Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science

Supporting Exploratory Testing by Automated Navigation Using the Model of System Under Test

by

KAREL FRAJTÁK

A dissertation thesis submitted to
the Faculty of Electrical Engineering, Czech Technical University in Prague,
in partial fulfillment of the requirements for the degree of Doctor.

PhD Programme: Electrical Engineering and Information Technology
Branch of Study: Information Science and Computer Engineering

August 2017

Supervisor:
doc. Ing. Ivan Jelínek, CSc.
Department of Computer Science
Faculty of Electrical Engineering
Czech Technical University in Prague
Karlovo náměstí 13
121 35 Prague 2
Czech Republic

Abstract and Contributions

Exploratory Testing (ET) is software testing technique, which is applicable to software development projects, in which test basis (design documentation) is not available, or at least not complete and consistent enough to allow the creation of efficient test cases. The principle of this technique is simultaneous learning, creation of the test cases and testing of the explored system under test (SUT).

The key factor for the efficiency of this technique a documentation of explored tester's path in the SUT and the exercised test cases. When this is not being documented properly, ET loses its potential efficiency. Documentation of the explored paths in the SUT also allows more accurate reports of found defects, together with the possibility of the more systematic creation of the test cases during the exploratory testing process, preventing potential duplicities in executed tests. Created test cases can be used later in the next testing phases (retesting of fixed defects or regression testing, for instance).

Currently, a large ratio of web applications is usually developed without any or sufficient underlying models. In the most of the cases, this is a consequence of required low development costs and usually short time-to-market in the competitive software development environment. Nevertheless, the model is still implicitly present in the SUT code and with proper techniques, it is possible to reconstruct it from the SUT.

Using the browser extension and server application, the framework automatically records selected actions of the exploratory testers in the SUT. Based on this recorded data, a screen-flow based model of the SUT is incrementally created and automatically updated. Based on this model and tester's position in the SUT during the testing process, the framework dynamically creates navigational test cases guiding the testers in the SUT and supports its efficient exploration.

The recorded model consists of parts representing SUT pages, forms, input elements of the pages, and action elements as submit buttons and links. Besides that, home page, error pages and transition between SUT pages are defined. The model constructs are accompanied by technical information and meta–data to support the model construction

and generation of navigational test cases.

In the test leader role, the tester can also extend the model with additional meta–data. For example, he can prioritize certain pages or selected actions as a result of a risk analysis of the SUT. Moreover, he can define equivalence classes determining the suitable test data for particular inputs (e.g. text boxes in the forms) for the later generation of the navigational test cases. The SUT model is described formally and defined constructs are used to document the proposal and the experiments verifying the framework functionality and practical efficiency.

During the dynamical generation of navigational test cases, several navigational strategies can be selected. These strategies are based on combinations of the inputs like the parts of the SUT previously explored by an individual tester or all testers in the team, priority of the particular pages marked by the test team leader, or complexity of explored page elements.

Performed experiments show that Exploratory Testing aided by this automated support is less resource demanding than Exploratory Testing performed manually only. With this support, also larger parts of the SUT were explored by the experiment participants in the defined time. Also, as defect injection experiments have shown, that the proposed automated support helped the testing teams to detect more inserted software defects than ET performed as a manual process only.

In particular, the main contributions of the dissertation thesis are as follows:

- Design and experimental implementation of automated method, which makes Exploratory Testing process more efficient in terms of spent resources, extent of explored SUT and found defects.
- Innovative combination of Exploratory Testing, Model–Based Testing and Reverse–Engineering.
- Definition of formal model of the underlying system under test, which serves as a basis for the framework functionality.
- Design of initial navigational strategies, ranking functions, and test data strategies, used in the framework. These strategies are supporting the individual as well as team exploratory testing.
- Practical applicability of the proposed framework to industrial software development and testing projects.

Keywords:

Model–Based Testing, Exploratory Testing, Test Automation, System Under Test Model, Test Data, Test Management.

Abstrakt

Metoda průzkumného testovaní (angl. Exploratory Testing, dále jen ET) je metoda přístupu k testování softwarových projektů vhodná pro situace, ve kterých není k dispozici žádná návrhová dokumentace, nebo je neúplná nebo nekonzistentní do té míry, aby ji bylo možné využít pro tvorbu testovacích scénářů. Principem tohoto přístupu je souběžné prozkoumávání testovaného systému, tvorba testovacích scénářů a testování tohoto systému.

Pro zajištění efektivity této testovací techniky je klíčová dokumentace prozkoumaných částí systému i vykonaných testovacích scénářů. Pokud není zajištěna, efektivita této techniky výrazně klesá. Tato dokumentace umožňuje i detailnější a přesnějších hlášení zjištěných defektů v testovaném softwaru. Další výhodou je systematičtější přístup k vytváření testovacích scénářů v průběhu průzkumného testování, který omezí případné duplicity v testech. Scénáře vytvořené v této fázi, pak mohou být použity později v dalších kolech testování, například při kontrolním testování opravených defektů nebo při regresním testování.

Aktuálně je řada webových systémů vyvíjena bez jakéhokoliv modelu nebo s pomocí nedostatečného či neúplného modelu. Tato situace je často důsledkem vysoce konkurenčního prostředí, ve kterým vývoj softwarových aplikací probíhá, tlaku na redukci nákladů na vývoj a nutností rychlého dodání výsledného produktu na trh. Nicméně model, který potřebujeme pro efektivní vytváření testovacích scénářů, je v principu implicitně přítomen v kódu testované aplikace a vhodnou technikou je možné jej z tohoto kódu zrekonstruovat.

Rekonstrukci tohoto modelu, společně se sledováním aktivit testera v testovaném systému a připojenými metadaty, lze využít pro automatickou navigaci tohoto testera v průběhu průzkumného testování. Tato podpora snižuje náklady na nutné pořizování dokumentace v průběhu testování, čímž zvyšuje transparentnost a efektivitu průzkumného testování. V této práci představujeme model a framework, který tuto podporu poskytuje.

Rozšiřující modul webového prohlížeče společně se serverovou částí umožňuje frameworku automaticky nahrávat akce prováděné testerem v testovaném systému. Nahrávaná data slouží k průběžnému vytváření modelu testovaného systému a k jeho aktualizaci.

Framework dynamicky vytváří navigační testovací scénáře na základě dostupného modelu a aktuální pozice testera v systému a tím zajišťuje vyšší efektivitu průzkumného testování.

Vytvořený model se skládá z částí reprezentujících jednotlivé stránky systému a jejich elementy jako jsou formuláře, vstupní pole, odkazy a tlačítka pro odesílání dat formulářů umístěná na těchto stránkách. Dále jsou jeho součástí domácí a chybová stránka a přechody mezi stránkami. Jednotlivé části modelu mohou být doplněny technickými detaily a metadaty, které jsou dále použity při tvorbě modelu nebo testovacích scénářů.

Tyto doplňující informace může do modelu přidávat vedoucí skupiny testerů. Na základě analýzy důležitosti jednotlivých částí testovaného systému může například určit prioritu pro vybrané stránky nebo akce. Dále může definovat třídy ekvivalence pro data zadávaná do vstupních polí na jednotlivých stránkách. Tyto informace jsou frameworkem využity při generování navigačních testovacích scénářů, jejichž součástí jsou doporučená testovací data.

Formálně popsané konstrukty modelu testovaného systému jsou v této práci použity jak pro dokumentaci vytvořeného frameworku, tak pro formulaci metrik použitých při experimentech ověřujících jeho funkčnost a praktickou použitelnost.

V průběhu dynamického generování navigačních testovacích scénářů může být použito několik navigačních strategií, které jsou popsány v textu práce. Tyto strategie využívají řadu vstupů, jako je informace o předchozích návštěvách dané stránky konkrétním testerem nebo jeho kolegy z daného týmu, priorita stránek nebo komplexnost stránek z hlediska počtu a druhu sledovaných prvků.

Provedené experimenty ověřily, že metoda průzkumného testovaní prováděného s automatickou podporou navrženého frameworku, ve srovnání s průzkumným testováním prováděným pouze manuálně, zvyšuje efektivitu této techniky v několika oblastech. Navržená automatická podpora snižuje čas potřebný na jednotlivé úkoly a vede testery k otestování větší části systému. Pokusy se zanesenými umělými defekty v testovaném systému ukázaly, že navržený framework pomohl testerům odhalit větší množství těchto zanesených defektů oproti skupině testerů pracujících bez jeho podpory.

Hlavními přínosy této disertační práce jsou:

- návrh a experimentální implementace frameworku pro zvýšení efektivity metody průzkumného testovaní z hlediska náročnosti na zdroje, rozsahu prozkoumaných částí systému a nalezených defektů,
- inovativní kombinace průzkumného testovaní, testování na základě modelu a reverzního inženýrství,
- definice formálního modelu testovaného systému, na jehož základě je postavena funkcionalita navrženého frameworku,

- pilotní návrh navigačních strategií a strategií pro doporučování testovacích data použitých ve frameworku. Tyto strategie jsou zaměřeny na podporu jak individuálního, tak týmového průzkumného testování, a
- praktická využitelnost navrženého frameworku v reálném vývoji softwarových produktů a testovacích projektech.

Klíčová slova:

Testování na základě modelu, průzkumné testování, automatizace testů, model testovaného systému, testovací data, řízení testů

Acknowledgements

First of all, I would like to express my gratitude to my dissertation thesis supervisor, doc. Ing. Ivan Jelínek, CSc. He has been a constant source of encouragement and insight during my research and helped me with numerous problems and professional advancements.

I would like to thank to my supervisor–specialist Ing. Miroslav Bureš, PhD. for sharing his knowledge in the field of software testing, being a mentor to me and providing great help with this work. Without him and his insightful comments the dissertation thesis would not have ever been finished. Special thanks go to the staff of the Department of Computer Science, who maintained a pleasant and flexible environment for my research. I would like to express special thanks to the department management for providing most of the funding for my research. Finally, my greatest thanks go to my family members, for their infinite patience and care.

My research has also been supported by the following grants:

- by the Technological Agency of Czech Republic, grant No. TH02010296 Quality Assurance System for Internet of Things Technology
- by the Grant Agency of the Czech Technical University in Prague, grant No.
  - SGS11/157/OHK3/3T/13 Automated support of manual testing of web software systems based on formal model of the application,
  - SGS14/076/OHK3/1T/13 Automated support of manual testing of web software systems based on formal model of the application,
  - SGS15/085/OHK3/1T/13 Automated support of manual testing of web software systems based on formal model of the application,
  - SGS16/090/OHK3/1T/13 Automated support for more efficient software testing,
  - SGS17/097/OHK3/1T/13 Automated methods for software testing.

Dedication

To my family and all the software testers in the world.

Contents

List of Figures

List of Tables

Abbreviations

AJAX	Asynchronous JavaScript And XML
API	Application programming interface
BPMN	Business Process Model and Notation
CIM	Computation Independent Model
CIT	Computation Independent Test
CR	Capture and Replay
CSS	Cascading Style Sheets
DOM	Document Object Model
EC	Equivalence Class
ET	Exploratory Testing
HTML	Hyper–text Markup Language
HTTP	Hyper–text Transfer Protocol
IFML	Interaction Flow Modeling Language
JSON	JavaScript Object Notation
MBT	Model–Based Testing
MDE	Model–Driven Engineering
OCL	Object Constraint Language
OMG	Object Management Group
PIM	Platform Independent Model
PIT	Platform Independent Test
SDLC	Software Development Life Cycle
SUT	System Under Test
UI	User Interface
UX	User Experience
UML	Unified Modeling Language
URL	Uniform Resource Locator
W3C	World Wide Web Consortium

Chapter 1

Introduction

In the last two decades, the dependency of various peoples' activities and social processes on software information systems increased significantly. Complexity and integration level of these systems is growing steadily, together with a growing demand for availability of these systems. From a commercial point of view, software projects are conducted in a highly competitive environment characterised by ever present pressure to optimise the costs and to make the time to market shorter. These requirements are frequently in contradiction. The cost optimisation and pressure to deliver a software product faster, together with its complexity can lead to a decline in final product quality, and, thus, production risks.

Such a situation creates a challenge to software testing and quality assurance methods, which have to be continuously evolved to work efficiently in the current state of the software development domain. For the systems with consistent and detailed underlying models and design specification, contemporary Model–Based Testing discipline offers a number of very well applicable methods to test these systems efficiently.

However, accurate, consistent and up–to–date design documentation or model is not present in a significant ratio of contemporary software projects in various business domains. Many factors can contribute to this state — the pressure to optimise the costs, use of naive software development methods, frequent change requests in the project scope, or lack of qualified software development resources can be examples of these factors. As a direct consequence, the efficiency of testing of these systems is also put at risk.

Setting an effective testing strategy for such projects is a challenge for software testing practitioner. More to that, this situation represents also an exciting research challenge. Can we provide an efficient and automated software testing methods also for these numerous software projects, in which the design documentation (and the information from which we can derive the testing scenarios) is biased and inconsistent to prevent standard testing process based on test cases prepared in advance before the test execution?

As a stage where the defects present in the system should be discovered and fixed, software testing represents an important part the software development life cycle [54]. The

current software testing is also considered to be a costly process. Study [35] estimates its costs to be even between 40% and 80% of the total costs of the development. As the First Boehm's law, based on empirical evidence from an extensive number of software development projects says, the price of defect detection and removal grows with the stage of the software project. Correcting a wrong requirement in a requirement catalogue can be a matter of half–hour discussion with the investor. Correcting a defect caused by this wrong requirement in production stage can impose extensive costs: It is not only really expensive because the application has to be reworked and released again, but it can annoy users to the point when they stop using the tested system, which can result in loss of potential income acquired by the tested system (see Fig. 1.1)[1]. An example of potential savings gained by a more efficient testing process and detection of defects are outlined in Fig. 1.2. Here, the shaded area represents the developers'costs due to an inadequate infrastructure for software testing [105].

