login	LeftClick	Login	Login,	LeftClick on the Login Button on the Login window	User
2019-09-14 13:49:18.859	Button	Save	LeftClick	Book	Login,Main, Book	LeftClick on the Save Button on the Book window	User

Table 4.1: A sample of the data collected from user action

Database Changes

When a user performs an action that influences the database, we collect this information: time, operation type, table name, the key value of the table, column name, old value, and new value. The old value is always null for an insert operation, and for delete, the new value is always null. We also generate a description, which is a human-readable sentence describing the action. A sample of the data collected from the database is presented in Table 4.2. This record shows that *author column* in *book* table with primary key of *25* has been *updated* from *Hemingway* to *Dickens* at the specified *time*. This row is recorded immediately after the left click on the save button mentioned in Table 4.1. We model every database change as a pair $<t,d>$ where t is time and d is the database change. All the database changes are saved in a table and exported in a csv file. This data is merged with the data that has been gathered from user actions.

Time	Operation Type	Table Name	Key Key	Column Name	Old Value	New Value	Description	Source
2019-09-14 13:40:10.824	Update	Users	Id=4	LastLogin	2019-09-14 10:45:33.624	2019-09-14 13:40:10.824	LastLogin column in User table updated from 10:45:33.624 to 13:40:10.824	DB
2019-09-14 13:49:19.154	Update	Book	Id=25	Author	Hemingway	Dickens	Author column in Book table updated from Hemingway to Dickens	DB

Table 4.2: A sample of the data collected from database

File system Changes

For getting file changes, we ask users to select a directory as their workspace. Then we monitor that directory recursively and save all the activities. This data contains the time of the change, type of the change (create a new file, delete a file, rename, size change, modify date change, last access time change), old value, new value, name, and path file. We model every file change as a pair $<t,f>$ where t is time and f is the file change. This data is saved in the form of a *csv* file.

Finally, we assess, clean and merge all the data from three sources and sort them by time. This dataset contains actions performed by users and changed in the database and file system.

4.3.2 Data Analysis

Merging all the collected data and sorting them by time provides a set of user actions with a human-readable description and consequences of those actions on the application, database, and file system. This data can be considered as a log of user actions. As these actions have been performed on a stable version of an AUT we can consider the very immediate impact of a user action on the application, database, or file system as expected result of that action. For example, when a user action such as $<t,a>$ has been recorded and the immediate record after this user action is a database change such as $<t+1,d>$ we consider $<t+1,d>$ as expected result of $<t,a>$. Having this data, we have two major challenges. First, what is the expected result of a user action? Second, how to identify the start and endpoint of a test case. In this subsection, we discuss what features in data can help us to tackle these problems.

Window Change

When a user action results in a change of the current window, it means the action is important in terms of the test. In this condition, the state of the application has been changed, and another window has been opened. We can consider this change as an expected result of a user action. For instance, when the user name and password are correct, by clicking on the login button on the login form, the main window opens, and if this does not happen, it is a bug. So we have two consequent user actions with different windows. $<t+1, a_j>$ is immediate action of $<t, a_i>$ so we can consider $<t+1, w>$ which is a window change, as expected behaviour of $<t, a_i>$. In the login example, when a user clicks on the login button, the window changes from the Login to the Main window. We call the step that causes this window to change a **compound step**. Compound step always include at least one test oracle. We show this kind of compound steps in the form of an array $<t, a, w>$. A compound step can have more than one oracle, such as when an action has impact on both application status and database and we show it in this way: $<t, a, w, d>$.

This data helps us understand the oracle of a step and contributes to understanding how the user is using the system. Figure 4.4 shows the window change graph. This graph reveals how the user is using the system. Each node shows a window. The size of the window indicated counts of actions the user has performed in the

window. For example, the user performs more actions in the Hold book form than in the other forms. Every edge indicates a path from a window to another window. The thickness of the edge shows that the path has been used by the user more frequently. For example, users went from the main form to the booking form more frequently. These graphs help to understand which form and action are more important. This graph also shows all the ways that a are being used to open a form. For example, there are three ways to open the return form.

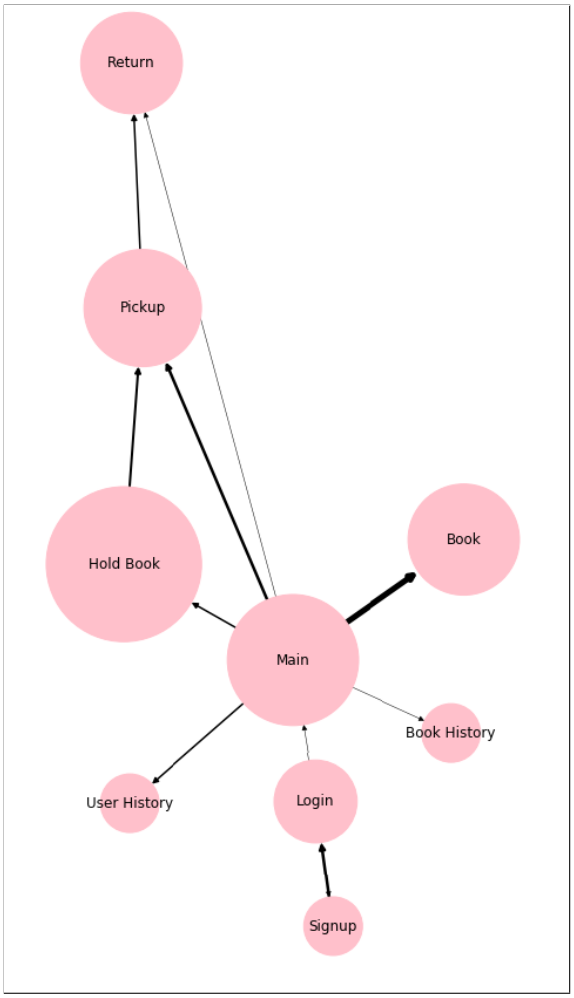


Figure 4.4: User form change graph.

Source Change

In the merged data set we know which row belongs to which source (user, database or file system). Therefore, when we see a source change in the data we can conclude that a user action resulted in a change to the database or file system. In other words, when we have a database change $<t, d>$ immediately after a user action $<t, a>$, we consider $<t, d>$ as expected result of $<t, a>$. For example, when a user clicks on the login button on the login form, last login time changes in the user table. We consider "LastLogin column updated from

[oldValue] to [newValue]" as expected result of "Click on login button" step. We consider this source change as another kind of compound step and we show it in the form of an array $<t, a. d>$. The same thing applies for file system change and we show it in similar way: $<t, a. f$.

From window and source changes we have three different types of compound steps:

- When we have a change in the application after an action: $<t, a. w>$
- When we have a database change after an action: $<t, a. d>$
- When we have a file system change after an action: $<t, a. f>$

Not all the user actions have oracles. Some of the user actions are independent without any effect on the database or file system. We call these user actions **simple steps**. All the GUI system test cases are combination of simple and compound steps. Figure A.15 shows a login window. Simple test steps and compound steps for this window are as follows:

- Type user name: $<t, a>$
- Type user password: $<t+1, a>$
- Click on the login button: $<t+2, a, w, d>$

In this example, the combination of two simple steps and one compound step, generates a GUI system test case with oracles for login.

Although extracting compound steps is necessary for generating oracles, it is not enough for generating GUI system test cases. In addition to oracles, we need to know the start and endpoint of a test case. In the login example, we should find a pattern to identify when the test started and ended. For this purpose, we analyze the similarity of consecutive steps and the time difference between steps.

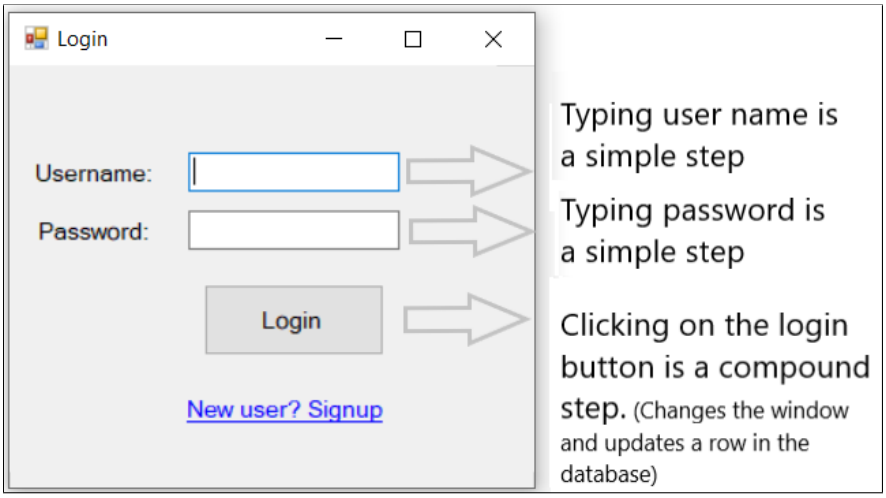


Figure 4.5: Login window simple steps and compound steps.

Similarity of steps

In window and source change analysis, we identified compound steps. In this section, we would like to know which compound and simple steps are executed together most frequently. Identifying steps that are frequently repeated together is a promising approach indicating that these steps are probably in the same test case. For example, in a login form user always types username and password and then clicks on the login button, these three steps most of the time are next to each other. To extract these related steps, we applied *n-gram* [13] to find groups of steps that are employed by a user more frequently. Identifying the number of n for n-gram is challenging. To understand what range of n is appropriate, we applied n-gram with n=2 to 10 to group the related test steps. N-gram with n=2 means that two steps are executed consecutively more than once, n=3 means three steps were followed after each other more than once, and so on. We call consecutive steps that are repeated frequently, *candidate test cases*. Figure 4.6 shows four distinct user actions (a_1, a_2, a_3, a_4) repeated 15 times in different orders. In this example, different candidate test cases with different lengths have been recognized (four candidate test cases with two steps, three candidate test cases with three steps, and two candidate test cases with four steps). By limiting the number of test cases to more than two, all the candidate test cases with a length of two are not eliminated. Figure 4.7 shows the number of candidate test cases for each n (from 2 to 10) in our dataset. For smaller numbers of n, we have more candidate test cases. With increasing n, the number of candidate test cases decreases. For n greater than 8, there are no candidate test cases. In other words, test cases in this system have a maximum number of seven steps, including simple and compound steps. For n = 2, we have many repeated consecutive steps. For instance, when a user types a user name and password on the login form, these two steps are bundled together, but these two steps do not form an independent test case. Considering candidate test cases with n = 2 as a test case does not make sense as it is too short. Considering n from 3 to 8, we were able to find some reasonable candidate test cases. Some examples of the test cases are shown in Table 4.3 for n = 3, 4, and 7.

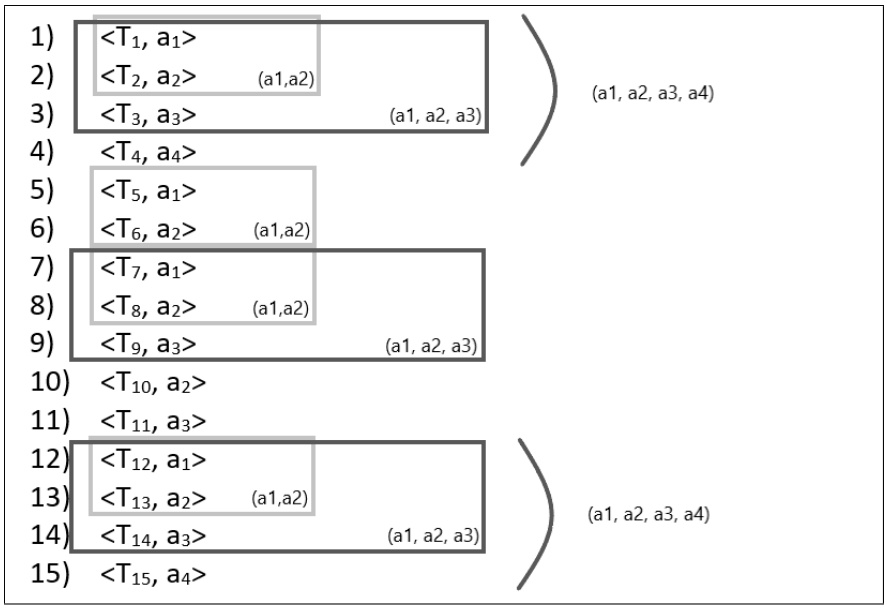


Figure 4.6: User action sequence example.

n	Step	Oracle	Summary
3	LeftClick on the Edit on the Books window		
	LeftClick on the Save Button on the Books window	Column Title in table Books should update from test to test2, Id=28	Edit a book
	LeftClick on the Close Button on the Books window		
4	LeftClick on the Edit on the Login window		
	LeftClick on the Login Button on the Login window	Main window should be opened	Login and open
		LastLogin column in User table should update	Book window
	LeftClick on the Books Button on the Main window	Book window should be opened	
	LeftClick on the Edit on the Books window		
7	LeftClick on the Edit on the Books window		
	LeftClick on the Save Button on the Books window	Column Author in table Books should update from test to test2, Id=23	
	LeftClick on the Close Button on the Books window		Edit a book
	LeftClick on the Close Button on the Main window		log out and login again
	LeftClick on the Edit on the Login window		and open Book window
	LeftClick on the Login Button on the Login window	Main window should be opened	
		LastLogin column in User table should update	
	LeftClick on the Books Button on the Main window	Book window should be opened	

Table 4.3: Test cases extracted from user footprint data using n-gram (n = 3, 4 and 7)

Candidate test cases that resulting after applying n-gram with n = k are set of bundled test steps that most probably include test cases with k steps. Total number of candidate test cases CTS_{total} can be calculated using equation 4.1 (m is maximum number of n in n-gram that we want to explore).

$$CTS_{total} = \sum_{n=3}^{n=m} CTS_n \tag{4.1}$$

We calculated the total number of candidate test cases for a dataset with 504 user actions on the book reservation system AUT. The result shows we have 76 candidate test cases. We also manually explored all the candidate test cases and realized only 10 of the candidate test cases are real test cases. The rest are test steps repeated together more frequently but do not form a reasonable test case. To handle this problem, we increased *depth* of the n-gram. It means we removed candidate test cases that are repeated less than three times (*d=3*). We repeated this for larger numbers of *d*s. For larger numbers of *d*, CTS_{total} decreases. For instance, in Figure 4.6 by changing d from two to three, a set of repeated actions less than three times has been omitted. It means (a_1, a_2, a_3, a_4) is not considered a candidate test case anymore. Candidate test cases resulted from larger numbers of d are more accurate. For our dataset d numbers more than eight did not introduce any candidate test cases. This number can be different for datasets with more data.

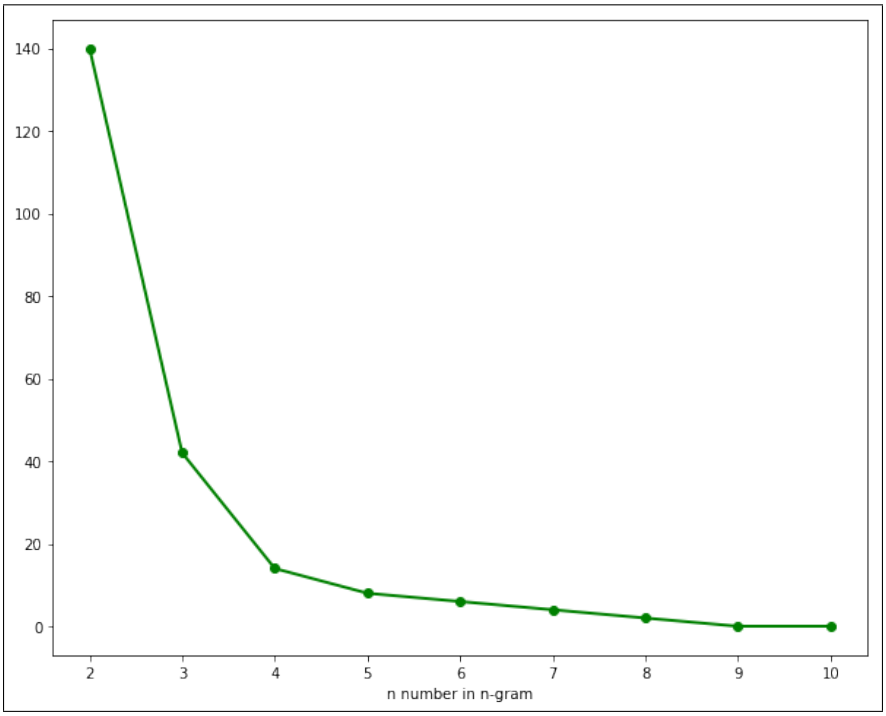
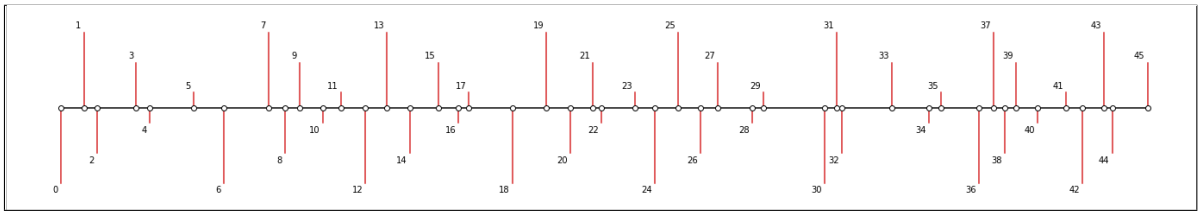


Figure 4.7: N-gram.