To ensure efficient testing of software projects, various strategies can be taken. On one side of the specter, we can position the Model–Based Testing, where high coverage test cases are generated from a suitable model of the system under test (SUT). On the other side of the spectrum, Exploratory Testing (ET) can be positioned. Here, test case scenarios are not prepared in advance and are defined during the testing process. Thus, Exploratory Testing is suitable for projects, where test basis is not present, or at least not complete and consistent to the extent allowing the creation of efficient test cases. In this Dissertation Thesis, we focus on a fusion of these two approaches, despite the fact, that such a combination may seem unusual. To explain the reasons, why we did so, let's start with a more detailed introduction of the Exploratory Testing technique.

1.1 Exploratory Testing

Let's begin with an analogy. Imagine you have a new game and you start playing that game, if you don't want to start with the tutorial and you have already played a similar game in the past — then, intuitively, you have a basic understanding of how this new game is played and what the goal is. The task of an exploratory tester similar — using

[1]The legend for the figure:

RD Requirements gathering and analysis/architectural design

CU Coding/Unit testing

IS Integration and component system test

ER Early customer feedback/beta test programs

PR Post–production release

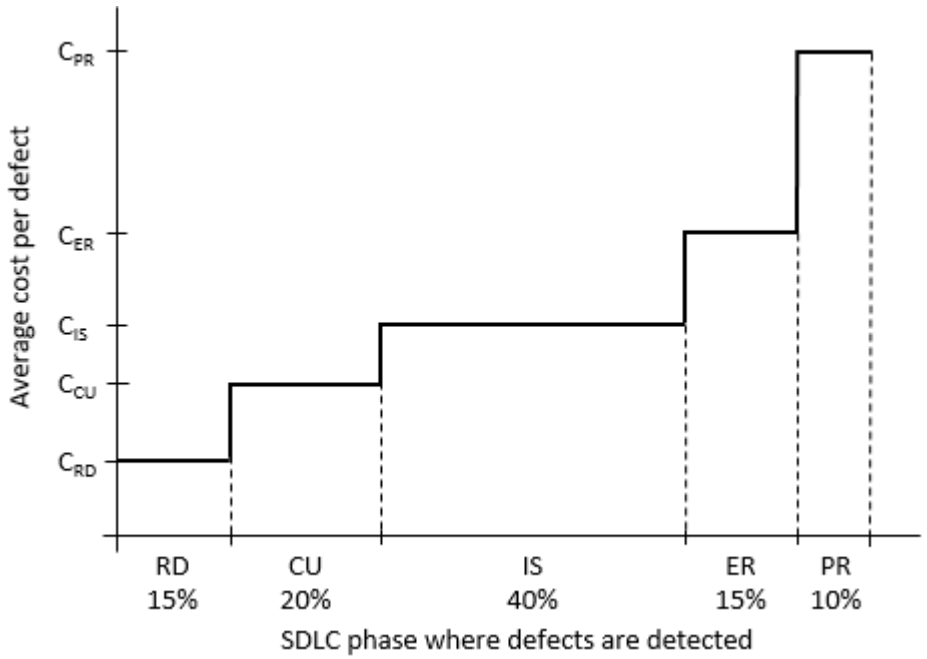


Figure 1.1: Average cost per defect shown by where defects are detected and percentage of defects defected in each SDLC phase [105]

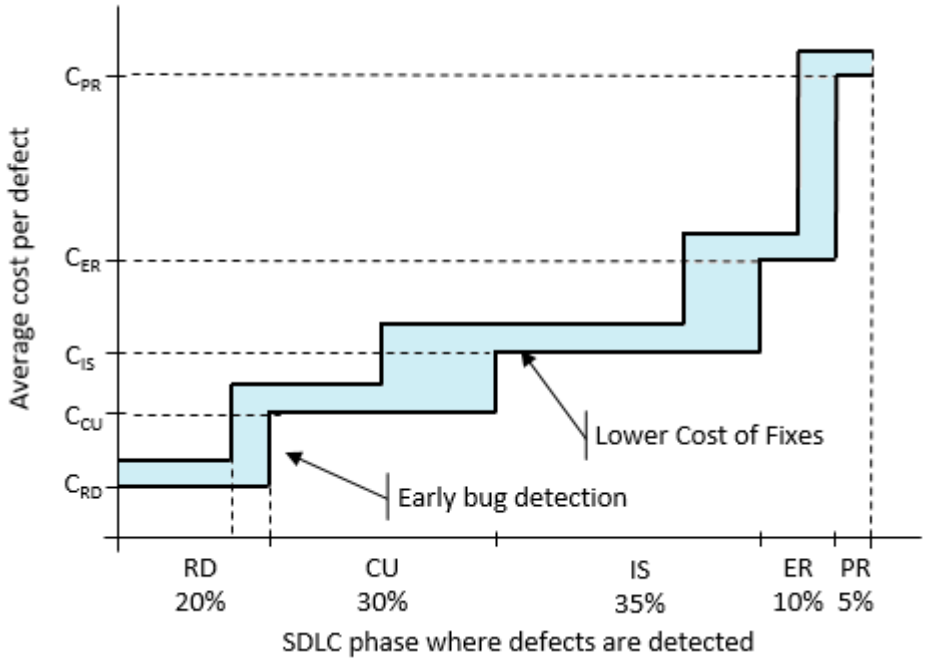


Figure 1.2: Example of cost reductions of detecting bugs and fixing them faster [105].

his intuition the tester is not trying to win the game but to detect as many defects in the tested software as possible.

More formally, Exploratory Testing is defined as simultaneous learning, test design, and test execution. The Exploratory Testing technique requires highly skilled and experienced testers who rely on their intuition and the knowledge of the SUT to detect possible defects [97].

Exploratory testing allows the full power of the testers' mind to help with bugs detection and to verify the functionality of the system under test without any limitations [101]. During the exploratory testing, the tester has to rely on his own experience and intuition, freely exploring the application. However, for the efficiency, the testing process should be planned and testing area predetermined and agreed in a document called test charter (it can be also called a test planning checklist). Also, the exercised tests, found defects and previously explored parts of the SUT have to be documented. These factors influencing an efficiency of the ET process have also been the subject of software engineering research [41].

Typically, the Exploratory Testing approach is being used as a contingency technique when consistent and up-to-date test cases and test basis are missing in the project. Also, in combination with the Error Guessing [46] technique it is being used as an addition to classical test cases based techniques to increase a test coverage.

To perform the ET efficiently, very important part of this technique is documentation of the explored paths in the SUT and the exercised test cases. When the explored path and exercised test cases are not documented properly, this technique loses its potential efficiency. Currently, a large ratio of web applications is usually developed without any or sufficient underlying models. In the most of the cases, this is a consequence of required low development costs and usually short time-to-market in the competitive environment. Nevertheless, the model is still implicitly present in the SUT code and with proper techniques, it is possible to reconstruct the model from the application. To achieve this, for instance, AJAX–enabled applications can be explored by a crawler tool[70] to identify clickable elements. Alternatively, reverse engineering technique can be used to add a cross–platform adaptation feature to the web application [7]. Input elements in HTML page can be analyzed to infer user interface patterns like login form or search field. Input elements in HTML page can be analyzed to infer user interface patterns like login form or search field [91, 34, 73]. For example, crawler–based approach is also used by A^2T^2 [2] tool for Android phone applications.

There is a number of these options and we will come back to discuss them thoroughly in the state–of–the–art review in Chapter 2. Before that, let's discuss benefits and limitations of the Exploratory Testing technique.

1.1.1 Situations Suitable for Exploratory Testing

Exploratory Testing is very well applicable in situations, where test cases cannot be designed in advance. This scenario often arises in the early stage of product development when the application is not stable and prone to changes, or in various situations, in which sufficient design documentation is missing. Nevertheless, there are more situations being a good candidate for ET application. Let's summarize these situations as follows:

- rapid feedback on product quality in a short span of time has to be provided,
- the product is in early development stage when the system is not stable,
- the scope and variations of a discovered defect has to be explored,
- scripted tests are not detecting many errors, and, mainly
- the requirements for the project are vague, documentation is incomplete and it is difficult to determine test cases to be run.

In these situations, several benefits of Exploratory Testing can be exploited.

1.1.2 Benefits of the Exploratory Testing

The Exploratory Testing technique provides following typical beneficial features:

Product Analysis. Testers are free to test the application executing many test cases. They are using their intuition and creativity to break the application. The testers focus more on using the application from the end–user point of view and discovering realistic defects.

Critical Defects Detection. The testers can start testing new features immediately without preparing the test scripts first. Again the testers focus on areas where they sense the defects can be detected.

Quick Product Feedback. The testers detect defects quite quickly and report it immediately back to the product manager or the development team. Thus, found issues can be fixed in a shorter time than those found in a standard software testing lifecycle, for instance [58]. The testing plan or schedule does not have to be altered unlike when scripted tests are executed.

Free form testing. There are no rules in exploratory testing. The testers can be quickly dispatched to test another part of the system under test or even another tested system. Guidelines should be made available to the testers to improve the efficiency of the testing.

Improve the Efficiency of Scripted Tests. In the ET, a number of defects might be left undetected. The technique can, however, improve existing test designs and refine the test scripts by using inputs and data collected.

These advantageous features make the Exploratory Testing suitable option for testing of software projects with missing, obsolete or incomplete design documentation, as well as a complement to standard testing process, in which a test design phase precedes a test execution phase.

1.1.3 Exploratory Testing Challenges

Besides its advantages presented in the previous section, the Exploratory Testing technique has also its drawbacks and challenges that should be addressed to perform this technique in an efficient way. These challenges are mainly:

Resources demand. Exploratory Testing technique requires highly experienced testers who are also familiar with the SUT and its details (to the extent of knowing rarely used features of the SUT). Highly experienced testers are usually expensive team resources, so practical trend is to make the testing costs lower and compromises are demanded by the project investor or manager. Obviously, Exploratory Testing can be performed with a more junior team whose members do not have knowledge of SUT functions and structure, but the efficiency of such testing is questionable.

Documentation. The testers must document explored SUT functions systematically and consistently — either to prevent duplicate tests or for the reproduction and fixing of the discovered defects. If this is not ensured, this technique loses its efficiency rapidly and converges to "free testing" without proper control.

Distribution of work. To prevent duplicities in tests and test data and to allocate resources efficiently, strong Test Lead's presence is required. The testers should be also involved in the planning of the execution of the test cycles.

1.2 Motivation

The motivation of this thesis arises from the challenges of Exploratory Testing performed manually, which have been outlined in section 1.1.3. The state–of–the–art triggers the following questions:

(1) Is there a possibility how to aid the Exploratory Testing by a suitable automated support?

(2) Can we get the best practices of the model–based testing and apply them in an extreme situation, when we have only an implicit model given by the already implemented SUT, without any underlying design model or documentation?

(3) Can we make the Exploratory Testing available also to a more junior testing team, letting a machine to take over part of administrative managerial tasks as a documentation of explored SUT parts or distribution of work to individual testers?

After the extensive literature survey, which we did during this project, we consider these problems to be not sufficiently solved yet. It seems that this area lays a bit outside of the main streams in the research, which is classical Model-Based Testing or research of Exploratory Testing performed manually, but seen rather from a managerial point of view. Our opinion is that a combination of these areas would deserve more attention.

1.3 Goals of the Dissertation Thesis

Based on the motivation, presented in the section 1.2, the goals of this Dissertation Thesis are the following:

1. Explore the possibility how to aid an Exploratory Testing process by a suitable automated support, which will (1) take over part of administrative overhead related to operational management of testing team, (2) will lead to more efficient exploration of SUT functions, (3) enables the testing team to detect more defects in the SUT and (4) will prevent duplication of executed tests by individual testing team members. By operational management in this context, we mean the assignment of particular testing tasks to individual testers and documentation of already explored SUT functions and exercised test data combinations.

2. Design a model of SUT, which will serve as a basis for this automated support. Explore already existing screen–flow based models and either adopt and adjust an existing model or to design an own model, if the situation requires.

3. Use the defined SUT model for a real-time generation of navigational test cases, which will help the exploratory tester to explore SUT in an efficient way. Formalize a mechanism of generation of these test cases. Include also test data suggestions to this process.

4. Implement a framework, which will guide the exploratory testers in the SUT and support them by the navigational test cases in accord with the defined model and principles of generation of these test cases. The framework will consist of three principal parts: (1) browser extension, which will analyzes the SUT front-end pages,

(2) back-end system, which will maintain the SUT model, generate the navigational test cases and present them to the testers in a separate guidance web application, (3) administration part, which will allow to browse, edit and visualize the SUT model, as well as manage the testing team using the framework.

5. Conduct case studies, which will verify the functionality of the implemented framework and compare its efficiency with Exploratory Testing performed manually. Test the applicability of the proposed system on at least three different system under tests. As one of these systems, select a SUT, which HTML front–end structure will by dynamically generated, which could cause the problems with analyzing of the front-end pages. Assess applicability of the proposed framework also for this type of SUT.

In this Dissertation Thesis we limit the scope of the systems under tests to web–based applications and information systems, providing a HTML–based user interface.

1.4 Structure of the Dissertation Thesis

The thesis is organized into five chapters as follows:

1. *Introduction*: Gives an initial description of the problem we are solving in this Dissertation Thesis, the motivation for our work and the goals of this Thesis. Finally, contributions of this Thesis are summarized in this chapter.

2. *Background and State-of-the-Art*: Analyzes the related work in several areas related to the topic of our research and summarizes the results of these literature surveys in the context of this Thesis.

3. *Proposed Solution*: Describes the functionality of the proposed framework and its internal structure. Then, this chapter contains definition of the model of the System Under Test, which serves as a basis for the framework functionality. Further on, the process by which automated navigation of the exploratory tester is generated, is described in this chapter. Finally, this chapter overviews selected implementation details of the framework.

4. *Experiments*: Describes four case studies we conducted to verify functionality and practical applicability of the proposed framework, discusses threats to the validity of performed experiments and gives an overview of alternative applications of the proposed framework.

5. *Conclusions*: Summarizes the results of our research, discusses the future work we are planning in the next two years horizon, and concludes the thesis.

Chapter 2

Background and State-of-the-Art

In this section, the relevant previous research and literature that relates to this Dissertation Thesis are reviewed. Several principal related areas are discussed, as the topic of this thesis is practically a combination of Exploratory Testing, Model Re–engineering and Model–Based Testing in its specific variant.

We start the overview with a general introduction to the **Model–Driven Engineering and Testing** discipline. Then, we discuss the state–of–the–art of the most relevant subpart of this discipline, the **Model–Based Testing**. Briefly, we comment also on **Model Validation and Model Checking** disciplines since they are related to the discussed area. Then, we discuss the state-of-the-art of the **Reverse Engineering** discipline.

Then, we summarize a literature survey in the area, which is conceptually close to the reverse engineering approach: we examine **Capture and Replay** approach used in the automated testing discipline. Further on, in section **Challenges in Dynamic Web Systems Testing** we discuss potential technical difficulties and challenges related to this recording.

Finally, we summarize related research dedicated to **Exploratory Testing** technique, which is the main concept we support by the framework presented in this Dissertation Thesis. As this discipline is confused with the **Error Guessing** technique by the practitioners sometimes, we comment also on this area. Finally, we summarize the presented discussion in the context of the goals of this Thesis.

2.1 Model–Driven Engineering and Testing

Model–driven engineering (MDE) [92, 110, 47] is a software development methodology focusing on creating and using the domain models as a basis for implementing the software system. Using the model has a significant advantage — it allows to specify the system much more exactly compared to a written text only. This can significantly lower probab-

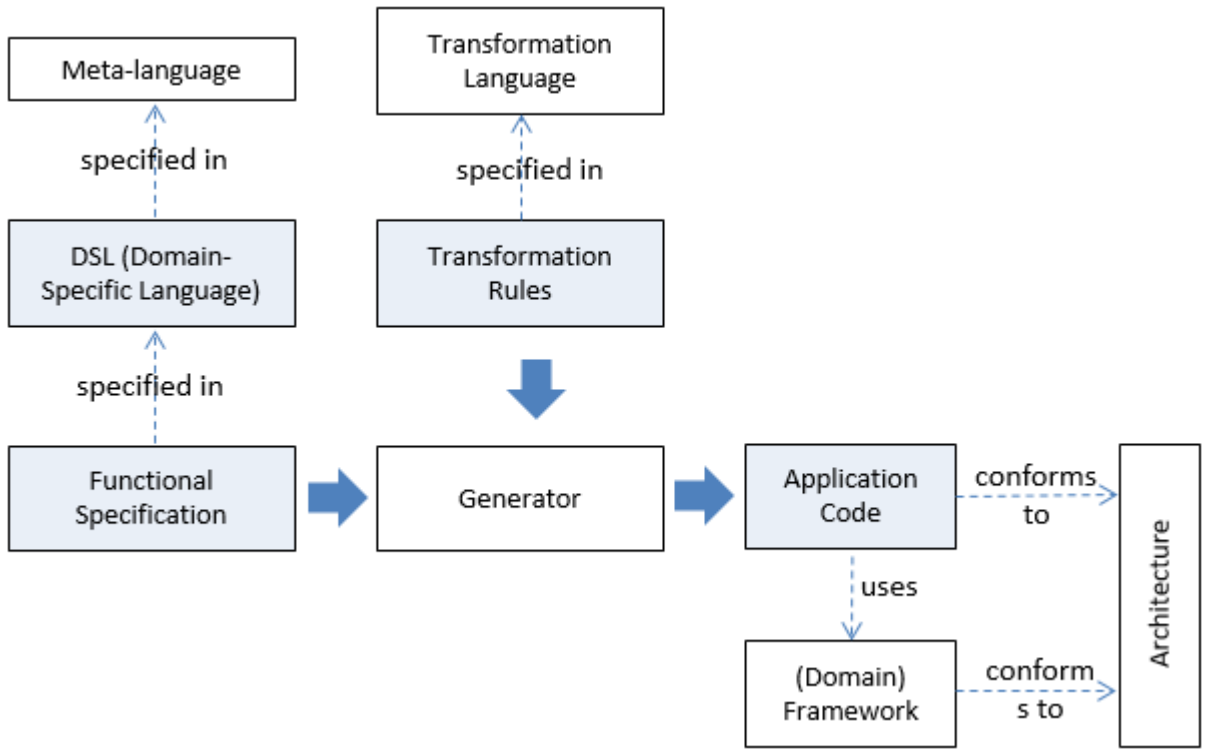


Figure 2.1: Overview of the Model–Driven Engineering approach [28]

ility of software defects, as wrong or ambiguous specifications cause a large ration of the defects. Domain specific models are an abstract representation of the entities and actions representing the application domain. MDE approach increases productivity by maximising the compatibility between systems via the reuse of the models. The models are then transformed into a more concrete code, and in the end, an executable source code of the system is generated, see Figure 2.1.

The principle of MDE is to use of system models and transformations to support all tasks of the software development life cycle (SDLC), spanning from analysis to testing. Modern MDE technologies use various models to represent different perspectives of the system at a different level of abstraction.

With the MDE tools, the productivity rates can increase, and a significant amount of source code can be generated. Though, someone has still to write and debug the transformation rules.

Aligning MDE methods and tool capabilities with the system acquisition strategy can improve system quality, reduce time to deliver the product, and reduce maintenance cost. Otherwise, using the methods and tools can result in increased risk and cost in development (extending and adding new features) and maintenance[1]. The tool–set must be carefully

[1]https://insights.sei.cmu.edu/sei_blog/2015/05/model-driven-engineering-automatic-code-generation-

evaluated before selected — switching to another tool (as well as the evaluation of that tool beforehand) later in development can be expensive.

As mentioned, MDE approach can be applied in various places in SLDC, testing and UI design included. The UI is usually created using domain–specific components and patterns. These components do have typical and expected behaviour across the domains (and even in the niche fields, for example, a medical software or production line visualization software, the standard behaviour is expected). The consistent application of these patterns increases the usability and User Experience (UX). The use of models, in this case, removes the need for the UI design, which is tedious and painstaking task (although the transformations of the model must be adjusted accordingly to achieve the desired level of user experience[80]).

From the perspective of this Dissertation Thesis, we are interested in quality assurance techniques employing this model–driven approach. The quality in MDE can be verified by three principal techniques: Model Validation, Model Checking, and Model–Based Testing, which we are going to discuss in the following sections.

2.2 Model–Based Testing

Black–box testing is defined as the examination of SUT functionality without the knowledge of its internal mechanisms. In this context, Model–Based Testing (MBT) can be discussed as the automation of black–box testing approach verifying the compliance of the implementation of a system with the specification — the model.

Abstract model representation of the SUT is provided as an input to a test case generator. The generator creates abstract or implementation specific test case scenarios. The concrete scenarios are then executed against the SUT. The key point here is the automation of the process, which allows generating an extensive number of test scenarios with a high level of test coverage.

In the model-based testing [107, 65, 53, 31, 95] the standard approach is the test case generation from the formal model of the system requirements and behaviour. The model is usually created manually from information specifications or requirements. Creating a formal model from the requirements also results in an immediate feedback. The modelers can reciprocally clarify the missing information in the specification. The model describing the system is usually an abstract, partial presentation of the desired behaviour of the system. Test cases generated from such a model are functional tests on the same level of abstraction as the model forming an abstract test suite. An abstract test suite cannot be executed directly against an SUT because, usually, there are physical details of the test missing. By employing a model transformation, an executable test suite is derived from a corresponding abstract test suite [82]. In some model–based testing environments, the models do contain enough information to generate executable test suites directly. A

and-beyond.html

mapping has to exist between the elements of the abstract test suite and executable code in the software to create a particular test suite. As test suites are derived from models and not from the SUT source code, Model–Based testing is usually seen as one form of black–box testing.

The MBT process usually has four stages [106]:

1. Building an abstract model of the system under test
2. Validating the model
3. Generating abstract tests from the model
4. Refining those abstract tests into concrete executable tests

The abstract–to–executable test suite transformation is usually applied mechanically. The common approach is to consolidate all test derivation related parameters into a separate package often known as "test requirements". This package can contain additional meta–data that are not captured by the model. The meta–data can define the risk level of selected application modules; expected relationship between the observed and controlled environment variables once the system is put in place, assumptions about the environment due to conditions external to the system, or the relations that the system must maintain with the environment [26], or the conditions indicating when the testing is finished (test exit criteria).

A number of concepts and approaches can be found in this area, as well as various types of underlying models, which are used for this process. Let's give some examples. A model transformation framework is presented in [40]. In this framework, forward engineering stream goes from computation independent model (CIM) to application code and the testing stream goes from computation independent test (CIT) specification to an executable test script. The authors are describing vertical transformation for composing the two streams and horizontal mapping for reflecting changes made in the modelling framework. The chain of transformations produce tests, meta models represent test cases for web applications at different levels of abstraction. The important part is the focus on automatic alignment of the platform independent test (PIT) specification after changes were made to PIM — the alignment is keeping the models synchronized. In this concept, different modelling languages are used on different levels: Business Process Model and Notation (BPMN) for Computation Independent Model (CIM) and WebML for Platform Independent Model (PIM). The WebML model enriches the BPMN process scheme with operational details.

As introduced above, the abstract test cases are often generated from the model, and these abstract tests cannot be executed against the SUT. The abstract tests can be mapped to concrete (physical) tests, but without a dedicated support, the data flows cannot be mapped to these tests. A test generation methodology — Abstract to Concrete Tests — was proposed in [16] which is using state chart diagrams and symbolic execution to resolve

the data flow between the components. In this concept, the abstract test can be then effectively mapped to a concrete test and executed.

Testers transform the abstract tests into concrete tests with (1) particularly selected test input values (test data) from the abstract determination of the data ranges in the abstract tests (if present there), (2) test oracles that determine the results to be asserted as a result of the executed tests and (3) physical details about the particular test steps and verify the correctness of test case execution. The problem which arises when we need to determine the correct behavior of the SUT and distinguish it from potentially incorrect behavior is called the "test oracle problem" [6]. Automation of the test oracle is essential to prevent manual determination of the SUT correct behaviour, which can be resource demanding task for large sets of test scenarios [62, 64]. A comprehensive survey on the testing using test oracles, a process called metamorphic testing, is provided in [93]. Even when a test oracle is automated, the problem is not solved completely, because the test oracle itself can contain defects, and, thus, provide misleading information about expected results of the tests.

A primary concern in the model–driven engineering is how to ensure the quality of the model–transformation mechanisms. One of the validation methods that is commonly used is model transformation testing addressing the efficient generation, selection the validity of the transformed models [56].

There are many general purpose modelling languages available for describing the system high–level model used in the industry. The most widespread and used is Unified Modeling Language (UML[2]).

Structural and behaviour diagrams of UML can capture and describe various aspects of the application. With the addition of constraints using Object Constraint Language (OCL), the UML model can be made more descriptive. OCL complements the UML model with the constraints that it cannot capture - for example, restrictions on a class model attribute (Age of a person cannot be a negative number).

Wide usage of UML as design modelling language also implies its use as a source for the Model–Based Testing techniques. Shirole at al. [96] conducted a survey to improve the understanding of the UML based testing techniques. Authors of this study focused on the usage of the behavioural specification diagrams. Related work can be tracked on areas of sequence diagrams [5, 100, 59], state chart diagrams [16], activity diagrams [55, 59, 52], or state machine diagrams [111] diagrams. The research approaches were classified by the formal specifications, graph theory, heuristics of the testing process, and direct UML specification processing. UML collaboration diagrams provide a complete path for a use case or the realisation of an operation. That makes them suitable for MBT purposes because they describe the connections between the functions provided by the software in a form that can be manipulated using an automated approach. Class diagrams and use–case

[2]http://www.uml.org/

diagrams are not precise enough for model based testing; hence additional description from dynamic behavioural models is required [107]. UML diagrams [33] have been also widely used as a source for code generation [60, 1, 67]. Use–case models [63] or specification of integration interfaces [85] can be used in this process as well.

Usually, the UML diagrams are transformed into more suitable application model, from which the test case scenarios are generated. Nevertheless, an accurate, consistent and detailed model is often very costly to create, and the maintenance and synchronisation of such model with the rapid development of the project are difficult. The more dynamic or chaotic the environment of the web applications development is, the more expensive is the necessity to update the model to reflect all updates and changes.

Unfortunately, the standard set of UML 2.0 diagrams does not directly focus on the user front–end interaction. UML notation was also used to model the user interface, but it does not have the capabilities to model all the nuances of the UI. Even though a user interface diagram was introduced with a user interface specialisation in [38], there are more suitable modelling languages available to model the user interface of an application, especially in the domain of the web applications.

The first example of these web modelling languages [90] is the Web Modeling Language (WebML [74, 12], see an example in Figure 2.2), which was created introducing visual notations and a methodology for designing complex data–intensive Web applications. Later on, this language then evolved into IFML [11, 13, 109] to cover a wider spectrum of the front–end interfaces and the data flow between the application front–end components. IFML was standardised in 2013. Its notation is easily extensible — new containers, components, events can be added or custom UML stereotypes applied as described in [14], where new components and events (swipe, camera event, location sensor event for mobile UI modelling) were added to define mobile–specific interfaces and interaction. It represents a prospective modelling tool to describe application front–end and a flexible and easily extensible notation.

Data driven application front–end is often built using reusable components (forms, list views, detail views, etc.). Components have expected behaviour — forms on the page are expected to be filled with data for further processing, lists show record details, etc. These operations can be modeled using the IFML notation (see an example in Figure 2.3). IFML capabilities are promising to generate front–end test case scenarios, as we have explored in one stage of this Dissertation Thesis project [A.9].