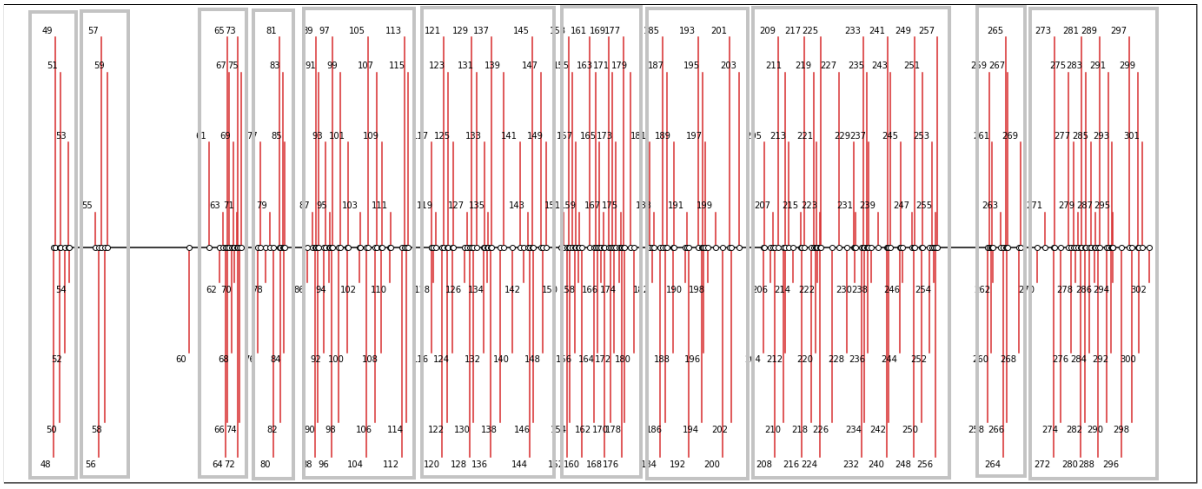
Time Analysis

One of our assumptions is that time difference between user actions can help identify test cases. In other words, when users stop working with the AUT for a while, we can infer that their work is completed, and

we can call it a test case. To understand the validity of this assumption, we analyzed the time difference between each user action. This time shows how long it takes before users to perform the next action. Figure 4.8 (a) shows 46 user actions performed within about four minutes. Minimum time difference in this sequence is one second, and the maximum is 13 seconds. The average time difference between each action is about 5 seconds. We analyzed these 46 user actions and realized that there are seven different test cases. The time difference between the last steps of the test case and the next user action in all test cases is more than average (5 seconds), and the average for time difference after the last step of the test case with the next user action is 7.5 seconds. The average time difference for steps inside test cases is 4.1 seconds. This shows that the time difference between steps can be a promising feature to find the start and end of test cases. Figure 4.8 (b) shows the timeline for 255 user actions in about 30 minutes. There are obvious gaps between user actions in this graph, which are shown with gray boxes. Finding a number for time difference that can be used as an indicator that user actions are in the same test case is challenging. In the next section, we evaluate effectiveness of each of the approaches.



(a) Four minutes time window



(b) 30 minutes time window

Figure 4.8: User actions timeline

4.4 Evaluation

In order to understand the effectiveness of our approach, we examine the following research questions.

RQ1:How accurate is the data collection tool?

RQ2:How accurate are test cases generated using n-gram?

RQ3:How accurate are test cases generated using time difference?

4.4.1 Experimental Process and Results

Accuracy of data collection (RQ1):

It is imperative to make sure data gathered from user actions is accurate and that users' major actions, database, and file system changes are recorded. For this purpose, we captured one user's screen while he was working with AUTs. Then, we compared the number of actual actions and recorded actions. We also analyzed each action's effect on the database and file system and compared it with recorded data. The result of this evaluation is shown in Table 4.4.

	application name	Number of actual actions	Number of recorded action	Accuracy
User action	Book	182	157	86.3%
Database changes		49	49	100%
File system changes		3	3	100%
User action	Buddi	103	81	86.3%
Database changes		-	-	-
File system changes		4	4	100%
User action	Rachota	112	86	76.8%
Database changes		-	-	-
File system changes		3	3	100%
User action	UPM	121	101	83.5%
Database changes				
File system changes		2	2	100%
User action	CRHM	252	211	83.7%
Database changes		-	-	-
File system changes		4	4	100%

Table 4.4: Accuracy of collected data

This data shows that all the database and file system changes are recorded, and nothing is missed. On the other hand, 13.7% of user actions have not been recorded. The tool fails to record user actions when the user performs actions very fast. We removed data related to missed user actions on both database and file system change dataset to reflect this phenomenon. The fact that database changes and file changes happen instantly after each user action helped us understand which record is related to a missed user action.

Accuracy of test cases generated using n-gram (RQ2):

To understand the accuracy of the test cases generated using n-gram we manually examined generated candidate test cases. For filtering candidate test cases to reasonable test cases, we had two major rules. First, the first step on the test case must be an appropriate step to start. For instance, if a candidate test case starts with *click on the save button*, this candidate test case will not be considered a real test case as some other steps are required to reach this step. Second, the test case's last step must be an appropriate step to finish the test case. For instance, if a test case finishes with a step such as *click on a row of a data grid view*, the step will be filtered out as some more steps are required to consider the test case a complete test case. Each iteration with different n generated candidate test cases with length of n. For smaller n, we have more but shorter candidate test cases. Table 4.5 shows accuracy of generated test cases for each n.

n	Number of candidate test cases	Number of actual test cases	Accuracy
3	50	9	18%
4	16	4	25%
5	8	2	25%
6	6	2	33.3%
7	4	2	50%
8	2	2	100%

Table 4.5: Accuracy of test cases applying n-gram

Generally, by increasing n, we have a better set of candidate test cases. Candidate test cases generated for smaller n are noisier. For example, a valid test case such as login failure due to incorrect username or password has been generated here, but many other consecutive steps that happen to be executed together are here as well. However, for larger n, probability of having random steps is lower.

Accuracy of test cases generated using time difference analysis (RQ3):

To examine the accuracy of test cases generated from time analysis, we evaluated them manually. We considered 255 user actions shown in Figure 4.8 and evaluated them manually. We considered time difference greater than the mean time difference to be a test case identifier. It means we took steps between two large time differences as a test case. We also did not consider test cases fewer than three steps. Considering these criteria, we found 28 test cases. We applied the same rules used for n-gram here and evaluated if a test case

is a valid test case. Rules applied for n-gram are not sufficient for this method. There is a possibility that the first step and the last step of a test case make sense, but the steps between are not sensible. Therefore, in addition to the rules used for n-gram, we added another rule to make sure that the test cases are internally integrated. The third rule is to ensure that the sequence of steps makes sense. As mentioned in Section 4.4, the data collection tool failed to collect all the data. So if a step has been missed on a test set, we exclude it from valid test cases. Applying all three rules, we identified 10 test cases out of 28 (35%) as valid test cases. Table 4.6 shows details of these test cases.

Step count	Number of candidate test cases	Number of actual test cases	Accuracy
3-6	21	9	42.8%
7-10	5	1	20%
More than 10	2	0	0%
Total	28	10	35%

Table 4.6: Accuracy of test cases applying time analysis

For test cases with more than ten steps, time analysis did not work at all. None of the candidate test cases with more than ten steps are real test cases. For shorter candidate test cases, this method can find one real test case out of five. This method performs better for short test cases where the length of test cases is three to five steps. Short test cases such as login, sign up, and user history is found using this method.

4.4.2 Discussion

As explained in Section 4.4.1, we have attempted to extract GUI test cases using user footprint data. Results that were achieved on two approaches, n-gram, and time analysis, are promising. N-gram is appropriate to find longer test cases more accurately, while time analysis is appropriate for shorter test cases. As long as there is little research that has been focused on test cases with oracles, we can only compare our approach to AUGUSTO [54] which identifies popular functionalities such as login and CRUD (Create, Read, Update and delete) operations. Although AUGUSTO is able to detect bugs related to popular functionalities, it fails to provide oracles such as expected changes on the database and file system. On the other hand, our approach is not limited to popular functionalities and can identify a complicated sequence of user actions and generate test cases. However, while our approach can find accurate test cases, it needs data that should be collected from users and the application under test. This is a limitation of our method. It means it cannot be applied to applications without user data. It requires data collection, which can be challenging for applications that are already in use in terms of security. End-users may find this process not trustworthy enough as their actions are being recorded. One way to alleviate limitations associated with this process is to provide a

trusted environment and ask end-users to perform their daily actions. This way, users will be more confident that there are no risks in collecting their data. Collecting user's data is useful for test case generation and is beneficial for understanding users' behavior. Graphs similar to Figure 4.4 can be generated for different applications and guide testers and developers to where the system focuses more.

4.4.3 Threats to Validity

We identified two threat to this research's validity. First,concerns our data collection method. We asked two users to work with the system while we were collecting the data. Both users were new to the applications and had not used the system before. Data could be different for expert users familiar with the system, and they may work faster. To limit this threat's effect, we verbally explained the functionality of the system to users before data collection.

Second, we used *WinAppDriver* to record user actions. This tool could miss some user actions if users perform the task very quickly. Therefore, some of the user actions are not collected. We have integrated the entire data to reflect the missing user action. This approach may cause some issues in the data. We minimized the possibility of missing a test case step by recording user actions for the same scenario several times.

4.5 Related Work

There have been a great number of studies on generating GUI test cases automatically [54, 81, 24, 16, 7, 75, 53, 14, 4, 98, 105, 5, 96]. We can categorize previous works into two major groups; first, tools that automate the process using only the source code or executable files of the AUT without any external resource, and second, tools that use external resources and the AUT resources.

In the first category, tools such as Testar [94] uses random techniques to generate test cases. Model-based techniques are more popular in this category. Tools such as AUGUSTO [54], GUITAR [71] and AutoBalckTest [55] use graph models of the GUI to generate automated test case generation. Finally, tools such as BARAD [28] follow a systematic technique. BARAD employs symbolic execution to extract data flow and event flow in the AUT. These tools have access to limited test oracles. For example, GUITAR relies on application crashes as a test oracle, and when an application crashes, it reports a bug. AUGUSTO tries to identify popular functionalities such as login and CRUD operations in an AUT, which is better in terms of generating test cases with oracles, but it is limited to popular functionalities.

On the other hand, other works consider more artifacts and generate test cases automatically. Some tools [7, 16] take snapshots of the application and analyze the elements of the system. Others take the system requirement documents and UML use cases as input and try to generate test cases from them [35, 10, 93]. Chang et al. [14] used computer vision to test applications through their graphical user interface. Their approach is only able to assert visual test oracles. These tools, similar to the other category, have access to

limited test oracles.

Brooks at al. [11] used usage profile and proposed a probabilistic model of the AUT. They were able to report the acceptable amount of bugs, but their approach, like the research mentioned previously, lacks suitable oracles. As we are using the user's data, our work falls into the second category. However, many work have been focused on automatically generating test cases, but few of them concentrated on test oracles. The major merit of our approach is that generated test cases contain test oracles.

Our research has some advantages over existing works in the literature. First, test oracles will help testers to be more accurate and find more bugs. When a test case has oracles such as expected change on the database, the tester will find more bugs per test cases, and the entire test will be more reliable. Second, as the generated test cases using this approach contain a natural language description, it can be a valuable document for testers and developers to understand the system faster and easier. Next, generated test cases are based on real user actions; therefore all the test cases point to a real user usage of the system. Finally, collecting user actions helps us understand how often users are employ different parts of a system to determine the priority of test cases.

4.6 Summary

Many applications are already in use, but there are insufficient test cases for them. Besides, developing test cases with oracles for these applications is costly. This chapter proposed a new approach to generate test cases with oracles containing a human-readable description, collecting user, application, database, and file system data. Data analysis shows many potentials in this kind of data, and high-quality test cases can be generated using such data. We applied n-gram on the data and found valid test cases with oracles. Moreover, time analysis of the data shows that time is an indispensable feature of the data, and test cases can be identified using this feature. We performed an empirical study on four applications under test and were able to find test cases where none of the state-of-the-art tools can find them. In the future, we intend to do more analysis on this data and apply other algorithms to find GUI test cases. Another potential project could be to label valid test cases in the dataset manually and to use machine learning approaches to predict new test cases in an application.

5 ComVis: A Comparative Visualization Approach to Test Data-Driven Applications

Much software, especially scientific software, works with a massive amount of data. As we mentioned in the limitations section of Chapter 3, usually, these systems take a large amount of data as input, apply scientific models, and return an extensive amount of data as output, which is difficult for automated tests to detect. During software production or re-engineering, each release of new versions can cause several data issues. Detecting the bugs that cause these data defects is challenging, and the quality assurance process may miss many of the issues. This chapter introduces a comparative data visualization approach that can help software testers and developers identify these data defects easier. Our user study confirms that the new approach finds more data issues when compared to an existing approach. Also, our quantitative participants' feedback shows that our approach is more user friendly and offers more dependable visualization to find more defects.

5.1 Introduction

A tremendous volume of data is being produced every day, and more software is needed to analyze and model these data. These data are used in various fields such as society administration, business, scientific research, and more [15]. There are many challenges regarding using this increasing amount of data, such as data collection, data storage, sharing it with other collaborators, analyzing it, and visualization [15]. When it comes to developing software that analyzes and visualizes the data, it is vital to ensure that the software does not change the data in an undesired way after each release. For this purpose, a robust testing process that can identify bugs related to data is required.

Many studies have focused on generating test data as input of the functions and methods using different approaches such as symbolic execution [64, 102], search-based software test data generation [60, 34, 61] and genetic algorithms [74]. These studies focus on generating input for a module and checking if the output is as expected. When the software's input and output are a massive, these techniques are usually not relevant. Moreover, most of the research has worked on function-level test data generation, which is not appropriate for system-level tests. This work investigates an approach that employs data visualization to help software testers and developers find data discrepancies more accurately and efficiently.

Data visualization of software data has been used in different software engineering areas such as development, maintenance, evolution, re-engineering, and reverse engineering [46]. There have been some uses of visualization in software testing as well [52], but most of them are focused on test planning and test case generation and leave the task of bug detection to testers.

Besides, in some cases, even when we have unit tests for all the functions, existing unit tests cannot identify

some high-level bugs. One of the main concerns of software teams is to ensure that old functionalities are working correctly after each release. In the case of modernizing a legacy system, the entire system's functionality must remain the same as in the original. The Cold Regions Hydrological Model (CRHM) application (We refer to this app as CRHM in this paper) is an example of a legacy system in the modernization process. One of the main features of CRHM is visualizing the output via line charts. Users also have an option to add two existing output files to the visualization. Therefore, they can compare outputs on a line chart and see the differences. However, these charts are not designed to examine outputs and find data issues. For better visualization purposes, we have developed a web-based Comparative Visualization Tool (ComVis) to visualize the CRHM application's output after each release so that testers and developers can find data defects more efficiently. This tool can be used for any pairs of flat data files, such as CSV files.
We designed a user study to evaluate the effectiveness of ComVis. As this user study has been done during the COVID-19 pandemic lockouts, we developed a tailored online survey to make sure we could capture all the data that we require. Schema of this survey is presented in Appendix A. Our study focuses on answering the following research questions.
RQ1: Is ComVis able to find more data issues than embedded CRHM visualization?
RQ2: Is ComVis easier to use than CRHM?
RQ3: What are the advantages and disadvantages of ComVis compared to CRHM?

This study's contributions are the following:

1. A comparative visualization method and its implementation to compare two flat data files.

2. A tailored user study to compare the effectiveness of our proposed approach with existing approaches.

The rest of this chapter is organized as follows. Section 2 describes our proposed approach. Next, Section 3 explains the characteristics of our experimental setup. Section 4 discusses the results of the study and provides analysis. A discussion of the results is presented in Section 5. Section 6 elaborates related work. Section 7 discusses the threats to the study's validity, and finally, the conclusion is provided in Section 8.

5.2 Proposed Approach

This section describes our proposed approach (ComVis) for finding data issues in data files. ComVis's main purpose is to help software testers and developers find data issues generated by software defects after each release using comparative visualizations. Figure 5.1 shows the architecture of the ComVis, which contains two layers:

- **Data layer(ETL pipeline)** In this layer, first, data from different file sources such as CRHM observation(OBS) files are extracted. Second, extracted data is validated to make sure data format and structure and columns are valid. Third, data types of columns are converted to the same data type

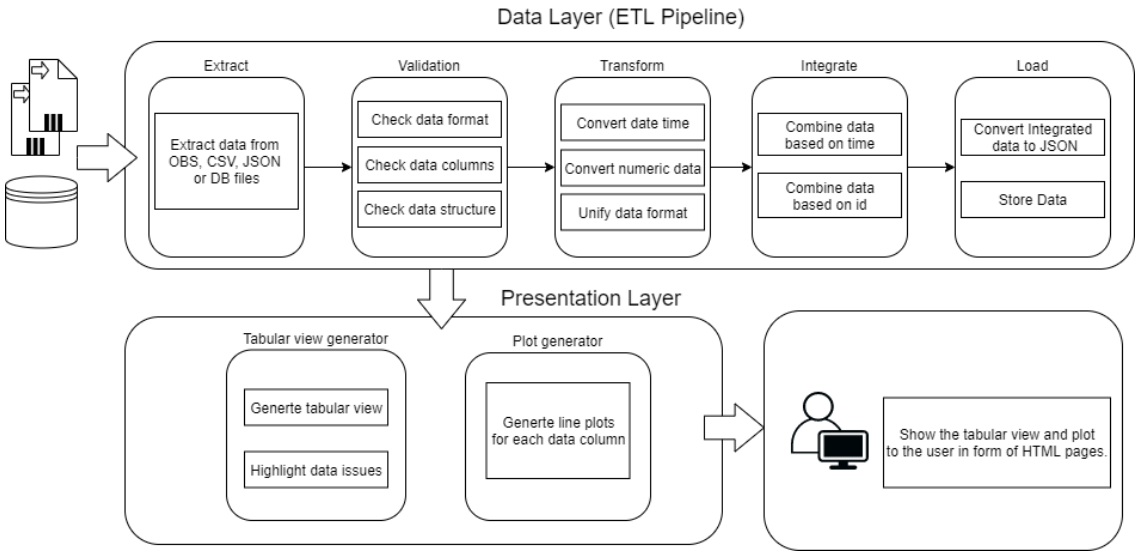


Figure 5.1: Architecture of the approach.

and format. Next, data are integrated into the same data structure, and finally, the integrated data are loaded to a JSON file, which is used as input of the presentation layer.

- **Presentation layer** This layer takes the JSON file from the data layer as input and generates two data visualizations. First, a tabular view of the data highlights the data issues, and second an interactive line plot that users can explore using features such as zoom-in and zoom out. A sample of both visualizations is shown in Figure 5.2 and 5.3.

	time	file1Col 1	file1Col 2	file1Col 3	file2Col 1	file2Col 2	file2Col 3
1366	1974-11-26 23:00	2.018630	2.018630	2.018630	2.018630	2.018630	2.018630
1367	1974-11-27 00:00	0.766445	0.766445	0.766445	0.766445	0.766445	2.018630
1368	1974-11-27 01:00	0.766445	0.766445	0.766445	0.766445	0.766445	2.018630
1369	1974-11-27 02:00	0.766445	0.766445	0.766445	0.766445	0.766445	2.018630
1370	1974-11-27 03:00	0.766445	0.766445	0.766445	0.766445	0.766445	2.018630

Figure 5.2: Tabular view

5.2.1 Data layer (ETL pipeline)

The primary purpose of this layer is to provide clean and valid data to the presentation layer. For this purpose, we have the following steps:

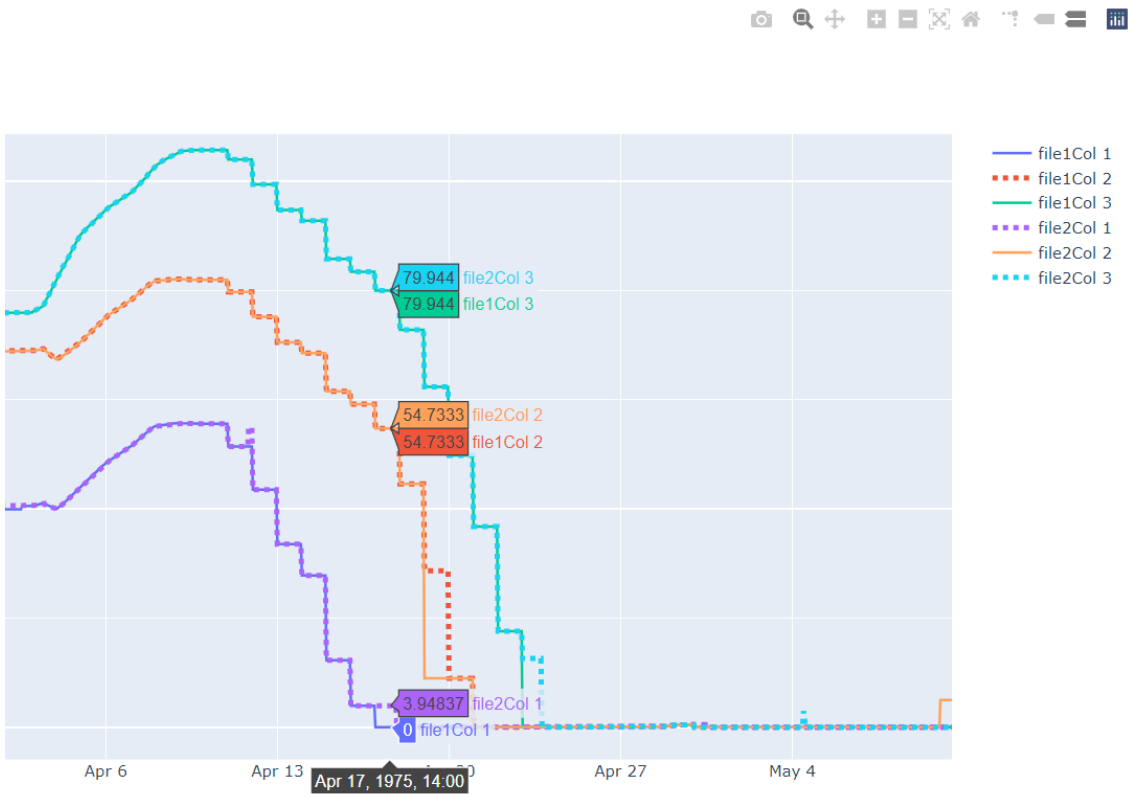


Figure 5.3: Interactive line chart

Extract

In this part, we extract data from different sources such as databases, CSV, JSON, and CRHM observation files(OBS). Observation files are the CRHM application's output. Each data entry of an OBS file has a key-value structure where value can be an array of values, and the key is usually a date and time value. We are interested in testing OBS files to ensure the modernizing process does not induce any data issues. For this purpose, we need to compare data from both before and after change data files. Next, we need to ensure the validity of the data.

Validation

Observation files have a specific format. These files have two major parts, a header, including column names and measurement units, and a data section. It is essential to check if the observation data files are valid. In this part, we check the format as well as columns and whether the data is structured correctly.

Transform

As we have two data files to compare, we need to unify the data columns. For example, data time format can be in the form of *YYYY-MM-DD HH:MM*in one file and *MM/DD/YYYY HH:MM* in another file. These

sorts of issues are handled in this part. Moreover, when the data is imported, all numeric data is stored as a string. All of the numbers are converted to numeric data and we made sure all data column pairs have the same data type.

Integrate

To find data discrepancies, we need to know based on which column we will perform the comparison. There are two ways to compare data, first based on the key (usually date and time) and second based on Id. When we compare two data files, we expect that data for a specific key should be the same. On the other hand, when we compare them based on Id, we expect that the entire data files will be the same. The second way takes keys into consideration in the process of comparison.

Load

Finally, the clean, validated, and integrated data is exported to a JSON file so the presentation layer can read and convert it to interactive data visualization.

5.2.2 Presentation layer

In this layer, the data from the data layer is read, and two kinds of visualizations are generated.

Tabular view

This view shows raw data next together with an emphasis on data issues. The main goal of this visualization is to show all the data issues accurately. There are two kinds of tabular view available to testers. First, the view in which data is joined based on the key. This view assumes that data with the same key should be the same. If there is an issue, the row will be highlighted. Second, the view of the data joined based on Id. This view takes the order of the data as an important factor in comparing the data. A sample of the tabular view is shown in Figure 5.2 In this example, there are data issues in row 1367 to 1370. The value of *file1Col3* and *file2Col3* for these rows should be the same but are different. Therefore, these rows are highlighted. On the other hand, all corresponding values for row number 1366 are the same and not highlighted.

Interactive line chart

Another visualization that we have used to identify data issues is an interactive line chart. Each line represents a column's values, and in cases of any difference in the data, it is identifiable on the plot by testers. A sample of the line chart is shown in Figure 5.3. In this sample, several data issues can be observed. For instance, on April 17th, 1975, 14:00 *file1Col1* and *file2Col1* are equal, and this can be recognized from both the line chart and values shown on mouse hover for each line. Users can perform the following actions to find data issues better:

Compare data on hover: By hovering the mouse on any point on the chart, users can see and compare values for all columns.

Toggle spike lines: The user is able to see lines on the graph indicating the exact x-axis and y-axis value for a specific data point.

Zoom-in and zoom-out: Users can zoom-in as much as they want and compare data on a smaller scale. They also can zoom-out to see a larger view of the plot.

Reset axes: In case users perform many zoom-ins and zoom-outs, they can reset axes to see the original view.

Download plot as png: Users can take a snapshot of the plot if they need to attach it to their bug report.

Disabling and enabling lines: To have a more concise line chart, users can enable and disable lines that they require by clicking on the respective legend.

5.3 Experimental setup

This section elaborates on our experimental setup. First, we describe the subject system that we have used to perform our experiment. Second, we provide the participants' demographics. Finally, the experiment process has been described.

5.3.1 Subject system

We selected the CRHM application as our subject system. This Windows-based application is used with hydrology experts to model and analyze water behavior in cold regions. For instance, this application is employed to predict water balance for some research basins in the prairies area[25]. Researchers collect various data such as snow depth and density, precipitation, and air temperature and estimate a basin's streamflow using CRHM application . One of the principal parts of this application is the visualization part, which provides line charts based on date and time. It is very critical to have accurate charts to be able to comprehend the data and predictions.

5.3.2 Participants' demographics

We asked 25 participants aged between 20 to 40 to take part in our experiment. Twenty of them finished the experiment (7 females and 13 males) in a one-week time limit. None of the 20 participants had prior knowledge about CRHM, nor were they exposed to ComVis. We asked them about their level of education, current role, and their testing and development experience. Figure 5.4 shows more details about participants. We also asked two experienced CRHM users to participate in our experiment to determine how ComVis performs compared to CRHM for users who have prior knowledge of CRHM and have not seen ComVis before.

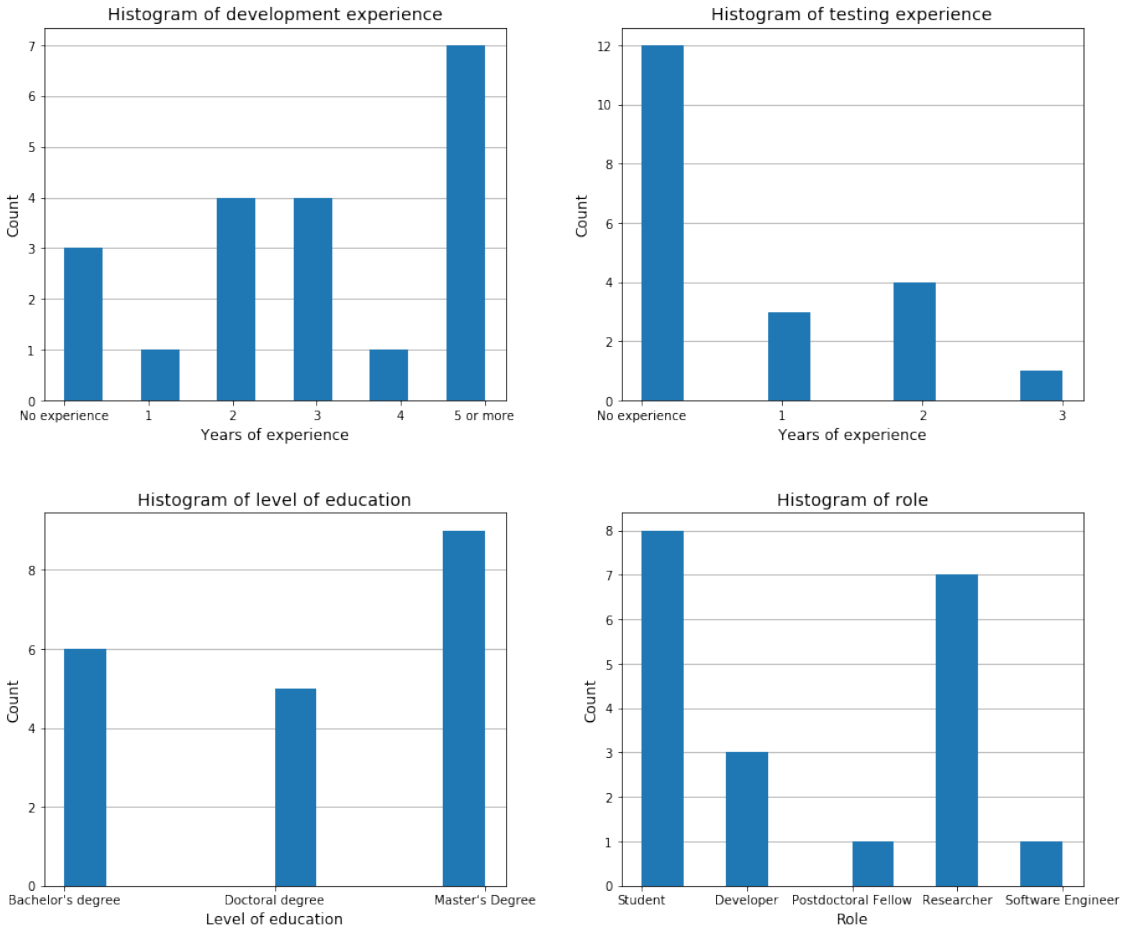


Figure 5.4: Participants' demographics

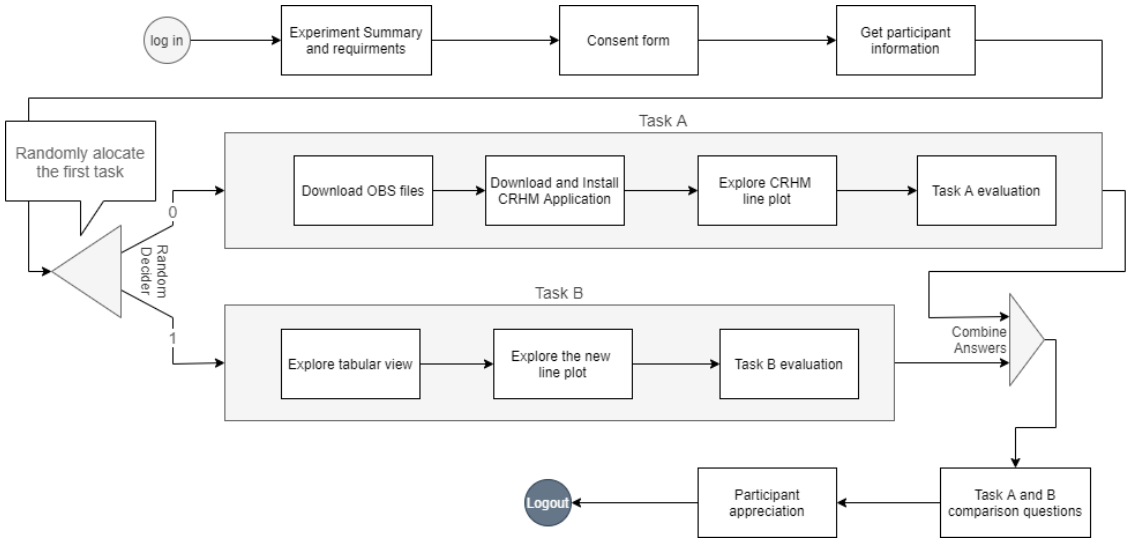


Figure 5.5: Experiment process

5.3.3 Experiment process

In this section, we describe the process that participants followed to complete the experiment. This experiment is a within-subjects design experiment, which means a user completes all the experimental conditions[31]; in this case, both Task A and Task B. Our main goal is to design an experiment where users can perform it on their own with minimum assistance. Moreover, it is vital to design an experiment that is not biased. For this purpose, we developed a web-based experiment application using the Flask framework that satisfies all our requirements. Figure 5.5 shows the experiment process.

All the steps shown in Figure 5.5 are self-exploratory. All the information that a user needed is mentioned on each level. To ensure that all explanations and instructions were clear, we asked three users to review all the content, and their feedback was used to improve the content.

5.4 Evaluation Results and Analysis

This section contains the analysis and results of the evaluation of each research question.