In Model–Based Testing, correctness and completeness of used models has the direct impact on the effectiveness of the generated test cases. Faults, which can remain undetected by these generated test cases can be detected with Exploratory Testing performed manually by experienced test engineers [23]. The combination of these concepts can also be traced in the literature — the tool–set proposed in [23] analyses the recordings of performed tests with the aim to identify inconsistencies on in system models. These identified

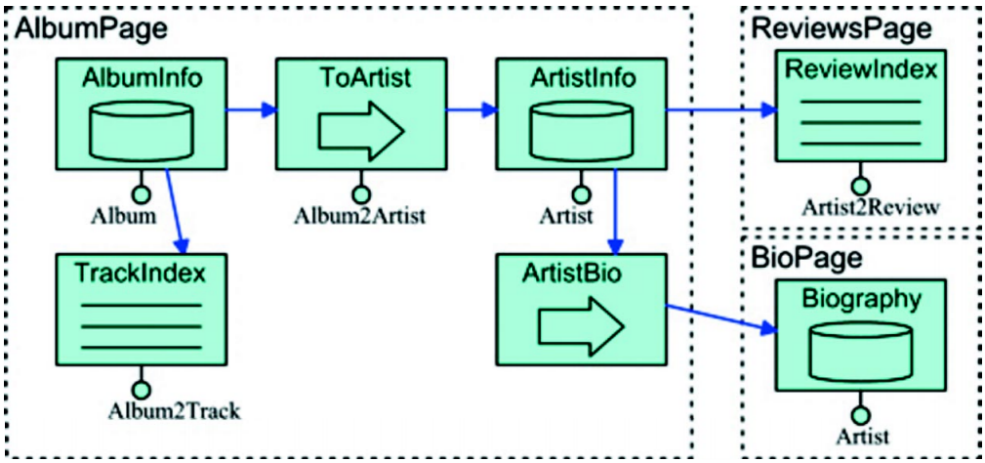


Figure 2.2: Example of WebML — a composition and navigation specification [22]

inconsistencies are used for the refinement of models to be able to generate more efficient test cases. The approach was applied in the context of an industrial case study to improve the models for model–based testing of a Digital TV system. After applying this approach, three critical faults were detected. These defects were not detected by the initial set of test cases, and they were also missed during the exploratory testing activities.

The MBT represents very prospective concept to create efficient test cases in an automated way. However, it is still limited be needed presence of the consistent and up–to–date model of the system under test. Here, the question arises: Are we able to use this powerful concept, when the model is partial only, or when this model is generated by reverse engineering technique? Let's explore more. After having a quick glance at a related area, Model Validation and Model Checking in Section 2.3, we will discuss the possibilities of reverse engineering in Section 2.4.

2.3 Model Validation and Model Checking

As related concepts to Model–Based Testing, we briefly outline also Model Validation and Model Checking areas. The importance of a having valid and consistent model before generation of the source code or test scenarios is essential. The model should be validated (its consistency checked) with respect to the defined criteria — the semantic and syntactic criteria (to verify if the model was created correctly and conforms the rules of the modelling language) or constraints defined by the author of the model (to verify if the model complies the rules and constraints of the domain). For example, when the model is represented using UML notation the constraints can be created using OCL that supplements UML with a code–like constraint expressions.

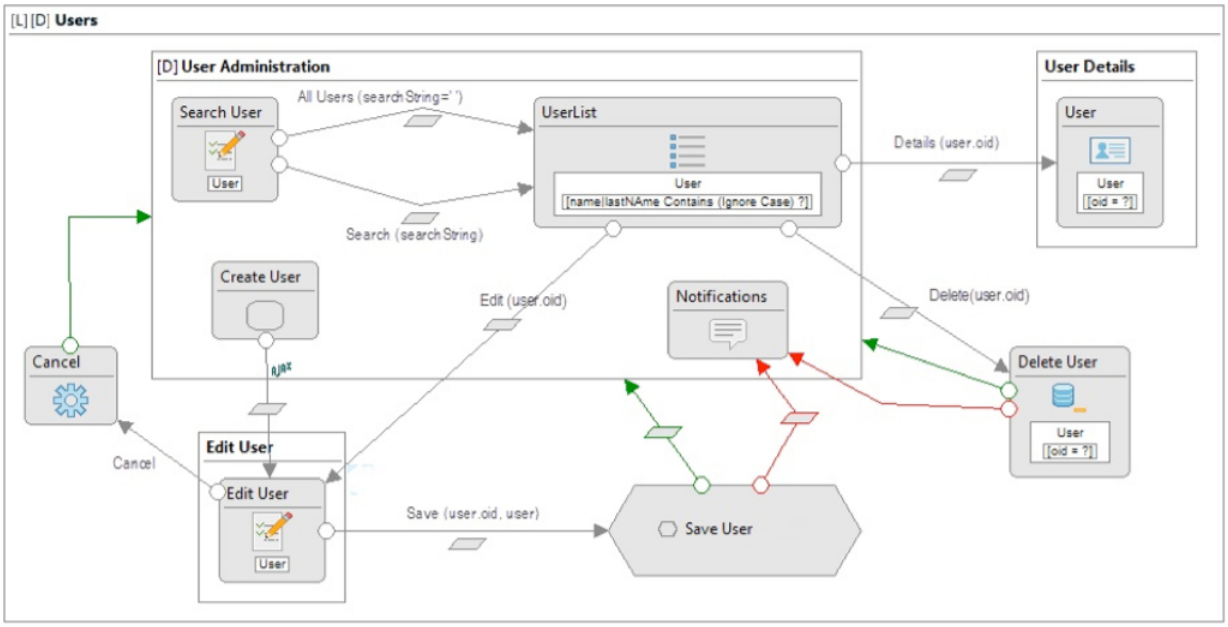


Figure 2.3: Example of an IFML model — an administrator site [27]

Model is validated with a special tool which often requires switching to a different perspective and paradigm (for example theorem solvers, Petri nets or solvers for Constraint Satisfaction Problems) and to learn how to use another tool–set. A more natural for persons authoring the model is to use the modelling language itself as described in [42], where UML and OCL are used. A valid model is an input of the model transformation (or a chain of model transformations) that should also be correct and valid and should be a subject to testing as well.

The system requirements are described in the specification. In model checking, a system is considered correct when all specification requirements are met [36]. Therefore the model must be precisely defined. The process of formalization of the SUT description can help in discovering inconsistencies in the model or reciprocally inconsistencies in the specification. The correctness of the model can be determined even before a single line of code is written preventing complications later and complicated modifications later in the software development stages.

The developers cannot wait for the analysts to complete and finalise the model or until the whole model has been completely checked, they often start writing lines of code before the model is finished. Even this incomplete model must be checked which is a challenging task. For instance, the authors of [4] dealt with this problem using probabilistic logic and employing a three–valued logic.

For the model checking process, temporal logic is also used as one of options to conduct this process [10, 66, 15]. Temporal logic is a decidable logic to reasoning about the

behaviour of the system in time. Temporal logic have temporal logic operators to quantify over the state of the system over time — for example *Always*, *Never*, *Next*. The properties of the system are described with the temporal logic operators, and the checker uses this description to reason over the states of the system to determine whether the specification is true or not.

The topic of model checking is beyond the scope of this thesis, more on this topic and the topic of model checking tools with the introduction to transition systems as a formal model of systems and temporal logic as a formal language for behavioural properties can be found in [8]. A detailed survey on six selected model checkers was conducted by authors of [39] to check a library system to identify features of the model checker that are required to validate the information systems.

2.4 Reverse Engineering

In the case when the SUT model is not available, Reverse Engineering Technique (RET) is a suitable approach to recreate it from the actual state of the SUT. The model of the application can be recreated using RET by analysing the static content of the HTML pages – namely the HTML elements – and to build a directed graph with web pages as nodes and transitions/links as edges. Not every web applications have static pages only; some content is dynamic, and the content has to be treated differently in this case. Analyzing of the content requires the execution of the code, and it can depend on the value of input variable(s). Prospectively, not only all the possible flows of screens and actions in the SUT but also flows representing business processes could be re–engineered. In this process, proper manual input to mark which sequence parts belong to the particular workflow is needed. Authors of [76] have created a tool called iMPAcT [75, 64] for mobile pattern testing. They are focusing on an automated testing of recurring behaviour, i.e. UI patterns. For a UI pattern to be matched, the current state of the application is analysed when an event is fired. The pattern preconditions must be verified, and all checks met. Each UI pattern has a test pattern associated with testing. A catalogue of UI patterns was created for the mobile application, which can be prospectively also used in reverse engineering of the SUT model.

With the evolution of development techniques, many legacy applications are left behind. Adding new feature to such application or fixing a bug might be difficult — for example when the application was developed using a programming language from the past. A systematic model driven reverse engineering process to generate an IFML representation from such applications is presented in [89]. The authors leverage IFML to modernise the front–ends of the framework–based legacy web applications created before the adoption of Model Driven Web Engineering.

The Exploratory Testing technique does not require precisely specified and documented

test cases. In [48, 50], it was even demonstrated that detailed test scenarios do not increase overall testing efficiency from the economic point of view. The reason is that many resources are spent on creating such detailed scenarios. For projects with low and medium demands on SUT reliability, this is worth considering. Also, lots of applications are usually developed without any or sufficient underlying models.

Another study proposed using ET approach to acquire knowledge for a model-based testing [32]. The main target area of Exploratory Testing is GUI testing where the tester tries to find defects and break the application by exploring the possible actions in the SUT using his intuition. Testers can start testing new feature immediately since the planning of the testing process is not necessary. In contrary to these advantages, certain disadvantages shall also be mentioned [94]. It is difficult to assess whether a feature was tested, the process is not monitored and tracked, the quality of testing is unclear and depends on the experience and skills of the tester. The problem is also to re-evaluate the test later.

Kim et al. [57] focused on generating formal specifications from manually written test cases – a possible synergy between Exploratory Testing and model-based testing. The SUT model is re–engineered to get a better insight into the legacy code. Reverse engineering provides an insight into the application without having its documentation available. Test scripts written for Sikuli [3] tool are transformed into a model with preconditions and post–conditions, screen, action, and input element identifiers. Sikuli is a tool for desktop automation that uses Optical Character Recognition (OCR) to identify elements visible on the screen. The knowledge of creating Sikuli scripts and the requirement to have some tests created in advance is required.

A tool called MobiGUITAR [3] focuses on automated model-based testing of mobile applications (a similar tool called GUITAR [79] was created for automated testing of GUI–driven desktop software written in Java). In the first step (called ripping), a state machine model of the SUT is created — the application is launched with a given start state, then a list of the events that can be triggered in that state is created. The tool then triggers the events from the list and analyses the new states the application is traversed in a breadth-first order. The crawler proposed by Mesbah et al. [70] works similarly – clickable elements on an AJAX–enabled web page are detected. Then the click action is triggered and when the response is returned from the server, and the document object model of the page is settled, the resulting state of the application is analyzed, and the system state machine model is created.

Open2Test test scripts are automatically generated from software design documents in [103]. A design document, design model and test model is extended with the information about the structure of the web page (identificators of the HTML elements) in the integration testing tool called TesMa [104]. When software specification changes, the latest scripts are regenerated from latest test design documents. This reduces the cost of the

[3] http://www.sikuli.org/

maintenance of the test scripts. However, the design documents must be extended with the detailed information before the test generation process can even start.

The structure of the web application is analysed to create a page flow graph (PFG) representing the flow of the pages in [84]. The graph captures the relationship between the web pages of the application, and the test cases are then generated by traversing this graph (all sequences of web pages). The PFG is then converted into a syntax model consisting of rules, and test cases are generated from this model. Unfortunately, the construction of the PFG is not described in the cited paper and an extra step is required to convert the PFG to syntax model to generate the tests later. Common elements, user interface (UI) patterns, are used when developers are creating the UI of the applications — examples of such pattern are Login, Find, Search. The users with a certain degree of experience know how to use the UI or how it should be used. As described in [77] it is possible to define and generic test strategies to test these patterns. Pattern based model testing uses domain specific language PARADIGM to build the GUI tests models based on UI patterns and PARADIGM based tools to build the test models, generate and execute the tests. To avoid a large number of test cases, they are filtered based on particular configuration or selected randomly.

Another solution, GUISurfer [98] automatically reverse engineers a behavioural model of the GUI from the source code of Java Swing-based GUI applications to produce a Haskell specification. This model of the application is then validated by running test cases. An alternative, REGUI2FM [81] reconstructs a GUI into a Spec# model, which can be used in Spec Explorer environment [108] to generate test cases. It also captures user actions into scenarios which are used for testing [78].

In relation to the Reverse Engineering of the SUT model, also the Capture and Replay concept can be discussed, which we are going to do in the following section.

2.5 Capture and Replay

The Capture and Replay (CR) technique is used for automated testing of the user interface and functionality of software applications. As the name suggests, the Capture and Replay tools operate in two modes — capture and replay. When using the tool in capture mode, testers run an application and record the interactions between the user and the application. Recorded interactions with the SUT can be persisted and repeatedly replayed later without human intervention in the replay mode. CR tools often support regression testing.

In the framework proposed in this Thesis, we are employing this technique to capture the interaction of the user with the SUT, which serves as a basis for reverse engineering of the SUT model.

The Capture and Replay scenarios are quite easy to create, and no unique skills of testers are required. But the testers have to manually the widgets and some of their

properties they are interested in during a capture session. This is also the cause of the problem with the Capture and Replay scenarios — the recorded tests tend to be very fragile because of the nature of the application development and evolution — changes to UI can break the test cases [61]. The first tools that were recording the mouse movements and the mouse event coordinates were very brittle since the smallest change, like changing the location of an element by few pixels, resulting in failing test. This problem was solved by capturing the UI elements and not the mouse coordinates.

Also, the sequence of captured interaction is not a complete test case unless the framework supports verifications. These verifications have to be added manually. For example, Selenium IDE[4], an integrated development environment for Selenium scripts implemented as a Firefox extension, is using the commands like `assertElementPresent`, `assertText`, `assertTextNotPresent`, ... to assert certain conditions.

The fragility of Capture and Replay scenarios is cited as a frequent problem; these tests usually fail as the required elements cannot be located because the structure of the page or their naming has changed. In a study [44] 153 versions of 8 different web applications were analysed, an observation was made that UI element locators caused over 73% of the test breakages, and attribute–based locators caused the majority of these. Failures are also caused usability improvements — page reloads or the lack of them, JavaScript pop–up boxes, using AJAX for server calls etc. In surveys conducted on this topic, for instance [17, 87], maintenance of front–end based automated tests has been reported as the main problem of this technology. Record and Replay approach displays more severe maintenance issues than its alternative, the Descriptive Programming, in which a program code creates the test steps and various reusable objects and design patterns can be used to optimise the tests to be more stable [18]. The results of another study support the idea of a vast number of test cases becoming unusable for modified GUI versions. The repairing algorithms, with four simple transformations, can repair more than half of the test cases, short test cases are less likely to become unusable [68]. Changes to parts of the system that dominate the result in a significant number of test cases make these tests unusable and failing.

Tools using capture and replay technique are targeting various testing (Windows, Linux, OS X) and tested platforms (mobile, desktop, web), programming languages and technologies. For instance, the user events of Android applications are captured and converted into Robotium[5] test scripts that can be executed to replay the recorded actions of users in [64]. The approach also allows inserting assertions when capturing user interactions for verifying the outputs of Android UI components for desktop applications [37].

Authors of [88] have conducted an experiment to evaluate selected Capture and Replay tools. The study also indicates that for simple testing tasks the effort of using a CR–based tool is lower than using an MBT tool, but with the increasing testing complexity, the

[4]http://www.seleniumhq.org/projects/ide/

[5]Robotium is an Android test automation framework, https://github.com/RobotiumTech/robotium

advantage of using MBT grows significantly.

Also, defect reporting can be aided and automated by the Capture and Replay approach: A tool called Reanimator[6] was originally designed to record web application crashes for later debugging — the recorded sequences of user actions were replayed to reproduce the problem. The usage of the tool is not limited to recording only; it can be used to create tutorials or automated browser tasks. Reanimator was inspired by Mugshot [72], a system that is capturing every event in a JavaScript program, which is allowing developers to replay those events in a deterministic way. The tool can replay every step the user has taken that led to a failure. This has the advantage over the commonly used error–based reporting systems since the stack trace gives insufficient information about the crash (it only provides a snapshot of the system after the failure).

Inspiration acquired from this area has served us during the design of the parts of the proposed framework, which is recording the testers' activity in the SUT. This area brings many technical challenges, which we explain in the following section.

2.6 Challenges in Dynamic Web Systems Testing

The current Web has practically evolved from a platform for publishing of static material to the major medium for learning, business, entertainment and many other areas of peoples' activity today. Users are no longer only passive consumers of Web content but also creators and providers of the content. New software development paradigms such as service–oriented and cloud computing are based on Web technologies.

The specifics of modern web applications differentiate them from any other software application and these details significantly affect the testing of such applications. In [30] the main differences between Web applications and traditional ones are discussed, including their impact on the testing (white–box, black–box and grey–box testing strategies). The advance of Web 2.0 applications brought new specifics the tests should deal with, and the question is what is coming in Web 3.0. Contemporary development styles of user interface construction are not HTML–based only, the user experience is dynamically enhanced using JavaScript making identification the HTML elements more demanding task. The dynamic nature of the UI makes it harder to identify the UI elements correctly. Crawlers, tools systematically visiting all pages of the web application, are quite often used to quickly collect the information about the structure, content and navigation relationships between pages/actions of the web application [102, 25, 19]. While the crawlers can go through the whole application very quickly, a sequence of steps by such crawl may differ from the sequence made by a manual tester. Also, some parts of the application could be not reachable by the crawler; also crawlers could meet difficulties when user authorisation is required in the SUT.

[6] https://github.com/WaterfallEngineering/reanimator/

Unlike desktop applications, web browsers are adding another level of interaction to the tested application. The tester is not limited to use only the SUT capabilities, but can also interact directly with the browser — reload the page, navigate back and forward using designated buttons, disable JavaScript, delete cookies, use auto–complete, close the browser window or tab, use browser extensions and others. These browser features are adding some un–predictability to how the user interacts with the application. Some older web applications even use frame–sets to organise multiple frames and even nest the frame–sets to make the layout of web pages identical and simplify the development. Each frame in a frame–set displays different web page. The interaction with the page then breaks into interaction with multiple pages which must be taken into consideration [99].

2.7 Exploratory Testing

Exploratory Testing technique itself is a subject of current software testing research. As explained in the introduction, this technique gives software testers certain level of freedom to design and execute the tests while exploring the product [45, 49, 83, 51]. The tester uses the data gathered from the execution of the first set of tests to conduct the next round of tests. ET can find critical defects in a shorter time. Unlike documented test cases, exploratory testing does not follow any testing rules. Testers with strong knowledge of the business and technical domain explore the application. By browsing through and using the application like a real user, testers are more likely to find issues that customers might face.

An industrial case study to evaluate the impact of education level and experience level on the effectiveness of ET was conducted by Gebizli et al. [24]. In this study, 19 practitioners, who have different education and expertise levels, were involved in applying ET for testing a Digital TV system. The results show that efficiency regarding the number detected failures per unit of time is significantly affected by both the educational background and experience. Experience also has a significant impact on the number of detected critical failures, whereas education has not. Though the formal education, formal training or test certification is not required and regular end–users regularly find and report bugs in systems even though they are not trained as testing professionals, the testing process encompasses a broad range of skills (planning, design, automation, exploratory testing). A certain level of formal proficiency in these areas is required [71].

Controlled experiments with 70 participants were conducted by authors of [9] to quantify the effectiveness and efficiency of exploratory testing. The report showed that ET (compared to testing with documented test cases) found a significantly greater number of defects and also found significantly more defects of varying levels of difficulty, types and severity levels (though the difference in numbers of reported false defects was not significant).

Another research was conducted to improve the efficiency of the ET process. A method

called team exploratory testing (TET) [86] improves the results of this technique, employing the team work more intensively. The motivation behind this idea is that higher number of people in quality assurance team increases the number of the found defects and that anyone, not only the testers, can report the defects. The testing can be performed for example by domain experts and thanks to the nature of ET; the testing session does not require any preparation or post–work from the participants. TET has similarities to software reviews and usability inspections. The TET sessions were found to be more efficient than other testing methods on average and were more likely to detect usability and UI–related defects while other testing vehicles detect more functionality related defects. The results of the research are not generalised for wider use but might be applicable to similar projects.

In our approach, we also employ the idea of the teamwork during the ET process. Differently to the TET approach, this teamwork support is not based on team sessions, but on the recordings of testers' activities in the SUT and subsequent optimisation of navigation provided to the tester.

2.8 Error Guessing

Error Guessing is usually interpreted as a testing technique separate from Exploratory Testing, for instance, [58, 46]. From a conceptual point of view, it can be discussed, if this technique can also be interpreted as a part of the ET process. For this reason, we included this technique to our state–of–the–art survey.

Software programs often produce incorrect behaviour when special cases data is provided as an input. This situation can arise when the developer forgets to handle this particular case or the situation was handled on one level of code but left unhandled elsewhere. For example, when the code tries to divide a value by zero or a purchase order is issued with the issue date in past.

Special cases will often depend on the data and the function of that particular part of the application. These special cases have to be identified by testers' intuition and experience. Consequently, determining special cases is also called error guessing [69]. The experienced testers are encouraged to think of situations in which the software may not be able to process correctly, which would result in an unexpected error. Experienced testers or users with long experience with a particular system usually have much higher ability to predict possible defects in such a system.

For these reasons, Error Guessing technique can be used as an effective complement to more formal techniques. Usually, the Error Guessing is performed after execution of test cases prepared by the formal techniques[7]. The objective of this technique is to focus on areas not covered by the other formal testing techniques.

[7] http://istqbexamcertification.com/what-is-error-guessing-in-software-testing/

Skill and experience of the tester are the key factors to the success of error guessing technique. Testing using formal techniques usually has strict rules for how to test the system. Here, the tester is using the system and the more he knows how it works, the more chances he has to detect the defect by addressing the components of the system where the system may fail.

The tester will try to provide input values that will very likely break the application — for example, when he notices a field whose value is a divisor in an operation, he will try to include zero value, blank input or no input at all; empty files and the wrong kind of data — alphabetic characters where numeric are required, invalid date values and other.

Similarly to the Exploratory Testing, also in Error Guessing requires a certain structured approach to be performed efficiently. This can be achieved by creating a list of possible system defects or malfunctions, followed by a set of tests aimed at their detection. Such a list can be based upon the testers' experience with the system under test. Also, consultations with other specialists having this knowledge can lead to the identification of such possible defects. Another possibility is common knowledge about why software fails[8].

2.9 Summary of the State of the Art

During thorough literature survey, we have not found a concept which would address the intended use case of the framework which we present in this Dissertation Thesis. In the individual areas as Model–Based Testing, Model Reengineering or Exploratory Testing, a number of related works are available; nevertheless, much less work is present in cross–over of these areas.

As an example of the combination of Model-Based Testing and Exploratory Testing, solution presented in [23] can be given. The authors analyse the recordings of performed tests with the aim to identify inconsistencies on in system models. These identified inconsistencies are later used for the refinement of models to be able to generate more efficient test cases. However, goals of this Dissertation Thesis differ, as we reconstruct the SUT model to generate the navigational support for the exploratory tester.

The majority of the related work is assuming the existence of the SUT model preceding the test case generation process. In this point, our approach differs, as we are going to create the SUT model dynamically and incrementally during the exploratory testing process. Also, the navigational test cases, which will be produced by the proposed framework are dynamic, as the inputs are not only the current state of the constructed model but also SUT parts explored by the particular testers or all testers in the exploratory testing team.

In our approach, we will use web application model which was created by adoption, modification and extension of the model by Deutsch et al. [29]. Nevertheless, during our work, the model underwent significant changes. From the related work, the model is

[8] http://istqbexamcertification.com/what-is-error-guessing-in-software-testing/

conceptually most close to the IFML standard. In our previous work, we were exploring the possibility to use IFML as an underlying model [A.9]. Nevertheless, specifics of the presented case led us to keep the model as defined in this Dissertation Thesis.

In this Thesis, we also explore the potential of organized teamwork during the ET process, which is an area covered by previous study [86]. Nevertheless, we approach this topic from a completely different viewpoint, our teamwork support is not based on team sessions as proposed in [86], but on the recordings of testers' activities in the SUT and subsequent optimisation of navigation provided to the tester.

Chapter 3

Proposed Solution

This section summarizes the functionality of the Test Analysis SUT Process Information Reengineering (**Tapir**) Framework, gives details about the underlying model of the SUT, on which the framework processes are based, and presents a mechanism, by which the automated guidance of the exploratory tester through the SUT is conducted. This mechanism includes generation of navigational test cases, which are presented by the framework to the exploratory tester and preparation of testing data to these test cases.

3.1 Principle of the Tapir Framework

The aim of the Tapir Framework is to make the Exploratory Testing of web-based system under tests more efficient by automation of activities related to

1. recording of the test actions performed by the exploratory testers in the SUT,

2. taking decisions, which parts of the SUT shall be explored in the next test steps, and

3. organization of the work for a group of exploratory testers.

The Tapir Framework tracks tester's activity in the browser and incrementally builds the model of the SUT based on its user interface. In this process, a web-based SUT with HTML–based user interface is assumed. Based on this model, which can be further extended by tester's inputs, navigational test cases (more details in Section 3.4) are generated. Together with this, explored paths in SUT are recorded for the individual exploratory testers. The navigational test case helps the testers to explore the SUT more efficiently and in a systematic way - especially, when considering a teamwork of more extensive testing group (typically 5 and more testers).

Technically, the Tapir Framework consists of three principal parts:

- **Tapir Browser Extension**. This extension tracks tester's activity in the SUT and sends the required information to the TapirHQ component. It also highlights the GUI elements of the SUT in a selected mode (elements already analyzed by Tapir Framework or elements suggested to explore in the next tester's step). The extension also analyzes the SUT pages during the SUT model build process. Currently, implementation for Chrome browser is available.
- **TapirHQ** implemented as a standalone web application, which guides the tester through the SUT, provides navigational test cases and allows Test Lead to prioritize the pages and links, enter suggested equivalence classes for the SUT inputs and related functionality. This part constructs and maintains the SUT model. The TapirHQ runs in separate browser window or tab, like a test management tool displaying the test cases for the SUT.
- **Tapir Analytics**, which allows to visualize the current state of SUT model and the particular state of SUT exploration. This part is also implemented as a module of TapirHQ, sharing the SUT model with the TapirHQ application.