5.4.1 RQ1: Is ComVis able to find more data issues than embedded CRHM visualization?

To answer this question, we provided two data files to the participants with five columns and approximately 5200 rows. One of the files did not contain any issues. We injected 30 different data issues into the other file expecting users to find them using CRHM application and ComVis. All the 30 data issues were scattered at random throughout the second data file. We stored the start and the finish time of each task to precisely calculate the time spent on each task. Results show that users were able to find notably more data issues

using the ComVis. Figure 5.6 shows the results. As our data is dependant, not normally distributed, and the sample size is not large, we selected the Wilcoxon Signed Rank test to make sure that the difference is statistically significant. Equation (1) shows how this test calculates test statistics. Test statistic calculated from this equation for time and count of mismatches is 41 and 8, respectively, which is less than the critical value mentioned in the Wilcoxon's critical value table [97] for a sample size of 20 with a confidence interval of 95%. This shows that the differences between both time and number of detected issues are statistically significant.

$$W = \sum_{i=1}^{N_r} [sgn(x_{2,i} - x_{1,i}).R_i] \tag{5.1}$$

From the results, we can conclude that ComVis is capable of finding more issues, but it takes more time. The mean number of mismatches for all users is 23.8 for ComVis, and 15.8 for CRHM. The mean of time consumed on ComVis and CRHM is 11 and 8.8 minutes respectively. It means, on average, users were able to find eight more issues at the expense of spending 2.2 more minutes on ComVis.

Results for experienced users are different. They find 24 issues using CRHM and 25 issues applying ComVis, on average. Although, the difference is not notable, the fact that, ComVis has performed almost the same as CRHM in finding the number of issues is promising. Time consumed by experienced users on CRHM is 3 minutes compared to 6 minutes for ComVis. Like novice users, they takes more time to finish when using ComVis.

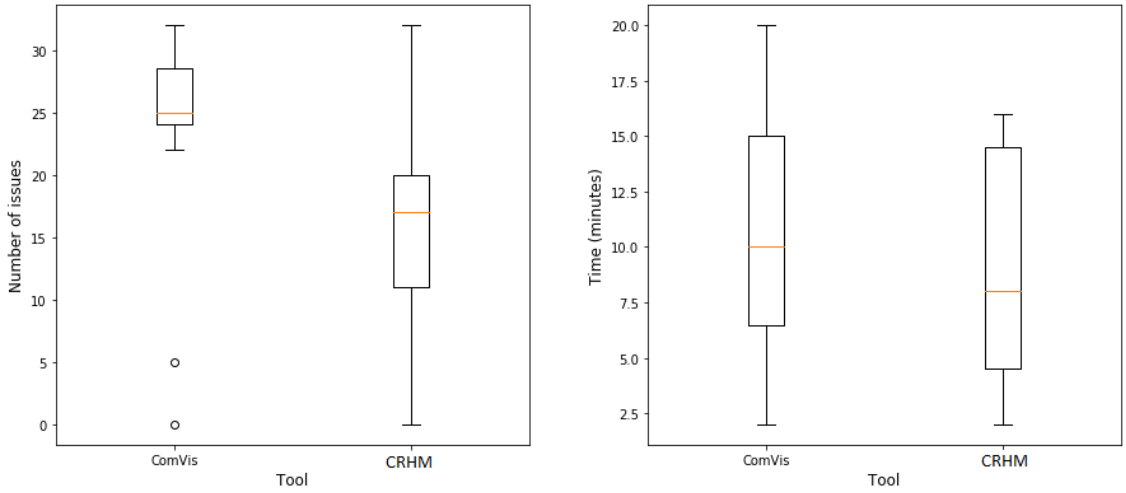


Figure 5.6: Box plot for number of data issues found and time spent on each task

5.4.2 RQ2: Is ComVis easier to use than CRHM?

We used the Nasa Task Load Index(TLX) tool to measure both tasks' workload to answer this question. We asked participants the following six questions after each task and analyzed the answers.

Q1: How mentally demanding was the task?

Table 5.1: NASA TLX Questions result statistics

Question	CRHM mean	ComVis mean	W	Statistically significant
Q1	3.58 — 2.5	4.47 — 4.5	43	Yes
Q2	2.79 — 2	2.79 — 2	70	No
Q3	3.05 —2	4.05 — 3	18	Yes
Q4	8.16 — 7.5	8.37 — 9	50	No
Q5	4.31 — 1.5	4.79 — 2	53	No
Q6	3.95 — 1.5	3.84 — 3.5	76	No

Q2: How physically demanding was the task?

Q3: How hurried or rushed was the pace of the task?

Q4: How successful were you in accomplishing what you were asked to do?

Q5: How hard did you have to work to accomplish your level of performance?

Q6: How insecure, discouraged, irritated, stressed, and annoyed, were you?

Participants provided a score between one (very low) to ten (very high) to answer these questions. The results of these questions for novice users are presented in Figure 5.7. Similar to RQ1, we used the Wilcoxon Signed Rank test to find out if the results are statistically significant. Results from equation (1) for all the question is shown in Table 5.1. As shown in the last column, only the difference between Q1 and Q3 results is statistically significant. This table also includes the mean value of each tool's responses for both novice and experienced participants. The second number on the mean columns is related to experienced users.

The results of the first question reveal that ComVis is more mentally challenging than CRHM for both novice and experienced users. The statistical test result confirms that this difference is statistically significant. Both tasks were almost the same with regards to the exertion of physical effort. The difference is not meaningful, with both tasks showing a low level of physical activity. Novice participants reported that they felt more rush while performing the task on CRHM. As in question 1, this difference is also statistically significant. However, experienced users felt more rushed using ComVis. Results indicate that participants performed slightly better to complete the CRHM task successfully, but this difference is negligible. Finally, in response to question 6, novice users reported less stress, irritation, discouragement, and insecurity while using ComVis. Nevertheless, this difference is not significant, either. On the other hand, experienced users reported greater levels of less stress, irritation, discouragement, and insecurity while using ComVis.

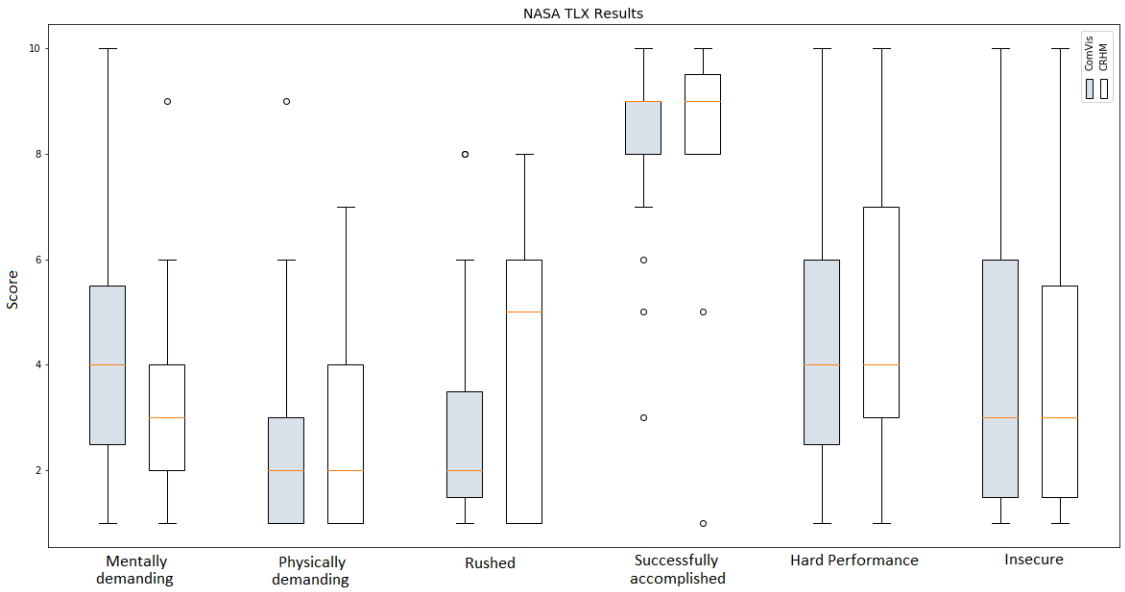


Figure 5.7: Box plot NASA TLX results

5.4.3 RQ3: What are the advantages and disadvantages of ComVis compared to CRHM?

To answer this question, we asked participants what they liked and disliked about each task. They described their ideas about each task in several sentences. This data was reviewed and organized using Thematic Analysis(TA) to discern themes related to each task's advantages and disadvantages. TA is an appropriate approach to research when we attempt to explore and understand people's perspectives, knowledge, experiences, or values for qualitative data [18]. There are two approaches in thematic analysis, deductive and inductive. The researcher has some hypotheses about the themes based on the literature and tries to find them in the deductive approach's qualitative data. On the other hand, in the inductive approach, the researcher does not have any predefined assumptions about the data [18]. We took the inductive approach of thematic analysis, which allows data to identify the themes itself. This approach helped us to understand participants' concerns. After performing inductive TA, we can understand what aspects of the study participants are communicating. Table 5.2 shows the themes extracted from the data, including codes and some quotes from the participants. The codes mentioned in the table assisted us in identifying the themes.

Inductive TA result analysis

Inductive TA analysis was conducted on user responses and five themes obtained (Visualization, Ease of use, Performance, Accuracy and Platform). In this subsection, we intend to elaborate on each theme and discuss how each task was performed in each theme.

Visualization: The way that each task has visualized the data is the primary concern of participants. Almost all the participants have mentioned this subject in their description. The vast majority of the participants prefer data visualization in ComVis. They have mainly specified options in the control panel of ComVis,

Table 5.2: Themes of qualitative data

Theme title	Codes	Quotes from participants
Visualization	Zoom Scroll Color Tabular Legend Panning Column Plot	Didn't like the **scrolling** and counting part at all. **Zooming** and **panning** on the **chart** was very primitive it leads to loss of context and frustration. I liked how easy it was to **zoom** in and out and **toggle** features. I liked the idea of showing different models in different **legend** and also seeing values by **zoom in** and **zoom out** facility.
Ease of use	Demanding Convenience	I didn't like the interface (not intuitive and **hard** to perform simple actions) Very **difficult** to scroll the data. Overview of information is **helpful** to identify close differences. Much better than the window application especially the numbers that appear when you hover on the graph made task much **easier**.
Performance	Fast Time consuming	I was quick but maybe I found very few issues. I liked it very much. This is super fast. Plot was good but the tabular took a lot of time since the data is huge. I didn't like the amount of scrolling to go through the tabular data.
Accuracy	Accuracy in finding more issues	It was easier to check stuff but I may have **miss** some data The tabular view was very **accurate and** **provide more confidence**
Platform	OS independence Integration	I like the idea of having web-based **platform-agnostic** applications that can be run on both my **Windows** and **Mac** machines. I liked the task because it is **integrated** in CRHM software.

such as zooming, panning, toggle, and hovering as benefits of visualization in this tool. On the other hand, the interface of CRHM was not popular among the participants. They suggested that it is counter-intuitive and harder to work with comparing to ComVis. The majority of the participants did not like the zooming option in CRHM and considered it hard to handle. One of the participants stated,"Zooming feature is very difficult to use for three reasons - a) rectangle box to select region is not colored, b) no proper zooming technique, c) zoom levels are absent." Only one user note "visualization is excellent" about CRHM without further explanations.

Ease of use: After visualization abilities, participants addressed how easy each tool is to use. Generally, other than zooming features that were hard for users to use and coloring, which made reading the graphs difficult, users considered CRHM an easy tool to use. Participants discussed the first part of the ComVis (finding data issues on the tabular view), a laborious and tiresome task to perform. Most of the participants complained about the amount of scrolling to find data issues on this task. Participants pointed out that features such as comparing values on hover made the second part of this task (finding data issues using a plot) easier.

Performance: The amount of time taken to perform each task was another issue that participants reported. Most of the users noted that ComVis is more time-consuming. It takes more time to load the visualizations comparing to CRHM. This is valid as the CRHM application is a native windows application, and it is faster than a web-based application with limited resources. One participant responded about CRHM: "I liked it very much. This is super fast."

Accuracy: Participants also highlighted the accuracy of each task. Participants consider ComVis more reliable regarding finding data issues. "It is difficult to see the difference in application.", one of the participants mentioned about CRHM. Results from the RQ1 support this finding as users could find more data issues in ComVis.

Platform: Some participants noted that the ability to perform ComVis on different platforms is an advantage of this tool. On the other hand, one participant liked the idea of the integration of the CRHM application and the required task.

To sum up, based on user reports, the ComVis is more user-friendly, straightforward, accurate, and platform-agnostic. CRHM is faster both in performing user commands and also quicker in finding issues.

5.5 Discussion

Finding data issues on data files caused by newly introduced bugs after each release can be a major concern of data-driven applications such as CRHM that use massive amounts of data every day. With introducing ComVis, we have demonstrated an approach to make the task of finding data issues more straightforward for testers and developers. We have added two types of visualizations to this tool to make finding issues both accurate and convenient. The study results showed four main findings:

- There is a meaningful difference between the number of data issues found using ComVis.
- Participants found ComVis more time consuming and mentally demanding.
- Using ComVis makes participants less rushed.
- Participants were more confident with the results of ComVis.

This section discusses these results by explaining possible reasons for them and concerns that these results raise.

Why can ComVis find more data issues? There are several possible reasons why ComVis performs better in finding data issues. First, ComVis uses an accurate tabular view that shows all the differences with details, making finding issues more feasible. However, users reported that working with the tabular view is time-consuming, but it results in discovering more data issues. Second, interactive visualization usage with the ability to zoom into data and comparing on hover could be another possible reason for finding better results. Several papers have suggested that interactive data visualization can result in a more detailed understanding of the data. Participants also reported that interactive data visualization had made their task more manageable.

Why is ComVis more time consuming and mentally challenging? We assume, however, the tabular view helps participants be more accurate in finding data issues, causing them to spend more time and energy on ComVis. Scrolling over the data to find the data issues takes a considerable amount of time and efficiency. Many participants had mentioned that they do not like the amount of scrolling to find data defects. This problem would be resolved by a new design that can indicate the data issues' exact location. We are planning to solve this problem in future studies. Moreover, ComVis is a web-based application and, compared to a native application in response to large amounts of data, is slower.

Why did Comvis make participants feel less rushed? We believe that because ComVis has a more familiar web-based and user-friendly interface, participants can relate to it and feel that they can finish the asked tasks in a reasonable time. Results from the qualitative analysis show that users find ComVis intuitive and easier to use, which can help them to work with it with peace of mind.

Why were participants more confident with the result of ComVis? The reason for more confidence in the outcomes of ComVis could be that users can see the data issues visually better in both the tabular and line chart view. Participants are able to see the differences with bright colors in the tabular view. In the line chart, the colors and shape of the lines make noticing the differences easier. As a result, users can be confident that they are not missing many data defects. Better zooming options mentioned by many participants can be another reason for users to be sure that they perform better with drilling down into the data.

Why are results for novice and experienced users not the same?

Table 5.3 shows the difference between novice and experienced participants in answering the research questions. However, ComVis took more time from all participants; experienced users find almost the same number of data issues using both ComVis and CRHM. This result is promising as it is their first time using

Table 5.3: Summary of results based on type of participants (Novice and experienced)

RQ	Novice	Experienced
RQ1: Number of detected issues	Finding considerably more issues using ComVis in longer time	Finding the same number of issues using ComVis in longer time
RQ2: Ease of use	ComVis is more mentally challenging but less rushed	ComVis is more mentally challenging, rushed and insecure
RQ3: Advantages and Disadvantages of ComVis	Better zooming options More accurate Responsive visualization platform-agnostic Slower	No installation needed Responsive visualization

ComVis. Since experienced users have used CRHM for a long time, we assume they are familiar with all the application options, resulting in finding more defects. For example, one of the features that novice users have complained about is the zooming features, experienced users have figure out how to use this feature before, and now they are comfortable with it.

As we discussed, ComVis makes Novice users less rushed, but it has the opposite effect on experienced users, and they also felt more insecure. One probable reason for this variation is that as experienced users are fully aware of processes in CRHM, they know exactly what they should expect from the system. On the other hand, it was their first time working with ComVis, which may have made them rushed and insecure.

Novice users have mentioned many advantages and disadvantages of ComVis and CRHM, but experienced users have only mentioned a few advantages of ComVis. One of the experienced users mentioned, "I am very familiar with CRHM application, so I do not find anything to be liked or disliked, as I have been seeing it for years."

Considering the advantages and disadvantages of ComVis and CRHM, we concluded that several changes are necessary to make this tool better. First, the tabular view should be changed so that users can find data defects faster and effortlessly. One possible solution for this could be additional visual cues showing the location of data issues. These kinds of changes can reduce the mental demand of ComVis. Second, we should add more types of visualizations to help testers and developers see only the data defects. This will help them to concentrate only on the data defects. Finally, we need to improve the tool's performance in a way that users do not see a considerable amount of time difference between ComVis and CRHM.

5.6 Related Work

Our work introduces a new comparative visualization approach to find data issues on data-driven applications that are not discovered by conventional software testing. This section, therefore, reviews the usage of visualization in software testing, testing data-driven applications, and the role of visual difference analysis in the literature.

5.6.1 Visualization in Software Testing

Visualization have been used largely in software engineering for various purposes such as software comprehension [83, 63], performance optimization [99, 39], change history analysis [101], bug localization [40] and investigating code quality [68]. Moreover, several pieces of research have been conducted to utilize the benefits of visualization in different areas of software testing such as test coverage, test results, test smell and life cycle of bugs.

Urata et al. [92] proposed an approach that can help identify parts of software that has been tested using Unified Model Language(UML) and generating a testing diagram. Results from these visualizations can be used to improve test coverage. Nakagawa et al. [70] also have suggested a procedure that finds links between test cases and specification documents. Using the similarity links, they provide a graph-based visualization to identify specification coverage of the tests. Both of these works have employed visualization to help testers to understand the coverage of their tests better.

In a recent study, Hammad et al. [33] have focused on visualizing test results to help software teams to understand and trace test cases and their results using different views. This work's primary goal is to keep software testers informed about test results for software with a substantial number of test cases. Likewise, an interactive visualization technique has been introduced by Opmanis et al. [72] that tries to find the root causes of software defects by visualizing the regression test results in a large-scale software application.

Several studies were also found that try to provide better information about test smell. TestQ [9] is a tool that provides visualizations and insights that help developers investigate test smells hot spots. This tool can also generate views that help testers and developers explore the test suite's design. In another study [69], researchers developed a code smell detection tool that helps developers find test smells in the development time and inform them using interactive visualizations.

D'Ambros et al. [20] developed a tool to visualize bugs history, as evolving entities which change over time. They introduced bug watch view that gives details about bug the history in a pie chart like visualization. They also introduced *System Radiography* view to study how the open bugs are distributed in the system over time. Similarly, Yeasmin et al.[100] introduce a visualization approach that helps new developers understand history of bugs of a software system.

As mentioned above, there have been many studies and tools that utilize visualization to alleviate software testing struggles. However, we did not find any research focusing visualization of the difference of output of

a method, object or subsystem before and after a change to find bugs introduced by the change. This type of visualization is beneficial to detect defects in applications that use a large amount of data. In the next subsection, we review testing approaches that have been done on data-driven application.

5.6.2 Testing data-driven applications

Testing data-driven applications is challenging as large amounts of data are incorporated into complex algorithms and models. Many studies have been conducted to address issues in this area. Morán et al. [66] introduced a testing technique called MRFlow (MapReduce data Flow) to detect data defects over MapReduce programs which is frequently used in Big Data applications. This approach investigates changes between the input and output to find bugs.

In another study, Li et al. [48] introduced BIT-TAG, a tool that generates test data for ETL applications in order to reduce the time of testing. They first create Input Domain Models (IDMs) from a sample of real data. They also generate some constraints manually from the application requirements. Having the data and constraints, they try to minimize the amount of data, considering retaining the data's effectiveness to reduce testing time. They show that they can detect the same number of defects with a small sample from the original data. Similar to BIT-TAG, Domino [2] is developed to generate test data to detect bugs in relational databases using domain knowledge. This tool's primary duty is to find flaws related to database design that causes software defects. Several other studies focus on testing data-driven applications, but none of them use visualization to detect bugs. This study introduces an approach that helps testers and developers explore the output data after a release and find software bugs using data defects.

5.6.3 Visual difference analysis

Features in visualization that help users to identify differences are beneficial in context comprehension and anomaly detection. There are several studies on different contexts that utilize visualization to identify differences. Williams et al. [78] designed a user study to assess the degree of identifying qualitative differences in three-dimensional models of molecular models using Virtual Reality(VR) devices. The result of this study proves that the capacity of participants improves when they can interact with and manipulate the visual simulation of the molecule. In another study [104], a tool has been introduced that visualizes differences between codes developed in trigger-action programming (TAP) languages, which usually is used to program the Internet of Things (IoT) devices such as smart home appliances. The differences visualized by this tool in TAP programs help users understand modifications and decide whether they require them or not.

Girschick et al. [29] also designed a change detection algorithm that automatically distinguishes differences between ULM diagrams, especially class diagrams, and visualizes them by animation and colors.

Many research projects try to practice visualization in software testing. Also, several of them are focusing on testing data-driven applications. Similarly, some studies investigate the visualization of differences, but little or no work exists in the intersection of these three fields to the best of our knowledge.

5.7 Threats to Validity

Several threats exist to the validity of this research. The first external threat to this research, similar to many other software engineering studies, is that the subject system may not be a suitable representative. We moderated this threat by selecting a real-world subject system that has been used by many users. Another threat to the internal and external validity of this research was being biased. To avoid systematic effects that interfere with the interpretation of experimental results, we took the following actions to balance the experiment to minimize bias effects. These actions improve the internal and external validity of the study.

5.7.1 Minimizing Novelty Bias

Usually, our preferences favor ideas that we have not experienced. On the contrary, we may prefer approaches because we have a long time familiarity with them[49]. As we do not have any control over this factor, we tried to select participants who are not familiar with the CRHM application. Moreover, because our approach is new, none of the participants have used it before. Out of 20 participants, only one has claimed that he/she has used the CRHM application before. Having 95 percent of the participant new to both parts of the experiment, we can confirm that we have eliminated novelty bias.

5.7.2 Eliminating Order Biases

In an experiment where we have several cases to evaluate, the order in which the evaluations are performed is essential. Several grounds can explain why ordering may cause biases. The first is that participants may not remember equally well the various evaluations in the order [73]. For instance, when a user performs task A first and then task B, by the time he/she evaluates how mentally demanding task B was compared to task A, he/she may not remember precisely how mentally demanding task A was. Second, participants may get tired from performing the first task and not be fresh, eager, or motivated enough to complete the second task. To alleviate the pain of order biases, we allocate the tasks to users randomly. Half of the users start with task A and finish by doing task B, and the other half start with task B and finish with task A. This approach helps us to minimize the effect of order biases.

5.7.3 Minimizing Experimenter Bias

Experimenters of a study can influence participants subconsciously in favor of their hypothesis [84]. There are several ways to reduce the experimenter effect, which is used by software engineering experiments in the literature [86, 27]. Blinding techniques such as single-blinding and double-blinding have been used widely in other disciplines such as medicine and psychology, and have been employed in software engineering research. In this approach, participants' allocation to tasks is hidden from participants (single-blind) or both participants and experimenters (double-blind). As we have used a within-subject experiment, participants

were asked to perform both tasks, but the experimenter is not aware of the order of participants' tasks. Rosenthal et al. [80] show that the experimenter's elimination of visual cues can reduce experimenter bias. Adair et al. [1] also show that verbal cues can cause experimenter bias as well. As our approach is entirely web-based, both experimenter visual cues and verbal cues are eliminated, minimizing experimenter biases.

5.7.4 Handling Sampling Bias

We have asked 25 persons who have affiliations with the University of Saskatchewan to participate in our experiment. One of the issues that can be a threat to such an experiment's external validity is sampling bias [59]. We selected our participants from different ages, genders, nationalities, and education levels to mitigate sampling bias. Our study participants are between 20 and 40 years old, a good sample representing the software industry population where 97.5 percent of them are under 43, with a mean age of 29 [67]. We also limited the number of participants from a single nationality to a maximum of 40 percent from 6 different nationalities. The level of education has been taken into consideration as well. Forty-five percent of participants have a master's degree, 30 percent have a bachelor's degree, and 25 percent have a Ph.D.

5.8 Summary

To test a data-driven software product more accurately, we introduced ComVis, a Comparative Visualization tool, to compare software output data after each release to discover data issues introduced by undetected bugs. This tool provides a tabular and interactive line chart visualization to users, helping them find data issues. Our evaluation shows that this approach is beneficial in finding data issues. Moreover, our quantitative analysis of participants' feedback shows that the proposed tool is easy to understand and use. We realized that the line chart was useful from users' feedback, but the tabular view could be simpler to use. For future research projects, we intend to focus on simplifying the tabular view in order to make the task more convenient for the users. Moreover, we intend to design more visualizations to cover various types of data. We believe our strategy is scalable and can be used for more complex data. Moreover, adding more types of visualizations based on the data can help testers and developers be more accurate in detecting new bugs.

6 Conclusion

6.1 Concluding Remarks

Re-engineering is a wise choice over redevelopment for a legacy system as it is more reliable and affordable; however, it comes with its risks. Software testing can significantly reduce risks associated with re-engineering a legacy system. In this thesis, we have focused on three studies to improve test automation in the process of re-engineering legacy systems, especially when the documents are absent. First, we introduced a testing model and an automation platform to detect functional and performance issues in the early re-engineering stages. This method performed well on a legacy system and showed that the system could find more bugs compared to manual testing. Second, to tackle the lack of test case generation problem for legacy systems, we proposed an approach that generates test cases by recording the user actions and the application under test reactions such as database and file system changes. We found the order of user actions useful in finding test cases with a large number of test cases and the time difference between actions for smaller test cases. The study results show that this approach can generate more accurate test cases than the state of the art approaches. Finally, in the third study, we introduced a comparative visualization tool that helps testers and developers find data issues after each release of the renovated system. This tool provides a tabular and a line chart visualization to assist in detecting data issues easier. The user study on 20 users shows that users can find 51% more issues than the embedded visualization tool in the legacy system; however, it takes 25% longer to finish the task. NASA-TLX assessment and quantitative analysis of the open-ended questions also prove that ComVis is easy to use and understand.

6.2 Future Directions

This thesis's primary purpose is to assist testers and developers in testing software products that are re-engineered. The following further research could engage in continuing this goal.

Automatic test execution of test cases

The approach described in Chapter 4 is implemented only for windows applications. Although the same process will work for Web and Mobile applications, there is a need to develop a data collection layer for each platform. Each of the platforms has its challenges. For example, in web-based applications, recording database changes is tricky as many users may use the system simultaneously. We intend to develop a data layer for mobile and web-based applications to use the procedure for test case generation for applications other than windows applications. To do so, we need to work with Selenium IDE (a browser extension that provides features to record user actions), which will help us record user actions.

Test case generation for Web and Mobile application

The approach described in Chapter 4 is implemented only for windows applications. Although the same process will work for Web and Mobile applications, there is a need to develop a data collection layer for each platform. Each of the platforms has its challenges. For example, in web-based applications, recording database changes is tricky as many users may use the system simultaneously. We intend to develop a data layer for mobile and web-based applications to use the procedure for test case generation for applications other than windows applications. To do so, we need to work with Selenium IDE, which will help us record user actions.

Investigating more algorithms for test case generation

The data collected for test case generation contains a sequence of user activities and application responses. This data is similar to other known data such as Time Series [1] and Location data. Therefore we can use the same solutions used to identify patterns in similar data. For example, in time series data, algorithms such as Recurrent Neural Network (RNN) [2] and Long Short Term Memory (LSTM) [3] have been widely used to extract historical patterns. Both RNN and LSTM also have been used for location data to find the most common routes for trips using Global Positioning Systems (GPS) data, which is similar to finding test cases in a set of user actions. In this thesis, we used n-gram and time analysis for test case generation, and we aim to find more algorithms to apply to our data to make sure that we are finding the maximum number of valid test cases.

Identifying popular functionalities

Mariani and et al. investigated popular functionalities of applications such as login and CRUD operations and generated test cases with oracles for them. Their approach is limited to a handful number of user operations. By collecting user data, extend their work by identifying more popular functionalities with more accurate oracles could be a prospective contribution.

Improving ComVis

The user study results in Chapter 5 showed that although the tabular view helps participants be more accurate, it is not easy to use. A study on how to display data differences in a tabular view can improve ComVis's usability. Therefore, we intend to conduct research into different possible alternatives to present discrepancies in two data sets. Moreover, we would like to evaluate more data visualizations such as line plots showing only differences to see how users respond to them.

[1] https://en.wikipedia.org/wiki/Time_series
[2] https://en.wikipedia.org/wiki/Recurrent_neural_network
[3] https://en.wikipedia.org/wiki/Long_short-term_memory

Reference

[1] John G Adair and Joyce S Epstein. "Verbal cues in the mediation of experimenter bias". In: *Psychological Reports* 22.3_suppl (1968), pp. 1045–1053.

[2] Abdullah Alsharif, Gregory M Kapfhammer, and Phil McMinn. "DOMINO: Fast and effective test data generation for relational database schemas". In: *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2018, pp. 12–22.

[3] Bashair Althani and Souheil Khaddaj. "Systematic review of legacy system migration". In: *2017 16th International Symposium on Distributed Computing and Applications to Business, Engineering and Science (DCABES)*. IEEE. 2017, pp. 154–157.

[4] Domenico Amalfitano et al. "Using GUI ripping for automated testing of Android applications". In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM. 2012, pp. 258–261.

[5] Stephan Arlt, Cristiano Bertolini, and Martin Schäf. "Behind the scenes: an approach to incorporate context in GUI test case generation". In: *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE. 2011, pp. 222–231.

[6] Stephan Arlt et al. "Lightweight static analysis for GUI testing". In: *2012 IEEE 23rd International Symposium on Software Reliability Engineering*. IEEE. 2012, pp. 301–310.

[7] Mohammad Bajammal and Ali Mesbah. "Web Canvas Testing through Visual Inference". In: *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2018, pp. 193–203.

[8] S Balaji and M Sundararajan Murugaiyan. "Waterfall vs. V-Model vs. Agile: A comparative study on SDLC". In: *International Journal of Information Technology and Business Management* 2.1 (2012), pp. 26–30.

[9] Manuel Breugelmans and Bart Van Rompaey. "Testq: Exploring structural and maintenance characteristics of unit test suites". In: *WASDeTT-1: 1st International Workshop on Advanced Software Development Tools and Techniques*. 2008.

[10] Lionel Briand and Yvan Labiche. "A UML-based approach to system testing". In: *Software and Systems Modeling* 1.1 (2002), pp. 10–42.

[11] Penelope A Brooks and Atif M Memon. "Automated GUI testing guided by usage profiles". In: *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM. 2007, pp. 333–342.

[12] The Digital Cave. *Buddi*. `http://buddi.digitalcave.ca/`.

[13] William B Cavnar, John M Trenkle, et al. "N-gram-based text categorization". In: *Proceedings of SDAIR-94, 3rd annual symposium on document analysis and information retrieval*. Vol. 161175. Citeseer. 1994.

[14] Tsung-Hsiang Chang, Tom Yeh, and Robert C Miller. "GUI testing using computer vision". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM. 2010, pp. 1535–1544.

[15] CL Philip Chen and Chun-Yang Zhang. "Data-intensive applications, challenges, techniques and technologies: A survey on Big Data". In: *Information sciences* 275 (2014), pp. 314–347.

[16] Lin Cheng et al. "GUICat: GUI testing as a service". In: *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2016, pp. 858–863.

[17] Elliot J. Chikofsky and James H Cross. "Reverse engineering and design recovery: A taxonomy". In: *IEEE software* 7.1 (1990), pp. 13–17.

[18] Victoria Clarke, Virginia Braun, and Nikki Hayfield. "Thematic analysis". In: *Qualitative psychology: A practical guide to research methods* (2015), pp. 222–248.

[19] E. F. Collins and V. F. de Lucena. "Software Test Automation practices in agile development environment: An industry experience report". In: *2012 7th International Workshop on Automation of Software Test (AST)*. 2012, pp. 57–63.

[20] M. D'Ambros, M. Lanza, and M. Pinzger. ""A Bug's Life" Visualizing a Bug Database". In: *2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*. 2007, pp. 113–120.

[21] Michller Walker Dan Hannigan. "World Quality Report 2015-16". In: (2016).

[22] Stéphane Ducasse, Tudor Girba, and Roel Wuyts. "Object-oriented legacy system trace-based logic testing". In: *Conference on Software Maintenance and Reengineering (CSMR'06)*. IEEE. 2006, 10–pp.

[23] CR Ellis et al. "Simulation of snow accumulation and melt in needleleaf forest environments". In: *Hydrology and Earth System Sciences* 14.6 (2010), pp. 925–940.

[24] Markus Ermuth and Michael Pradel. "Monkey see, monkey do: effective generation of GUI tests with inferred macro events". In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM. 2016, pp. 82–93.

[25] X Fang et al. "Prediction of snowmelt derived streamflow in a wetland dominated prairie basin." In: *Hydrology & Earth System Sciences Discussions* 7.1 (2010).

[26] Mark Fewster and Dorothy Graham. *Software test automation*. Addison-Wesley Reading, 1999.

[27] Davide Fucci et al. "An external replication on the effects of test-driven development using a multi-site blind analysis approach". In: *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 2016, pp. 1–10.

[28] Svetoslav Ganov et al. “Event listener analysis and symbolic execution for testing GUI applications”. In: *International Conference on Formal Engineering Methods*. Springer. 2009, pp. 69–87.

[29] Martin Girschick and Tu Darmstadt. “Difference detection and visualization in UML class diagrams”. In: *Technical university of darmstadt technical report TUD-CS-2006-5* (2006), pp. 1–15.

[30] Dorothy Graham, Erik Van Veenendaal, and Isabel Evans. *Foundations of software testing: ISTQB certification*. Cengage Learning EMEA, 2008.

[31] Anthony G Greenwald. “Within-subjects designs: To use or not to use?” In: *Psychological Bulletin* 83.2 (1976), p. 314.