The overall schema of the framework is depicted in 3.1. The Tapir Framework defines two principal user roles:

1. **Test Lead** - senior team member, who explores the SUT first before letting the testers perform detailed tests. Besides the Tester's functionalities (see further on), the Test Lead has the following principal functionalities available:
   a) Prioritization of the pages, links and user interface action elements of the SUT. During the first exploration, the Test Lead can determine a priority of the particular screens and related elements. This priority is saved to constructed SUT model and is used later on in the navigational strategies (see Section 3.4.2).
   b) Definition of suitable input test data. During the first exploration, the Test Lead can define Equivalence Classes (ECs) for individual input fields detected on the particular page. The ECs are saved to the SUT model and used later on in the process when generating navigational test cases. Also, after the definition of ECs for all inputs of the form on the particular page, the Test Lead can let the Tapir Framework generate the test data combinations using external Constrained Interaction Testing (CIT) module.
2. **Tester** - team member, who is being guided by the Tapir Framework to explore the previously unexplored parts of the SUT. For the particular tester, the test lead selects particular:
   a) Navigational strategy, determining the suggested path in the SUT suggested to the tester in the generated navigational test cases, and

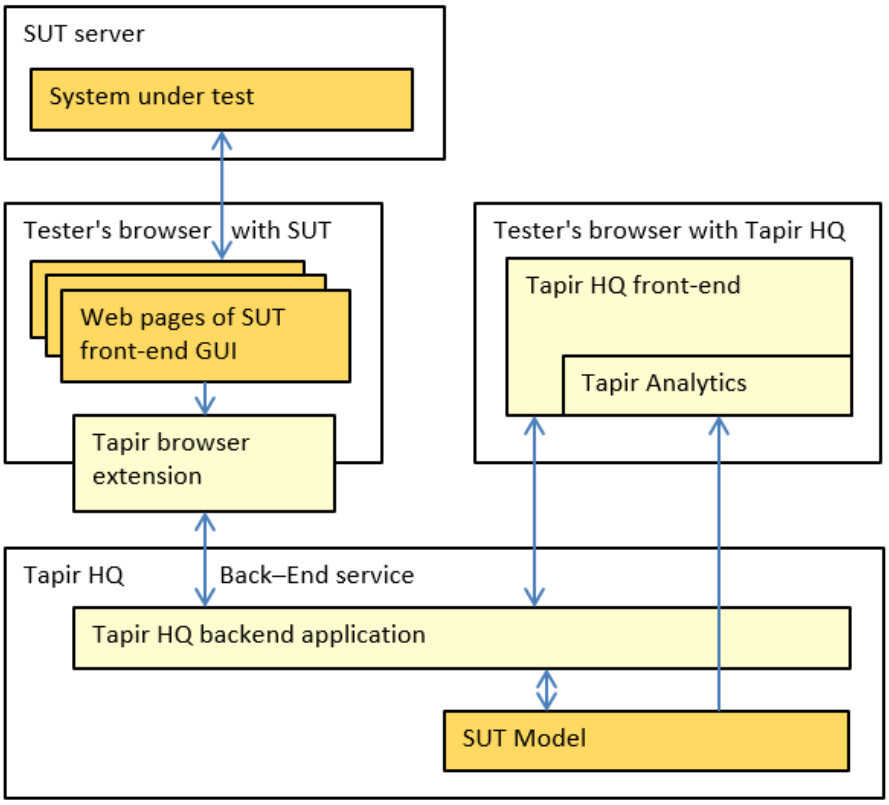


Figure 3.1: Overall schema of the Tapir Framework

b) test data strategy, determining, which test data will be suggested by the framework to be filled into the SUT forms and similar inputs.

The Test Lead can change navigational and test data strategy of particular testers dynamically during the testing, to reflect current state and priorities in the testing process. These strategies are explained further on in sections 3.4.2 and 3.4.3. The role of Test Lead is not mandatory in the process - the Tapir Framework can be used by a team of exploratory testers without a necessity to define this role. In such a case, functions related to prioritization and test data definitions are not available to the team and navigational strategies for team members are set by framework administrator.

3.2 System Under Test Model

For the purpose of systematic navigation of Exploratory Testers in the SUT, during our work and experiments with the Tapir Framework, we have evolved the following model. Originally, the model was inspired by the web application model proposed by Deutsch et al. [29]. During the work on the Tapir project, the model underwent major changes.

T denotes all exploratory testers testing the SUT. The set T includes testers $t_1, \ldots, t_n$. A tester can be given a role of Test Lead.

Definition 3.2.1 (System under test)**.** The SUT is defined as a tuple

$$(W, w_0, w_e, S, I, A, L, M)$$

where:

- W is a set of SUT pages,
- $w_0 \in W$ represents the home page (defined page, from which the exploratory testers start exploring the SUT),
- $w_e \in W$ represents the standard error page displayed during fatal system malfunctions (for instance an exception page in J2EE applications),
- S is a set of SUT state values,
- I is a set of input elements displayed to the user on the web pages of SUT user interface,
- A is a set of action elements (typically <`form`> element submit buttons),
- L a set of link elements displayed to the user,
- $M \subseteq W$ a set of user interface master pages. The Master Page models repeating components of the SUT user interface, for example, a page header with a menu or a page footer with a set of links. The definition of the Master Page is the same as a Web Page and the Master Pages can be nested (see definition 3.2.3).

Definition 3.2.2 (Data range)**.** $range(i)$ denotes particular data range which can be entered in an input element $i \in I_w$. The $range(i)$ can be either interval, or a set of discrete values (items of a list–of–values for instance). Then, $range(I_w)$ contains these ranges for the input elements of I_w.

Definition 3.2.3 (Web Page)**.** A Web Page $w \in W$ is a tuple $(I_w, A_w, L_w, \Theta_w, \phi_w, M_w)$, where

- $I_w \subseteq I$ is a set of input elements,
- $A_w \subseteq A$ is a set of action elements,
- $L_w \subseteq L$ is a set of link elements located on page w. As a Web Page w can contain more action elements which can perform actions with more than one form displayed on the page, in our notation $I_a \subseteq I_w$ contains a set of input elements connected to action element $a \in A_w$.

- Θ_w is a set of action transition rules $\theta : w \times range(I_w) \times A_w \rightarrow w_{next}$, where $w_{next} \in W$ is a SUT web page displayed to the user as a result of submitting an action element $a \in A_w$ with particular input data entered in input elements $I_a \subseteq I_w$.

- Φ_w is a set of action transition rules $\phi : w \times L_w \rightarrow w_{next}$, where $w_{next} \in W$ is a SUT web page displayed to the user as a result of clicking on a link element $l \in L_w$. Web pages accessible from a Web Page w by defined transition rules Θ_w and Φ_w are denoted as $next(w)$.

- $M_w \in M$ is a set of Master Pages of the page w, this set can be empty.

The model of Web Page and related concepts are depicted in 3.2. Parts of the model automatically re–engineered by the Tapir Framework during the exploratory testing process are depicted by a white background. Of these parts, elements specifically related to the interaction of the tester with the SUT are depicted by dotted background. Meta–data entered by Test Lead during the exploratory testing process are depicted by blue–gray background.

The SUT model is continuously built during the exploratory testing process. Team of testers T contributes to this process; W_T denotes SUT pages explored by the whole team, whereas W_t denotes SUT pages explored by tester $t \in T$. By analogy, L_T resp. A_T denotes SUT link, resp. action elements explored by the whole team and L_t resp. A_t denotes link, resp. action elements explored by tester $t \in T$.

By principle, a link or action element can be exercised more times during the test exploration process, also a page can be visited more times. To capture this fact, $visits(w)_t$, $visits(l)_t$, resp. $visits(a)_t$ denotes number of visits of the page w, link element l, resp. action element a by tester t. Further, $visits(w)_T$, $visits(l)_T$, resp. $visits(a)_T$ denotes number of visits of the page w, link element l, resp. action element a by all testers in testing team T.

For each input element $i \in I$, the Test Lead can define a set of equivalence classes $EC(i)$.

Definition 3.2.4 (Equivalence class)**.** Equivalence class $EC(i)$ determines the input test data, which shall be entered by the exploratory testers during the tests to an input element $i \in I$. When the equivalence class is not defined, $EC(i)$ is empty. Furthermore, for each $ec(i) \in EC(i)$ it holds, that if $range(i)$ is an interval, then $ec(i)$ is a sub–interval of $range(i)$, if $range(i)$ is a set of discrete values, then $ec(i) \in range(i)$.

$$\bigcap_{ec(i) \in EC(i)} ec(i) = \emptyset$$

for each $i \in I$.

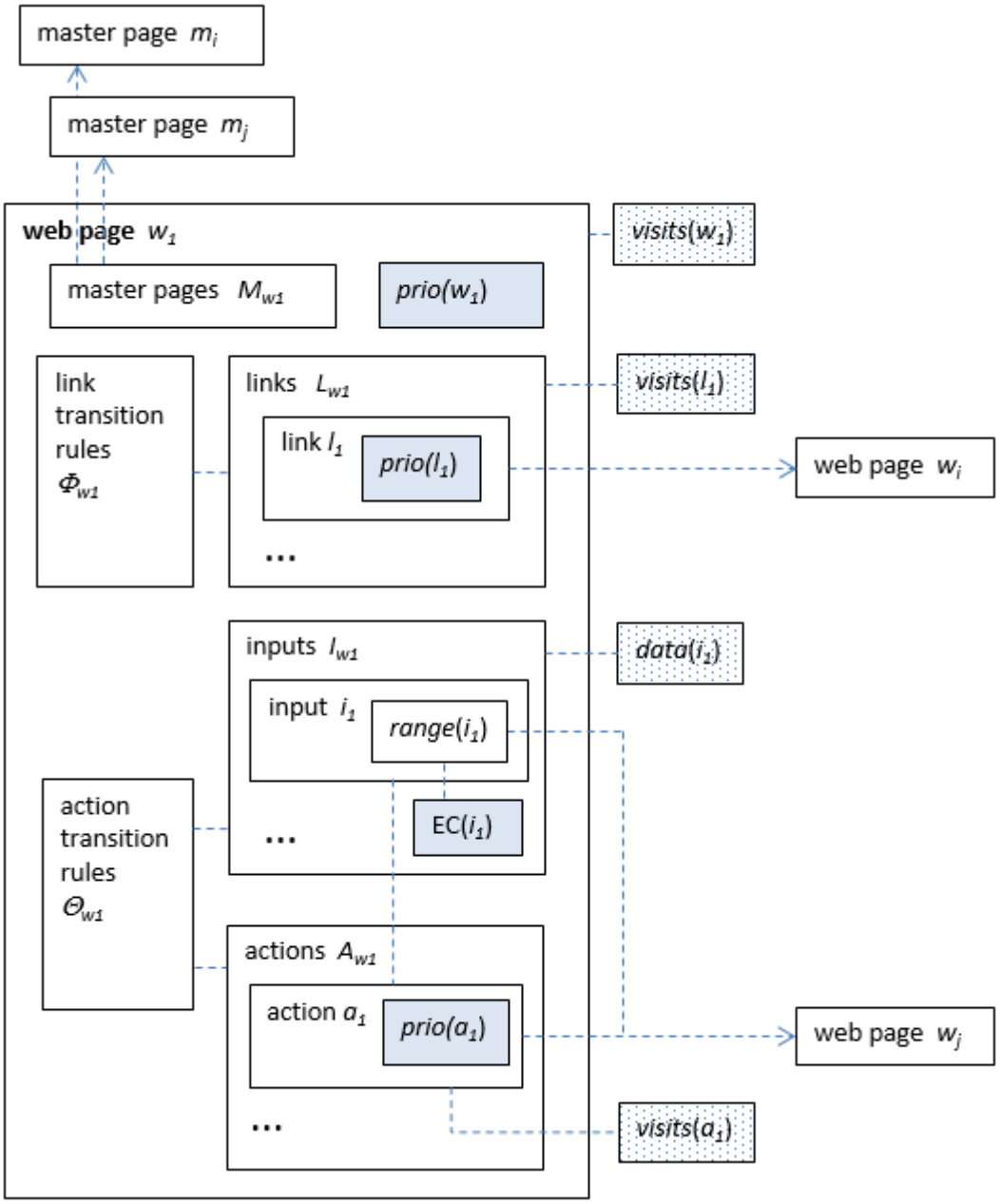


Figure 3.2: Model of SUT Web Page and related concepts

Equivalence classes can be defined dynamically during the exploratory testing process: some can be removed from the model, another added.

The data range $range(i)$ and equivalence class $ec(i)$ can contain an interval or a set allow Data range — comment interval values vs. discrete values (typically items from a list–of–values)

Moreover, $data(i)_t$, resp. $data(i)_T$ denotes a set of test data values entered to input element i by tester t, resp. by all testers in testing team T during the testing process.

A set of test data combinations entered by tester t, resp. by all testers in testing team T in input elements $I_a \subseteq I_w$ connected to action element a is denoted as

$$data(I_a)_t = \{(d_1, \ldots, d_n) \mid d_1 \in data(i_1)_t, \ldots, d_n \in data(i_n)_t,\ \ i_1 \ldots i_n \in I_a\}$$

resp.

$$data(I_a)_T = \{(d_1, \ldots, d_n) \mid d_1 \in data(i_1)_T, \ldots, d_n \in data(i_n)_T,\ \ i_1 \ldots i_n \in I_a\}$$

The Test Lead can also set a priority for selected elements of the SUT model. This is denoted as $prio(\mathfrak{X})$, $prio(\mathfrak{X}) \in \{1 \ldots 5\}$, 5 is the highest priority. $\mathfrak{X}$ can be particular web page $w \in W$, link element $l \in L$, or action element $a \in A$.

3.2.1 Discussion

During the evolution of the Tapir Framework, we explored also alternative modelling possibilities. In this section, we briefly present explored possibilities and we explain the reasons for selection of the final model.

The first explored possibility was to base the SUT model on the **UML** notation. Such an approach would have a potential advantage, which can be used in the testing process: the Tapir Framework would produce UML specification of the actual tested version of the system under test. Nevertheless, there were several reasons, why we dismissed this idea after several initial experiments. First, for automated computational processing, an underlying formal data structure is needed, so we would need to continuously convert UML to this structure. This might be a challenging task. Second, in the Tapir Framework we record model of the screens (SUT pages W), actions which trigger a transition to a next page in the SUT (A and L) and testing data which have been entered in input elements of the pages (I). In UML 2.0, there is not a diagram, which explicitly supports such a model and adoption of the current diagrams would be demanding task.