[32] R. K. Gupta, P. Manikreddy, and A. GV. “Challenges in Adapting Agile Testing in a Legacy Product”. In: *2016 IEEE 11th International Conference on Global Software Engineering (ICGSE)*. 2016, pp. 104–108.

[33] Maen Hammad et al. “Multiview Visualization of Software Testing Results”. In: *International Journal of Computing and Digital Systems* 9.1 (2020).

[34] Mark Harman. “Automated test data generation using search based software engineering”. In: *Second International Workshop on Automation of Software Test (AST'07)*. IEEE. 2007, pp. 2–2.

[35] Bill Hasling, Helmut Goetz, and Klaus Beetz. “Model based testing of system requirements using UML use case models”. In: *2008 1st International Conference on Software Testing, Verification, and Validation*. IEEE. 2008, pp. 367–376.

[36] “https://docs.microsoft.com/en-us/visualstudio/test/use-ui-automation-to-test-your-code?view=vs-2019”. In: ().

[37] “https://github.com/2gis/Winium.Desktop”. In: ().

[38] H. Hungar, T. Margaria, and B. Steffen. “Test-based model generation for legacy systems”. In: *International Test Conference, 2003. Proceedings. ITC 2003.* Vol. 2. 2003, 150–159 Vol.2.

[39] K. E. Isaacs et al. “Combing the Communication Hairball: Visualizing Parallel Execution Traces using Logical Time”. In: *IEEE Transactions on Visualization and Computer Graphics* 20.12 (2014), pp. 2349–2358.

[40] James A. Jones, Mary Jean Harrold, and John Stasko. “Visualization of Test Information to Assist Fault Localization”. In: *Proceedings of the 24th International Conference on Software Engineering*. ICSE '02. Orlando, Florida: Association for Computing Machinery, 2002, pp. 467–477. ISBN: 158113472X. DOI: 10.1145/581339.581397. URL: https://doi.org/10.1145/581339.581397.

[41] Shrinivas Joshi and Alessandro Orso. “SCARPE: A technique and tool for selective capture and replay of program executions”. In: *2007 IEEE International Conference on Software Maintenance*. IEEE. 2007, pp. 234–243.

[42] Ivan Jovanovikj, Marvin Grieger, and Enes Yigitbas. "Towards a model-driven method for reusing test cases in software migration projects". In: *Softwaretechnik-Trends, Proceedings of the 18th Workshop Software-Reengineering & Evolution (WSRE) & 7th Workshop Design for Future (DFF)*. Vol. 32. 2. 2016, pp. 65–66.

[43] Ivan Jovanovikj et al. "Model-Driven Test Case Migration: The Test Case Reengineering Horseshoe Model". In: *International Conference on Advanced Information Systems Engineering*. Springer. 2018, pp. 133–147.

[44] Eun Ha Kim, Jong Chae Na, and Seok Moon Ryoo. "Implementing an effective test automation framework". In: *2009 33rd Annual IEEE International Computer Software and Applications Conference*. Vol. 2. IEEE. 2009, pp. 534–538.

[45] Edward Kit and Susannah Finzi. *Software testing in the real world: improving the process*. ACM Press/Addison-Wesley Publishing Co., 1995.

[46] Rainer Koschke. "Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey". In: *Journal of Software Maintenance and Evolution: Research and Practice* 15.2 (2003), pp. 87–109.

[47] Jiri Kovalsky. *rachota*. `http://rachota.sourceforge.net/en/index.html`. 2013.

[48] Nan Li et al. "Applying combinatorial test data generation to big data applications". In: *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2016, pp. 637–647.

[49] Hsin-I Liao, Su-Ling Yeh, and Shinsuke Shimojo. "Novelty vs. familiarity principles in preference decisions: task-context of past experience matters". In: *Frontiers in psychology* 2 (2011), p. 43.

[50] F Lindstrom and I Jacobson. "Re-engineering of old systems to an object-oriented architecture,"". In: *OOPSLA, ACM* (1991).

[51] Chien Hung Liu et al. "Capture-replay testing for android applications". In: *2014 International Symposium on Computer, Consumer and Control*. IEEE. 2014, pp. 1129–1132.

[52] Kuruvilla Lukose et al. "Design Study for Creating Pathfinder: A Visualization Tool for Generating Software Test Plans using Model based Testing." In: *VISIGRAPP (3: IVAPP)*. 2018, pp. 289–300.

[53] Ke Mao, Mark Harman, and Yue Jia. "Sapienz: Multi-objective automated testing for Android applications". In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM. 2016, pp. 94–105.

[54] Leonardo Mariani, Mauro Pezzè, and Daniele Zuddas. "Augusto: Exploiting popular functionalities for the generation of semantic GUI tests with oracles". In: *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE. 2018, pp. 280–290.

[55] Leonardo Mariani et al. “Autoblacktest: Automatic black-box testing of interactive applications”. In: *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE. 2012, pp. 81–90.

[56] Gale L Martin and Kenneth G Corl. “System response time effects on user productivity”. In: *Behaviour & Information Technology* 5.1 (1986), pp. 3–13.

[57] Sonali Mathur and Shaily Malik. “Advancements in the V-Model”. In: *International Journal of Computer Applications* 1.12 (2010), pp. 29–34.

[58] Martin Mc Hugh et al. “An agile v-model for medical device software development to overcome the challenges with plan-driven software development lifecycles”. In: *2013 5th International Workshop on Software Engineering in Health Care (SEHC)*. IEEE. 2013, pp. 12–19.

[59] Bree McEwan. “Sampling and validity”. In: *Annals of the International Communication Association* (2020), pp. 1–13.

[60] Phil McMinn. “Search-based software test data generation: a survey”. In: *Software testing, Verification and reliability* 14.2 (2004), pp. 105–156.

[61] Phil McMinn et al. “SchemaAnalyst: Search-based test data generation for relational database schemas”. In: *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2016, pp. 586–590.

[62] Atif Memon, Ishan Banerjee, and Adithya Nagarajan. “What test oracle should I use for effective GUI testing?” In: *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.* IEEE. 2003, pp. 164–173.

[63] Leonel Merino and Oscar Nierstrasz. “The medium of visualization for software comprehension”. PhD thesis. Universität Bern, 2018.

[64] Christophe Meudec. “ATGen: automatic test data generation using constraint logic programming and symbolic execution”. In: *Software testing, verification and reliability* 11.2 (2001), pp. 81–96.

[65] Ian Molyneaux. *The art of application performance testing: from strategy to tools.* ” O’Reilly Media, Inc.”, 2014.

[66] Jesús Morán, Claudio de la Riva, and Javier Tuya. “Testing data transformations in MapReduce programs”. In: *Proceedings of the 6th International Workshop on Automating Test Case Design, Selection and Evaluation*. 2015, pp. 20–25.

[67] Patrick Morrison and Emerson Murphy-Hill. “Is programming knowledge related to age?” In: *Companion to the Working Conference on Mining Software Repositories*. Citeseer. 2013, pp. 1–4.

[68] H. Mumtaz et al. “Exploranative Code Quality Documents”. In: *IEEE Transactions on Visualization and Computer Graphics* 26.1 (2020), pp. 1129–1139.

[69] Emerson Murphy-Hill and Andrew P. Black. "An Interactive Ambient Visualization for Code Smells". In: *Proceedings of the 5th International Symposium on Software Visualization*. SOFTVIS '10. Salt Lake City, Utah, USA: Association for Computing Machinery, 2010, pp. 5–14. ISBN: 9781450300285. DOI: 10.1145/1879211.1879216. URL: https://doi.org/10.1145/1879211.1879216.

[70] H. Nakagawa, S. Matsui, and T. Tsuchiya. "A Visualization of Specification Coverage Based on Document Similarity". In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 2017, pp. 136–138.

[71] Bao N Nguyen et al. "GUITAR: an innovative tool for automated testing of GUI-driven software". In: *Automated software engineering* 21.1 (2014), pp. 65–105.

[72] Rudolfs Opmanis, Paulis Kikusts, and Martins Opmanis. "Visualization of large-scale application testing results". In: *Baltic Journal of Modern Computing* 4.1 (2016), p. 34.

[73] Lionel Page and Katie Page. "Last shall be first: A field study of biases in sequential performance evaluation on the Idol series". In: *Journal of Economic Behavior & Organization* 73.2 (2010), pp. 186–198.

[74] Roy P Pargas, Mary Jean Harrold, and Robert R Peck. "Test-data generation using genetic algorithms". In: *Software testing, verification and reliability* 9.4 (1999), pp. 263–282.

[75] Mauro Pezzè, Paolo Rondena, and Daniele Zuddas. "Automatic GUI testing of desktop applications: an empirical assessment of the state of the art". In: *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*. ACM. 2018, pp. 54–62.

[76] Margot Postema and Heinz W Schmidt. "Reverse engineering and abstraction of legacy systems". In: *INFORMATICA-LJUBLJANA*- 22 (1998), pp. 359–371.

[77] Margot Postema and Heinz W Schmidt. "Reverse engineering and abstraction of legacy systems". In: *INFORMATICA-LJUBLJANA*- 22 (1998), pp. 359–371.

[78] Rhoslyn Roebuck Williams et al. "Subtle Sensing: Detecting Differences in the Flexibility of Virtually Simulated Molecular Objects". In: *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems*. CHI EA '20. Honolulu, HI, USA: Association for Computing Machinery, 2020, pp. 1–8. ISBN: 9781450368193. DOI: 10.1145/3334480.3383026. URL: https://doi.org/10.1145/3334480.3383026.

[79] Linda H Rosenberg and Lawrence E Hyatt. "Software re-engineering". In: *Software Assurance Technology Center* (1996), pp. 2–3.

[80] Robert Rosenthal and Kermit L Fode. "Psychology of the scientist: V. Three experiments in experimenter bias". In: *Psychological Reports* 12.2 (1963), pp. 491–511.

[81] Jonathan Saddler and Myra B Cohen. "EventFlowSlicer: Goal based test generation for graphical user interfaces". In: *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation*. ACM. 2016, pp. 8–15.

[82] Zahra Sahaf et al. "When to automate software testing? decision support based on system dynamics: an industrial case study". In: *Proceedings of the 2014 International Conference on Software and System Process*. ACM. 2014, pp. 149–158.

[83] Reinhard Schauer and Rudolf K Keller. "Pattern visualization for software comprehension". In: *Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No. 98TB100242)*. IEEE. 1998, pp. 4–12.

[84] Rupert Sheldrake. "Experimenter effects in scientific research: How widely are they neglected". In: *Journal of Scientific Exploration* 12.1 (1998), pp. 73–78.

[85] Liu Shuping and Pang Ling. "The research of V model in testing embedded software". In: *2008 International Conference on Computer Science and Information Technology*. IEEE. 2008, pp. 463–466.

[86] Boyce Sigweni and Martin Shepperd. "Using blind analysis for software engineering experiments". In: *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*. 2015, pp. 1–6.

[87] Adrian Smith. *Universal Password Manager*. `http://upm.sourceforge.net/index.html`. 2013.

[88] Harry M Sneed. "Estimating the costs of a reengineering project". In: *12th Working Conference on Reverse Engineering (WCRE'05)*. IEEE. 2005, 9–pp.

[89] Harry M Sneed. "Risks involved in reengineering projects". In: *Sixth Working Conference on Reverse Engineering (Cat. No. PR00303)*. IEEE. 1999, pp. 204–211.

[90] Andreas Spillner and H Bremenn. "The W-MODEL. Strengthening the bond between development and test". In: *Int. Conf. on Software Testing, Analysis and Review*. 2002, pp. 15–17.

[91] Suresh Thummalapenta et al. "Automating test automation". In: *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press. 2012, pp. 881–891.

[92] S. Urata and T. Katayama. "Proposal of Testing Diagrams for Visualizing Test Cases". In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. 2013, pp. 483–484.

[93] Marlon Vieira et al. "Automation of GUI testing using a model-driven approach". In: *Proceedings of the 2006 international workshop on Automation of software test*. ACM. 2006, pp. 9–14.

[94] Tanja EJ Vos et al. "Testar: Tool support for test automation at the user interface level". In: *International Journal of Information System Modeling and Design (IJISMD)* 6.3 (2015), pp. 46–83.

[95] F. Wang and W. Du. "A Test Automation Framework Based on WEB". In: *2012 IEEE/ACIS 11th International Conference on Computer and Information Science*. 2012, pp. 683–687.

[96] Lee White and Husain Almezen. "Generating test cases for GUI responsibilities using complete interaction sequences". In: *Proceedings 11th International Symposium on Software Reliability Engineering. ISSRE 2000*. IEEE. 2000, pp. 110–121.

[97] Frank Wilcoxon. "Probability tables for individual comparisons by ranking methods". In: *Biometrics* 3.3 (1947), pp. 119–122.

[98] Qing Xie. "Developing cost-effective model-based techniques for GUI testing". In: *Proceedings of the 28th international conference on Software engineering*. ACM. 2006, pp. 997–1000.

[99] K. Xu et al. "CloudDet: Interactive Visual Analysis of Anomalous Performances in Cloud Computing Systems". In: *IEEE Transactions on Visualization and Computer Graphics* 26.1 (2020), pp. 1107–1117.

[100] Shamima Yeasmin, Chanchal K Roy, and Kevin A Schneider. "Interactive visualization of bug reports using topic evolution and extractive summaries". In: *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE. 2014, pp. 421–425.

[101] Y. Yoon, B. A. Myers, and S. Koo. "Visualization of fine-grained code change history". In: *2013 IEEE Symposium on Visual Languages and Human Centric Computing*. 2013, pp. 119–126.

[102] Jian Zhang, Xu Chen, and Xiaoliang Wang. "Path-oriented test data generation using symbolic execution and constraint solving techniques". In: *Proceedings of the Second International Conference on Software Engineering and Formal Methods, 2004. SEFM 2004.* IEEE. 2004, pp. 242–250.

[103] Lu Zhang et al. "Time-aware test-case prioritization using integer linear programming". In: *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM. 2009, pp. 213–224.

[104] Valerie Zhao et al. "Visualizing Differences to Improve End-User Understanding of Trigger-Action Programs". In: *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems*. CHI EA '20. Honolulu, HI, USA: Association for Computing Machinery, 2020, pp. 1–10. ISBN: 9781450368193. DOI: 10.1145/3334480.3382940. URL: https://doi.org/10.1145/3334480.3382940.

[105] Xiaochun Zhu et al. "A test automation solution on gui functional test". In: *2008 6th IEEE International Conference on Industrial Informatics*. IEEE. 2008, pp. 1413–1418.

Appendix A

User Study

A.1 Schema of the User Study

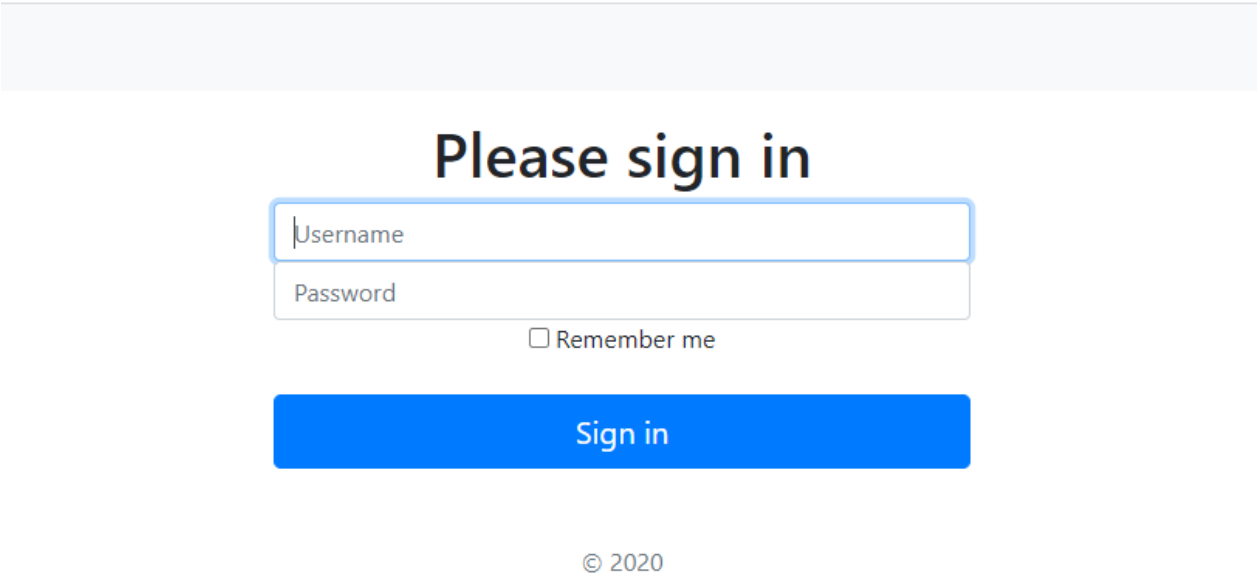


Figure A.1: Log in Page

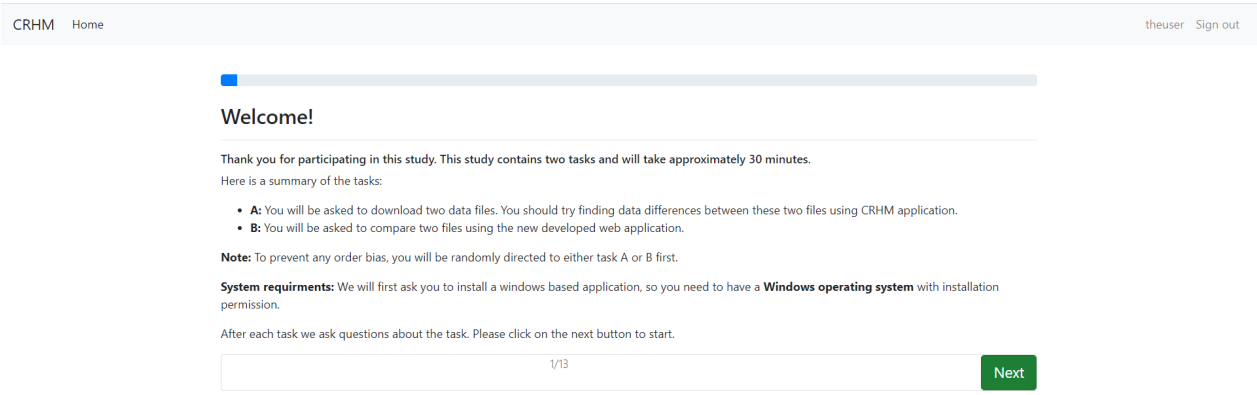


Figure A.2: Welcome page

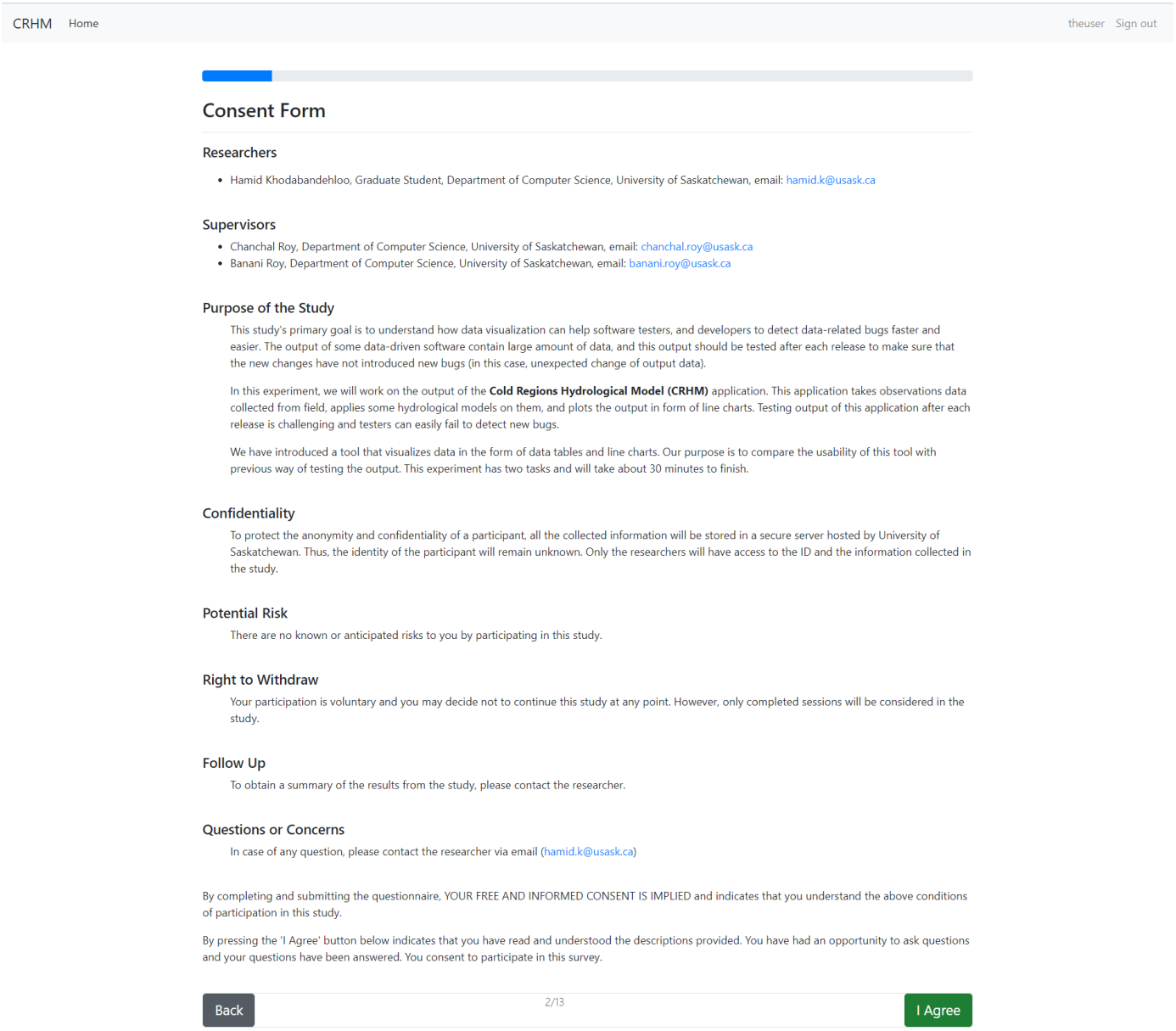


Figure A.3: Consent Form

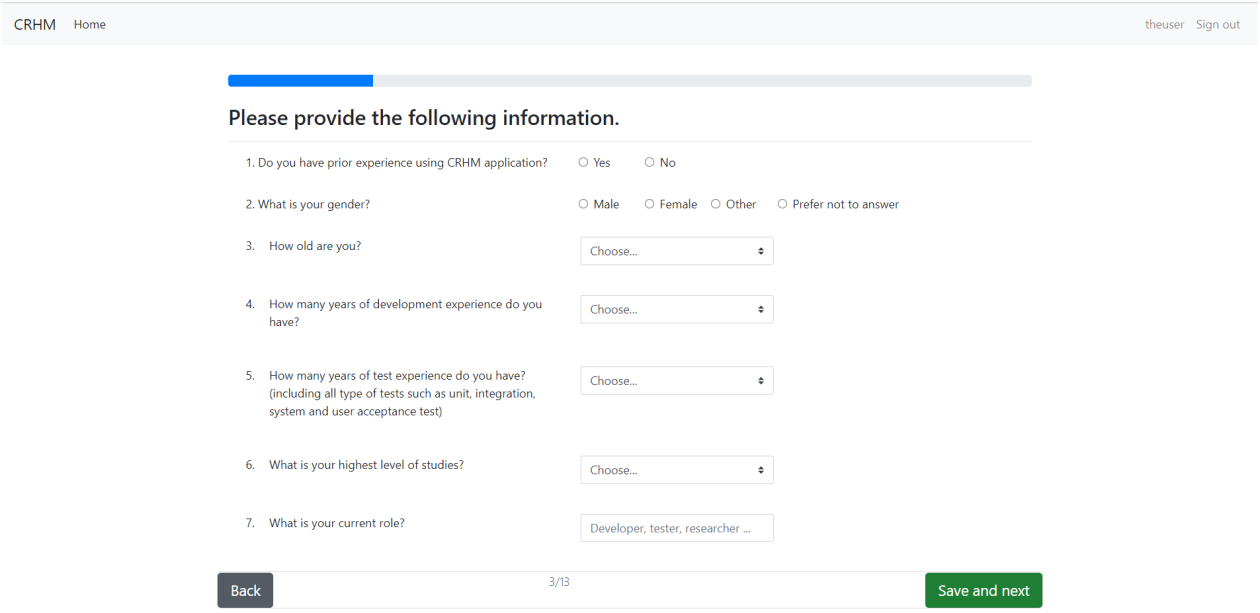


Figure A.4: User Information

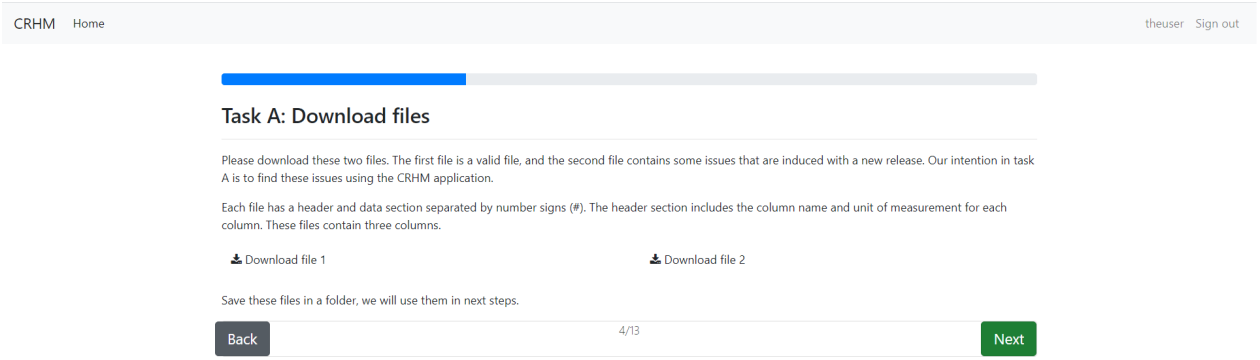


Figure A.5: Download data file

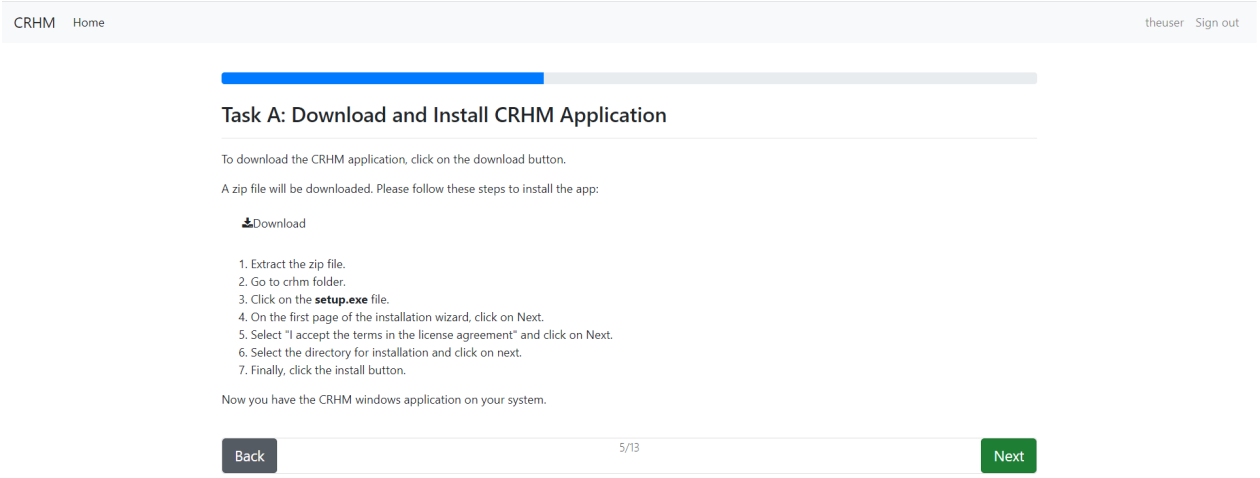


Figure A.6: CRHM guid

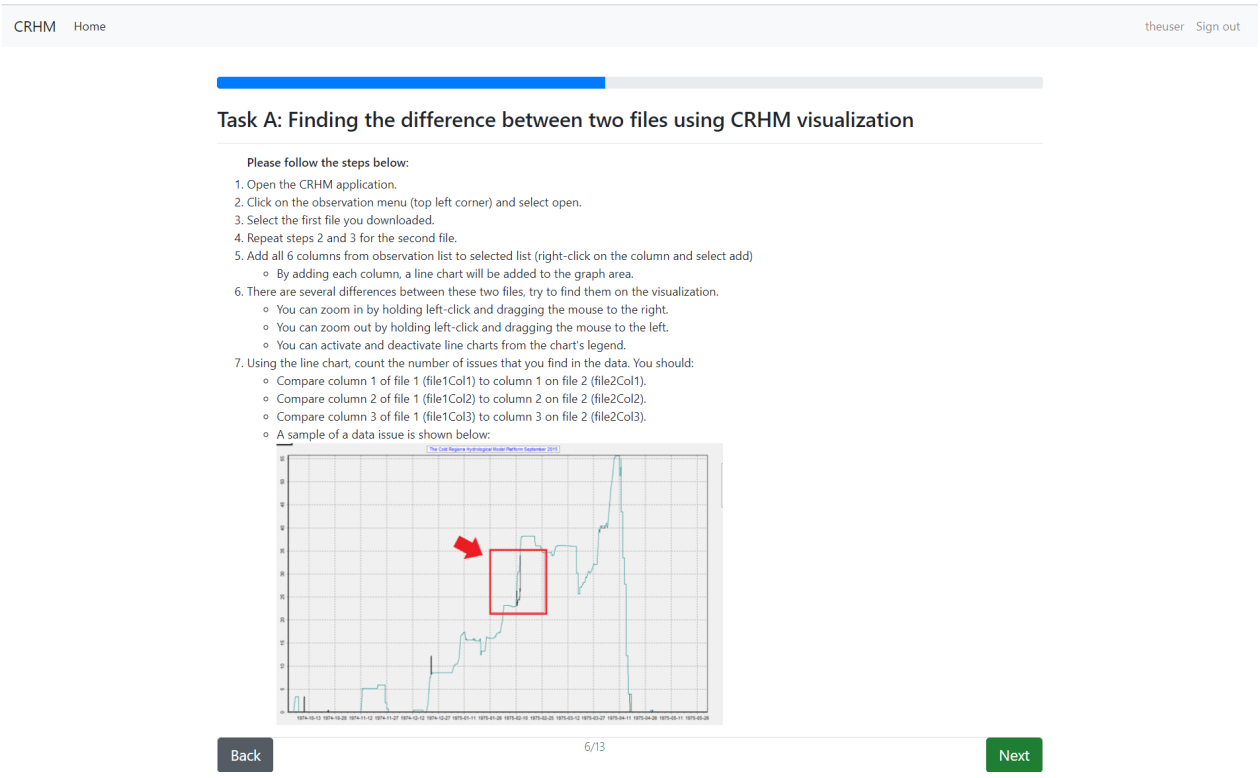


Figure A.7: Ask A guid

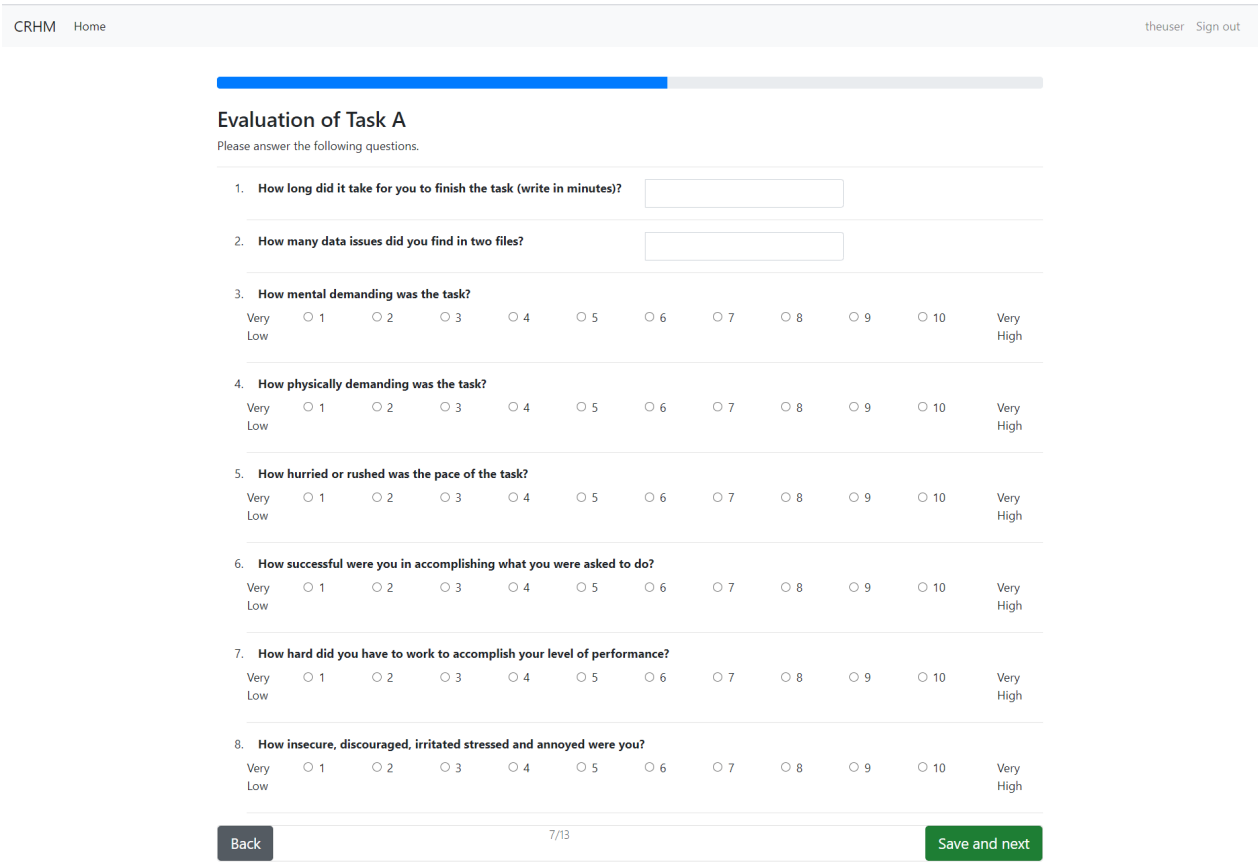