However, another possibility was available to explore — the **IFML** language. The use–case of the Tapir Framework and the underlying model we need for its functionality is conceptually most close to the IFML standard. In the middle of the evolution of the framework, we considered this alternative as prospective, and we did several experiments using the IFML as an underlying model [A.9]. Finally, we decided to return back to the

model as specified in Section 3.2. There were two major reasons for this decision: (1) As in case of the UML, for computational processing, an underlying formal data structure is needed. Real–time conversions of IFML model to such a structure would bring an additional overhead and complexity. (2) The case of the Tapir Framework became to be too much specific to just adopt the IFML. In the SUT model, we store the history of the exercised tests, entered testing data, equivalence classes and other related elements as specified in Section 3.2. Thus, we finally decided to use a straightforward model, specifically created for this case.

3.3 Build of the Model During Exploratory Testing

During the Exploratory Testing, the SUT model is incrementally built and updated by the following process:

1. When a tester $t \in T$ navigates to a web page $w \in W$ of the SUT, the browser tracking extension detects all link, action, and input elements and add them to temporary collections initialized for sets L_w, A_w and I_w.

2. The captured data are analyzed and used to build the model:

   a) If any of discovered elements have not been explored previously, the whole set of elements is added to the model.

   b) Otherwise, the model is updated by adding the newly discovered elements to the corresponding model collection.

   c) Discovered link elements are analyzed. The internal links are included in the model, the external links are ignored.

3. The recorded model can be also modified manually by the tester. This is a contingency option for the situations when the SUT model is recorded incorrectly due to various technical obstacles in the SUT.

During analysis of a web page $w \in W$ and its elements, the events like a form submit (form submit button is clicked) or navigation in the SUT (link element is clicked) are recorded. The actions which are performed only in the browser environment are ignored. For instance, we do not capture the process of choosing the colour by dragging the colour picker handle on the colour palette view, but the result of such action — filling in the value of selected colour into an input element that will be submitted later. Thus, the proposed solution does not support testing of the detailed behaviour of the page components.

In this process, SUT menu links are treated differently: clicking on these links usually starts a tested use case, but clicking on these links during the execution of particular test

can break the execution of tested use case. Thus, we treat these transitions in the SUT in a different way to the link transitions, which are the part of tested use case.

Then, in the web-based SUT, tested use cases usually consist of filling in the form input elements and clicking a button to submit the data to a server-side of the application. The action triggered by that button and the filled data are recorded, and a new transition is created. In the exploratory testing process, the tester can repeat already executed actions – for example when adding another order item to an order. In that case, only state data are recorded.

3.4 Generation of Navigational Test Cases from the Model

As explained above, the Tapir Framework generates high–level navigational test cases aimed to guide the group of exploratory testers through the SUT. The primary purpose of these test cases is to guide the tester in the SUT. The test cases are created dynamically from the SUT model during the ET process.

3.4.1 Structure of the Navigational Test Case

The navigational test cases are constructed for individual tester $t \in T$ dynamically from the SUT model during the exploratory testing process. The navigational test case is constructed for actual page $w \in W$ visited in the SUT and helps the tester to decide the next step in the exploratory testing process. The structure of the navigational test case is the following:

1. Actual page $w \in W$ visited in the SUT,
2. L_w (list of all link elements leading to other SUT pages accessible from the actual page). In this list, following information is given:
   a) $L_w \cap L_t$ (links elements leading to other SUT pages accessible from the actual page visited previously by the particular tester t),
   b) $visits(l)_t$ for each $l \in L_w \cap L_t$,
   c) $L_w \cap L_T$ (links elements leading to other SUT pages accessible from the actual page visited previously by all testers in the team T),
   d) $visits(l)_T$ for each $l \in L_w \cap L_T$, and
   e) $prio(l)$ and $prio(w_l)$for each $l \in L_w$. Link l leads from the actual page w to a page w_l.
3. A_w (list of all action elements leading to other SUT pages accessible from the actual page). In this list, following information is given:

a) $A_w \cap A_t$ (action elements leading to other SUT pages accessible from the actual page visited previously by the particular tester t),

b) $visits(a)_t$ for each $a \in A_w \cap A_t$,

c) $A_w \cap A_T$ (action elements leading to other SUT pages accessible from the actual page visited previously by all testers in the team T),

d) $visits(a)_T$ for each $a \in A_w \cap A_T$, and

e) $prio(a)$ for each $a \in A_w$.

4. The five best candidates/elements to explore in the next test steps for each of navigational strategies assigned to the tester t by the Test Lead. These elements are ordered by their rank computed by respective navigational strategy. As particular tester can have more navigational strategies available, these suggestions are displayed for each of assigned navigational strategies in a separate list and the tester can choose the optimal one, according to his personal testing strategy, choose within the navigational strategies set by the Test Lead. The elements suggested to explore are:

   a) A link $l_{next} \in L_w$ and page $next(w)$ suggested to explore in the next test step, or

   b) An action element $a_{next} \in A_w$ suggested to explore in the next test step.

5. For each $a \in A_w$, if $I_a \neq \emptyset$:

   a) $data(I_a)_t$,

   b) $data(I_a)_T$,

   c) for each $i \in I_a$:

      i. Suggested $ec(i) \in EC(i)$, determined by the test data strategy (see section 3.4.3) set by the Test Lead. Based on this suggestion, tester t can select a particular data value from $ec(i)$ to enter it to the input element i. for the actual test,

      ii. $data(i)_t$ (all test data data previously entered by the particular tester i to the input element i), and

      iii. $data(i)_T$ (all test data data previously entered by the all testers in the testing team T to the input element i).

6. Previous test data combinations entered in I_a, leading to display the error page w_e (typically a J2EE exception page for instance) or a standardized error message which can be recognized by the Tapir Framework (typically a PHP parsing error message or application specific error message formated in unified standard way for instance).

7. Notes for testers, which can be entered by the Test Lead to page w, all link elements from L_w and all action elements A_w. The Teas Lead can enter these notes as simple text fields (the notes are not defined in the model in the section 3.2).

The concepts of navigational test case are depicted in Figure 3.3. In this schema, L_{next} stands for suggested link elements to explore and A_{next} stands for suggested action links to explore. Web page $w_i \in next(w)$ and $w_j \in next(w)$.

In Figure 3.3, $L_w \cap L_T$ (links elements leading to other SUT pages accessible from the actual page visited previously by all testers in the team T) and $A_w \cap A_T$ (action elements leading to other SUT pages accessible from the actual page visited previously by all testers in the team T), are not depicted. These two parts of navigational test case are only an analogy to depicted $L_w \cap L_t$ (links elements leading to other SUT pages accessible from the actual page visited previously by the particular tester t) and $A_w \cap A_t$ (action elements leading to other SUT pages accessible from the actual page visited previously by the particular tester t).

3.4.2 Navigational Strategies

To create the navigational test cases during the exploratory testing process, several navigational strategies specified in Table 3.1 can be used. A navigational strategy determines a principal way how the tester will be exploring the SUT. The most of the navigational strategies can be further adjusted using the particular ranking function, specified in Table 3.2. The navigational strategies cover guided exploration of new SUT functions for all testers individually or as a collaborative work of the testing team, the same process enhanced by navigation driven by priorities of SUT pages, link and action elements, or regression testing for a defined historic period. This last strategy is also applicable to retests of defect fixes after a new SUT release.

Navigational strategy determines SUT user interface elements suggested for actual SUT page w in the navigational test case (see 3.4.1). Input of this process is an application context (tester t and related meta–data) and actual state of SUT model specified in 3.2. By the rules specified in Table 3.1 and ranking functions specified in Table 3.2, a list of $l \in L_w$ and $a \in A_w$, sorted by these rules and functions, is created.

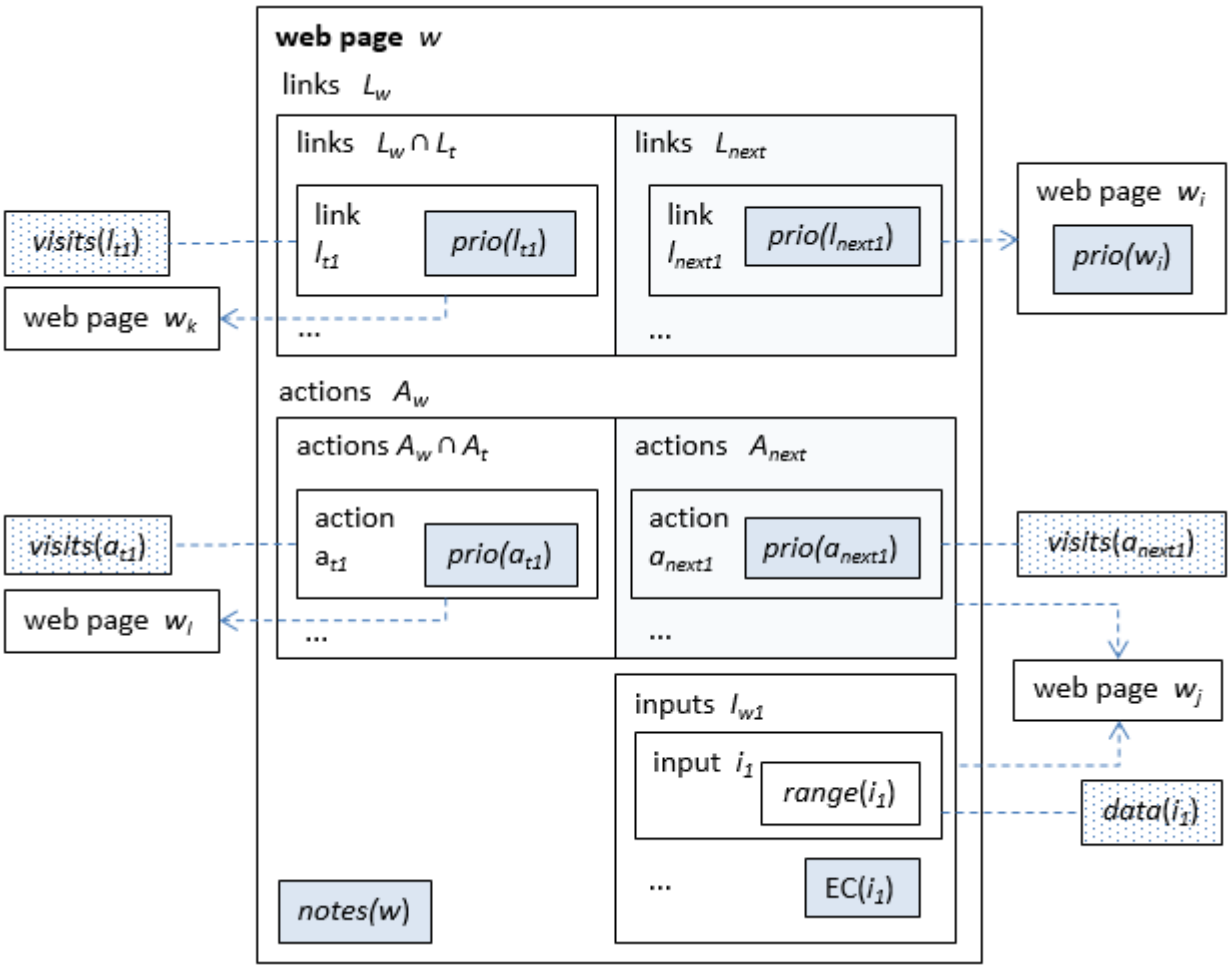


Figure 3.3: Model of navigational test case and related concepts

Navigational strategy	Rules for element suggestion for page w and tester t. Elementε can be link element $l \in L_w$ or action element $a \in A_w$.	Ranking functions (see Table 3.2) used	Use case
RANK_NEW	ε satisfying the following conditions: (1) $visits(\varepsilon)_t = 0$, AND (2) ($\varepsilon$ has the highest $ElementTypeRank(\varepsilon)$ OR a page $w_n \in next(w)$ to which ε leads has the highest $PageComplexityRank(w_n)$), AND (3) $\varepsilon \in w \in W \setminus M$ are preffered to $\varepsilon \in w \in M$	*ElementTypeRank* *PageComplexityRank*	Exploration of new SUT functions
RANK_NEW_TEAM	As RANK_NEW, (1) modified to: $visits(\varepsilon)_T$ has the minimal value among all $\varepsilon \in L_w$ and $\varepsilon \in A_w$.	*ElementTypeRank* *PageComplexityRank*	Exploration of new SUT functions
RT_TIME	ε satisfying the following conditions: (1) $visits(\varepsilon)_t > 0$, AND (2) time elapsed from the last exploration of ε by tester $t > LastTime$ constant, AND (3) $\varepsilon \in w \in W \setminus M$ are preferred to $\varepsilon \in w \in M$	-	(1) Retesting of defect fixes, (2) Regression testing
PRIO_NEW	ε satisfying the following conditions: (1) $visits(\varepsilon)_t = 0$, AND (2) $prio(\varepsilon)$ has the maximal value among all $\varepsilon \in L_w$ and $\varepsilon \in A_w$, AND (3) if ε is a link element $l \in L_w$, page $w_n \in next(w)$ has the highest $PriorityAndComplexityRank(w_n)$, AND (4) $\varepsilon \in w \in W \setminus M$ are preffered to $\varepsilon \in w \in M$	*PriorityAnd ComplexityRank*	Exploration of new SUT functions by priorities set by Test Lead
PRIO_NEW_TEAM	As RANK_NEW, (1) modified to: $visits(\varepsilon)_T$ has the minimal value among all $\varepsilon \in L_w$ and $\varepsilon \in A_w$.	*PriorityAnd ComplexityRank*	Exploration of new SUT functions by priorities set by Test Lead

Table 3.1: Navigational strategies

Rank	Definition
$ElementTypeRank(\varepsilon)$	IF ε is link THEN $ElementTypeRank(\varepsilon) = 1$ IF ε is action element THEN $ElementTypeRank(\varepsilon) = 2$
$PageComplexityRank(w_n)$	$PageComplexityRank(w_n) =$ $((((\mid I_w \mid \cdot inputElementsWeight$ $+\mid A_w \mid) \cdot actionElementsWeight +\mid L_w \mid)$ $\cdot linkElementsWeight$ $I_w \in w_n,\ A_w \in w_n,\ L_w \in w_n$
$PriorityAndComplexityRank(w_n)$	$PriorityAndComplexityRank(w_n) =$ $((((prio(w_n) \cdot pagePriorityWeight$ $+\mid I_w \mid) \cdot inputElementsWeight$ $+\mid A_w \mid) \cdot actionElementsWeight +\mid L_w \mid)$ $\cdot linkElementsWeight$ $I_w \in w_n,\ A_w \in w_n,\ L_w \in w_n$

Table 3.2: Ranks used in navigational strategies

Ranking function *ElementTypeRank* is used for both link elements $l \in L_w$ and action elements$a \in A_w$. The *PageComplexityRank* is used for link elements $l \in L_w$ only. In case of actions elements we are not able to determine exact SUT page following the process triggered by an action element a, as entered test data also can play a role in decisioning which page will be displayed in the next step (refer to the SUT model in Section 3.2).