Figure A.8: Task A evaluation

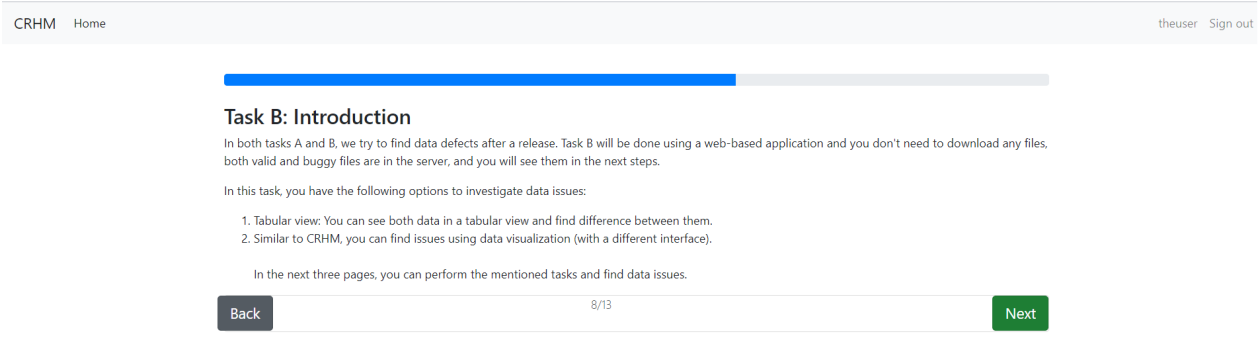


Figure A.9: Task B introduction

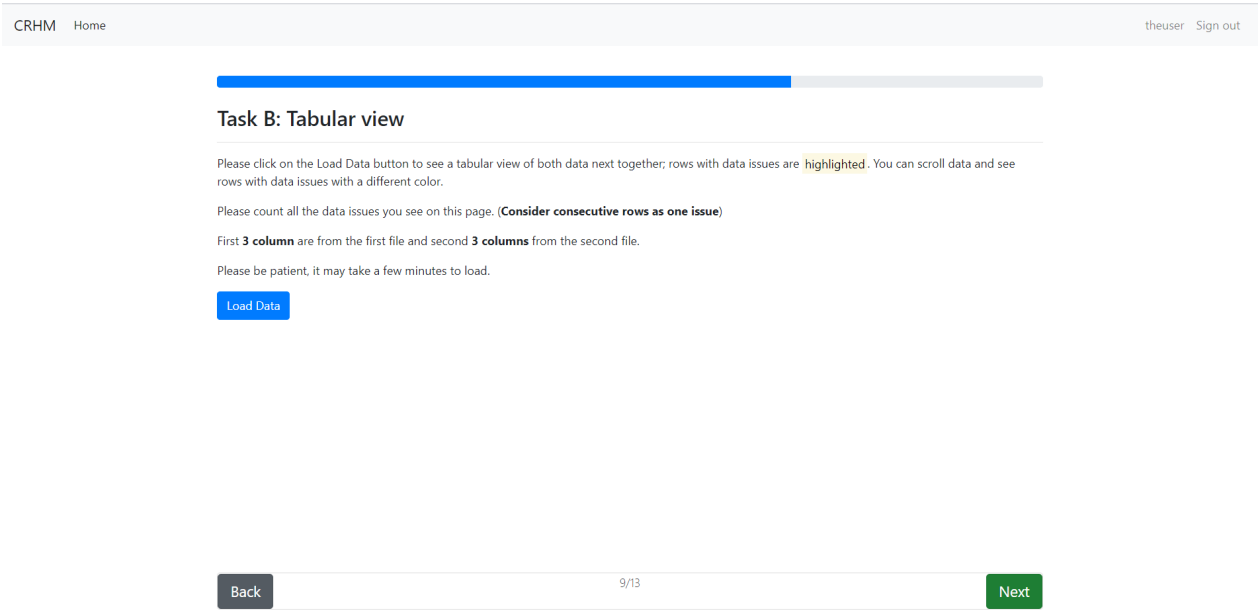


Figure A.10: Tabular view

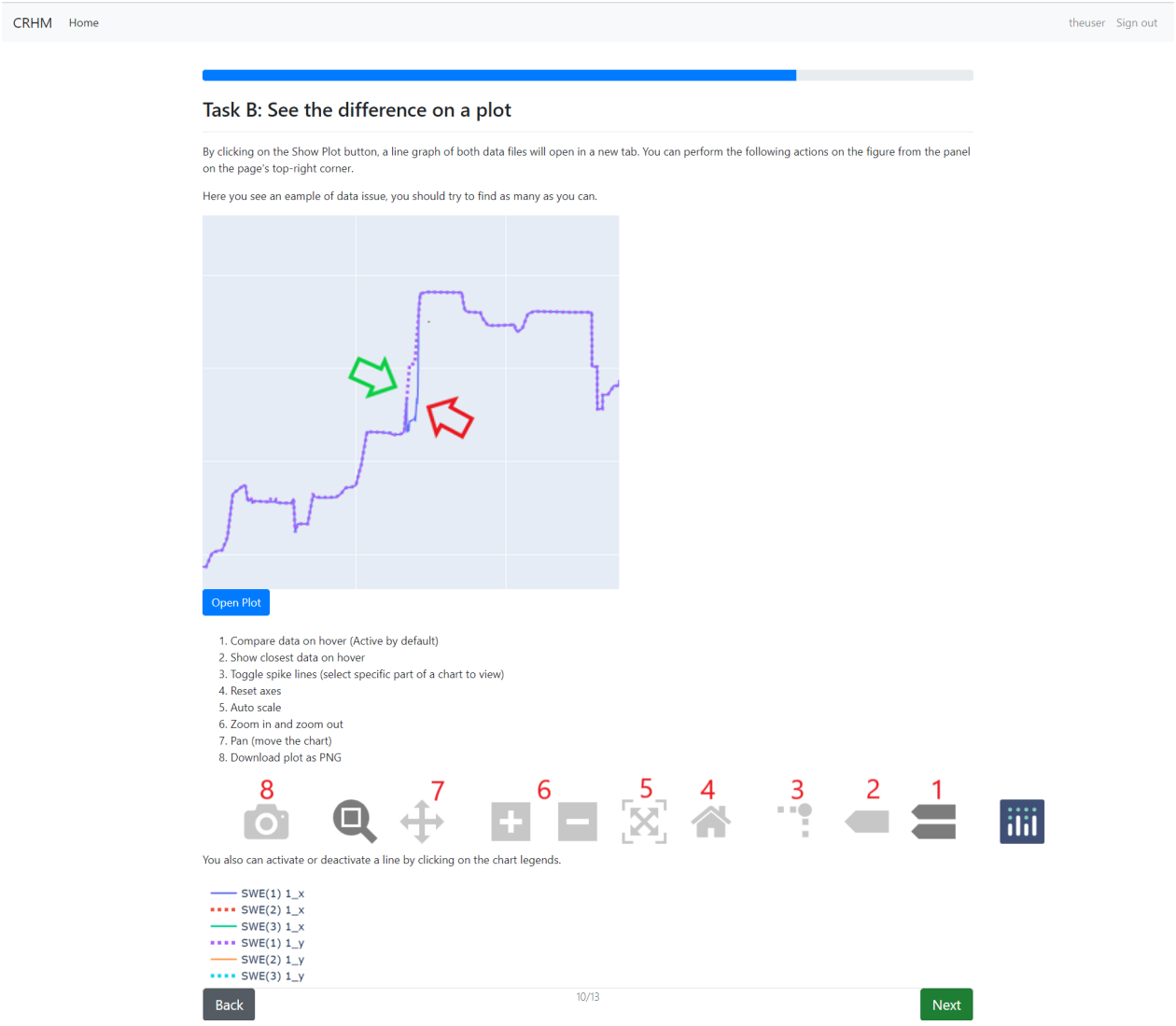


Figure A.11: Task B plot guid

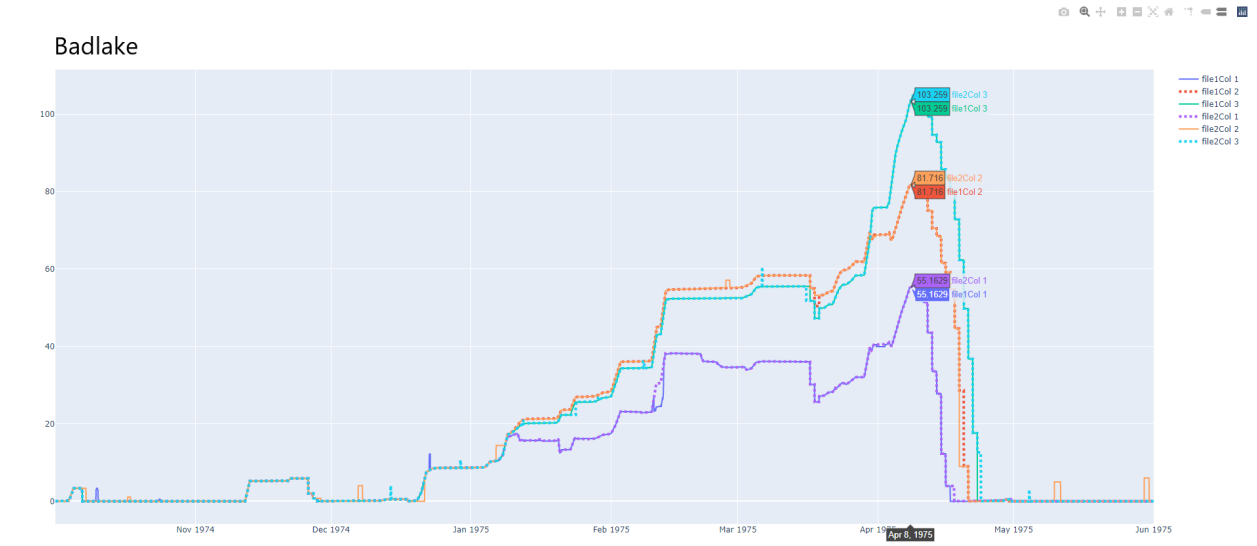


Figure A.12: Interactive line chart

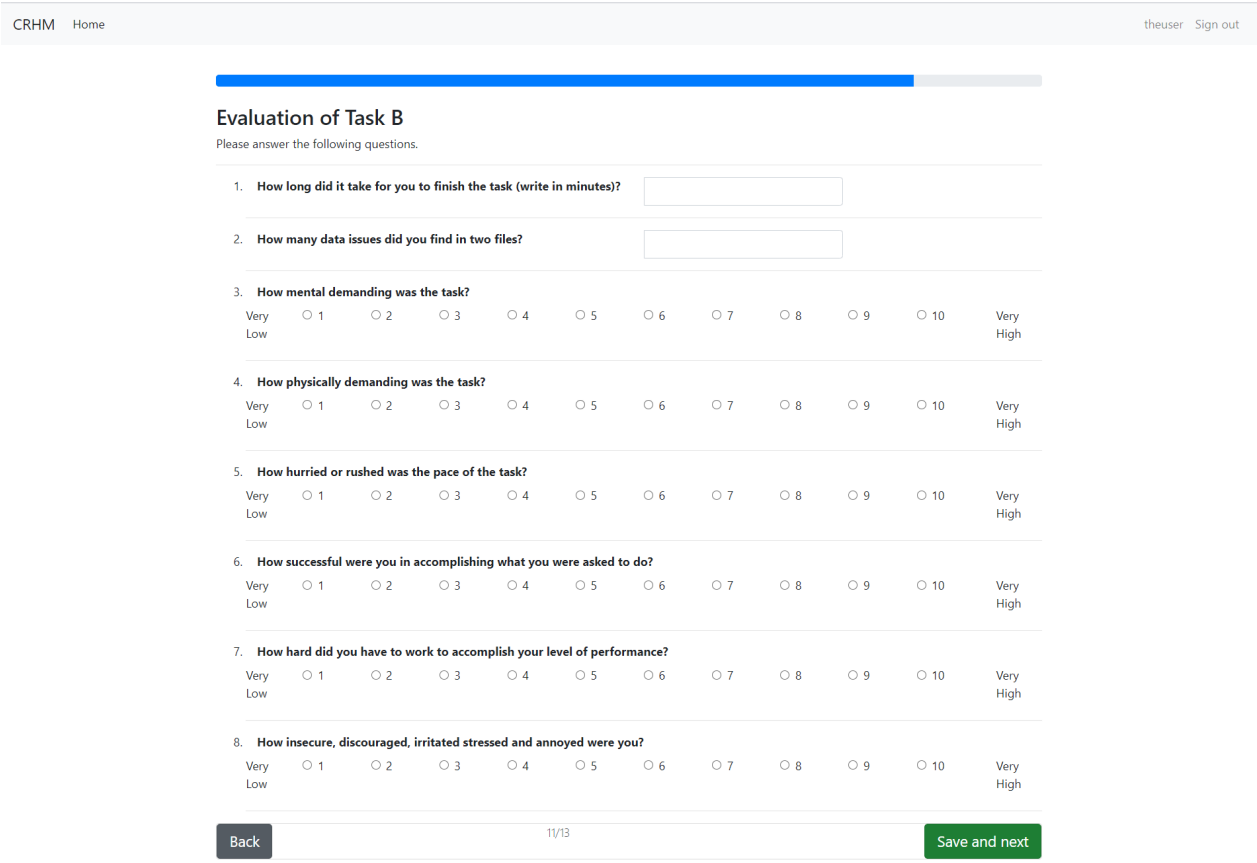


Figure A.13: Task B evaluation

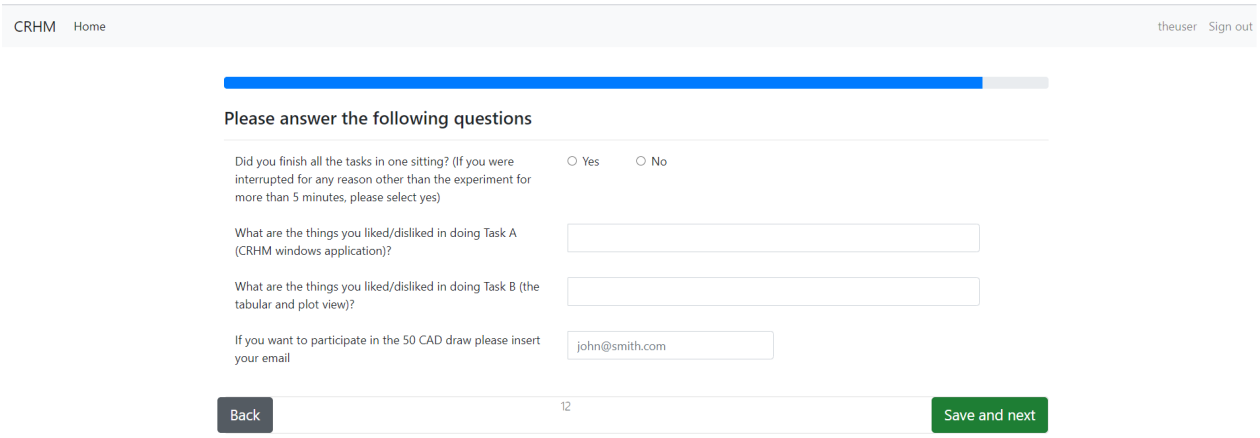


Figure A.14: Like and dislike question

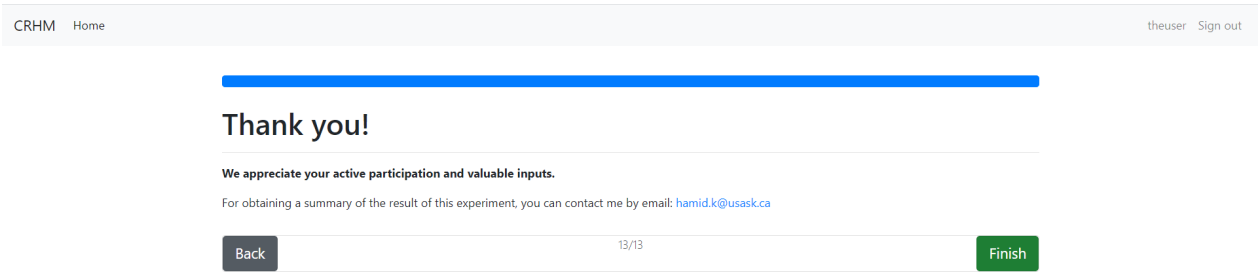


Figure A.15: Appreciation page