In the *PageComplexityRank* ranking function, constants $actionElementsWeight$, $inputElementsWeight$, and $linkElementsWeight$ determine how strongly the individual page action elements, input elements and link elements are preferred in determination of the page $w_n \in next(w)$, which is suggested to be explored in tester's next step via exercising a link element leading to w_n. The increase of particular constant will cause the pages with higher numbers of particular elements to be more preferred. All of these constants can be set by Test Lead dynamically during the exploration testing process. Their default value is 256. Without any change, pages with higher number of forms, then with higher number of input fields, the higher number of action elements and finally with higher number of link elements are considered as more complex for the testing purposes and suggested to be explored first.

In the *PriorityAndComplexityRank* priorities of the SUT pages set by Test Lead plays the strongest role in the determination of the suggested next page to explore. Constant $pagePriorityWeight$ determines, how strong role plays this prioritization. Then, the decision is influenced by the number of input fields, the number of action elements and finally by the number of link elements. The constants $actionElementsWeight$, $inputElementsWeight$, and $linkElementsWeight$ have the same meaning and function as in the *Page-*

ComplexityRank.

3.4.3 Test Data Strategies

During the construction of navigational test cases, test data are suggested for input elements $I_a \subseteq I_w$ connected to action elements $a \in A_w$ of the particular page $w \in W$. For this suggestion, test data (1) previously entered by the testers ($data(i)_t$ and $data(i)_T$ for each $i \in I_a$) and (2) equivalence classes defined by the Test Lead ($EC(i)$ for each $i \in I_a$) are used.

For this process, test data strategies described in Table 3.3 are available. These test data strategies are specifically designed for different cases in the testing process: retesting of defect fixes, regression testing or exploration of new test data combinations.

Test data strategy	Description	Use case
DATA_REPEAT_LAST	For each $i \in I_a$, suggest the value of $data(i)_t$ used in the last test made by tester t on page w. If $data(i)_t = \emptyset$, no suggestion is made.	(1) Retesting of defect fixes, (2) Regression testing
DATA_REPEAT_RANDOM	Suggest a randomly selected test data combination from $data(I_a)_t$. If $data(I_a)_t = \emptyset$, no suggestion is made.	Regression testing
DATA_REPEAT_RANDOM_TEAM	Suggest a randomly selected test data combination from $data(I_a)_T$. If $data(I_a)_T = \emptyset$, no suggestion is made.	Regression testing
DATA_NEW_RANDOM	For each $i \in I_a$: if $EC(i) \neq \emptyset$, suggest a $ec(i) \in EC(i)$, such that $d \notin ec(i)$ for any $d \in data(i)_t$ if $EC(i) = \emptyset$, suggest a value $d \in range(i)$, such that $d \notin data(i)_t$	Exploration of new test data combinations
DATA_NEW_RANDOM_TEAM	For each $i \in I_a$: if $EC(i) \neq \emptyset$, suggest a $ec(i) \in EC(i)$, such that $d \notin ec(i)$ for any $d \in data(i)_T$ if $EC(i) = \emptyset$, suggest a value $d \in range(i)$, such that $d \notin data(i)_T$	Exploration of new test data combinations
DATA_NEW_GENERATED	The Tapir engine suggests combination, which was not used previously by individual tester t. Combination of test data is taken from a pipeline of test data combinations created by a combination interaction testing (CIT) module, connected by the defined interface	Exploration of new test data combinations
DATA_NEW_GENERATED_TEAM	As DATA_NEW_GENERATED_TEAM, modified to: combination, which was not used previously by any tester of the testing team T	Exploration of new test data combinations

Table 3.3: Test data strategies

As in the case of navigational strategies, the test data strategies for independent exploration of the SUT by individual testers or team collaboration are available. They are marked by postfix "_TEAM" in name of the test data strategy.

Test data strategy DATA_NEW_RANDOM_TEAM aims at minimization of particular test data variants entered repeatedly by multiple testers during the exploration of new test data variants - either by chance, or by a not suitable organization of work. Another case is intended testing of defect fixes or regression testing, where DATA_REPEAT_LAST, DATA_REPEAT_RANDOM and DATA_REPEAT_RANDOM_TEAM strategies are available to save tester's overhead remembering the last entered test data. The team strategy DATA_REPEAT_RANDOM_TEAM can make the process even more efficient by minimizing the particular test data variants entered multiple times by multiple testers during the regression testing.

Equivalence classes entered by the Test Lead during his pioneer exploration of the SUT contribute to the prevention of entering test data, which are actually belonging to one equivalence class, thus exercising the same SUT behavior according to the SUT specification.

Possibility to connect the Tapir Framework to a combination interaction testing module (DATA_NEW_GENERATED and DATA_NEW_GENERATED_TEAM strategies) makes the process further more controlled and systematic — only the efficient set of test data combinations are used by the testers to exercise the SUT functions.

3.5 Framework Architecture and Implementation Details

As introduced in Section 3.1, the Tapir Framework consists of three principal parts: **Tapir Browser extension**, **TapirHQ** and **Tapir Analytics** module. The tester interacts with the SUT in a browser window with installed extension. In the second window, the tester has opened TapirHQ front–end application, which serves as the test management tool. Here, suggestions for the navigational test cases are presented to the tester. The Analytics module can be accessed by the testers, Test Leads or administrator as a part of the TapirHQ module and allows visualization of the actual state of the SUT model. The overall physical architecture of the Tapir Framework is depicted in 3.4. The Tapir Framework back–end part is physically distributed amond two back–end systems, (1) the TapirHQ Server, which ensures generation of the TapirHQ application front–end, communication with the Tapir Browser Extension and identity management of the Tapir Framework users, and (2) Tapir BE Server, maintaining the SUT model and providing computational functions related to the model.

In this section, we give additional implementation details of the functionality of the individual framework modules.

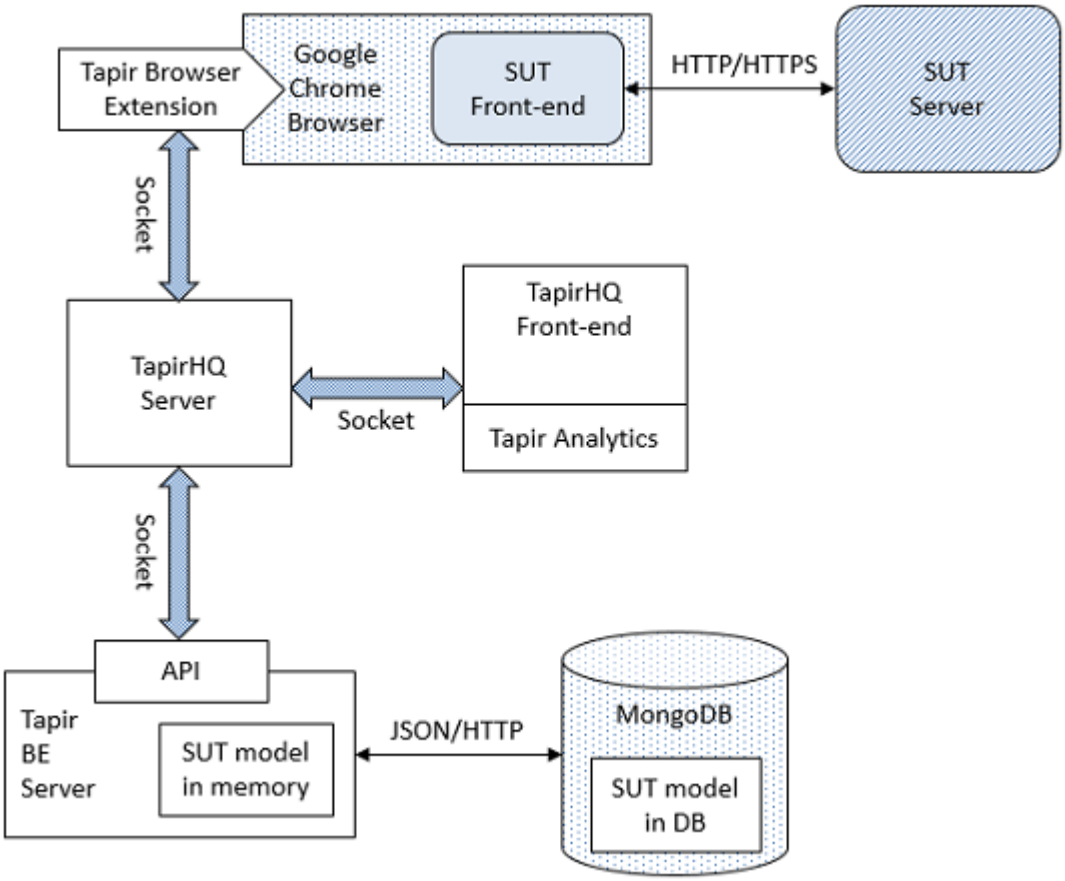


Figure 3.4: Overall architecture of the Tapir Framework

3.5.1 Tapir Browser Extension

The browser extension is implemented as a Chrome browser extension in JavaScript. It analyzes the structure of the current page, intercepts the internal browser events — since it would be difficult to intercept them on the server side (e.g. page was loaded or redirected, user navigated back, authentication is required, ...) and registers event handlers for all the links and buttons on the page. All events relevant to the Tapir Framework functionality are intercepted and tracked. Also, the browser extension has a functionality to highlight SUT page elements in a mode selected by the Test Lead (elements already analyzed by the Tapir Framework or elements suggested to explore in the next tester's step).

The extension runs on top of the page with full access to the document object model of this page. In the previous stages of the implementation, we have been experimenting with JavaScript snippet code injected into each page of the web application, but the extension to the browser was proven to be a more stable solution.

3.5.1.1 SUT Page Analysis

To analyze the web pages W of the SUT and to capture all the relevant data, set of activities is performed by the Tapir Browser Extension. Although the content of the SUT page $w \in W$ is created using the common technologies (HTML, CSS and JavaScript), the logical structure of the SUT page has to be analyzed to update the SUT model. For a new SUT analyzed by the Tapir Framework, the code of the page content analyzer (being part of the Tapir Browser Extension) had to be adapted to match its nuances.

The analyzer extracts selected properties for every link, input and action element. CSS selector is extracted for each element, the element will be later located using this selector (when more than one element is matched, then elements are selected with the help of additional extracted properties). In this section we present technical details of this analysis for the main elements of the SUT page w, the action elements A_w, input elements I_w and link elements L_w.

Action elements. An action element is represented by an HTML `input` element with attribute `type` set to `submit` that is nested under a `form` element. When rendered on the page, it has the appearance of a button — a button submitting the form. Currently, from the Tapir Framework perspective, the following properties are extracted for the parent form element:

- action — the URI of a program that processes the form information (value of the attribute with the same name)
- method — the HTTP method (POST or GET) that the browser uses to submit the form (value of the attribute with the same name)
- name – the name of the form (value of the attribute with the same name)
- CSS selector.

For the action element $a \in A_w$, the following properties are extracted:

- value — the text displayed (value of the attribute with the same name)
- type — in this case, the value is `submit`
- CSS selector.

Input elements. An input element is represented by an HTML `input` element with attribute `type` not set to `submit` (including hidden input elements) or by `<select>` or by `<textarea>`) element that is nested under a `form` element. The following properties are extracted for an input element $i \in I_w$:

- name (value of the attribute with the same name)
- label — the label is not part of the element and the detection of its value differs, for example the value is located in `<label>` element whose value of `for` attribute matches the value of `id` attribute of the input element
- type of the element (the value of the `type` attribute for `input` elements, name of the element otherwise)
- CSS selector.

During this analysis, corresponding input elements I_a are assigned to particular action element $a \in A_w$.

Link elements. A link element is represented by an HTML `<a>` element with `href` attribute. Modern web application use `<a>` elements with the use of JavaScript as action elements. Links with `href` pointing to local element (starting with '#' symbol, `<a href="#heading">...</a>`) or with JavaScript code (`<a href="javascript:...">...</a>`) are excluded. For a link element $l \in L_w$, the following properties are extracted:

- value — the inner text value of the link
- location — the URL where the link goes to (the value of the `href` attribute)
- query — the optional part of the URI containing data that does not fit conveniently into a hierarchical path structure; cutting off the query string it is possible to get the system identifier of the target node
- local — value indicating local or external link
- target — value specifying where to open the linked document (value of the attribute with the same name)
- CSS selector.

The page content analyzer recursively traverses the Document Object Model (DOM) tree of the web page and collects the information on the elements mentioned before. The analyzer is grouping the elements into larger groups — for instance, HTML element `<form>` is not defined as a SUT model element, but on the level of HTML code, it is an element that is a parent to input, and action elements, and possibly also links. Another example is a horizontal menu on the of the page. This example is depicted in Figure 3.5. The groups are emphasized with the green border. During the process, the blocks can be further analyzed to determine master pages M, particular forms and relevant groups of input elements I.

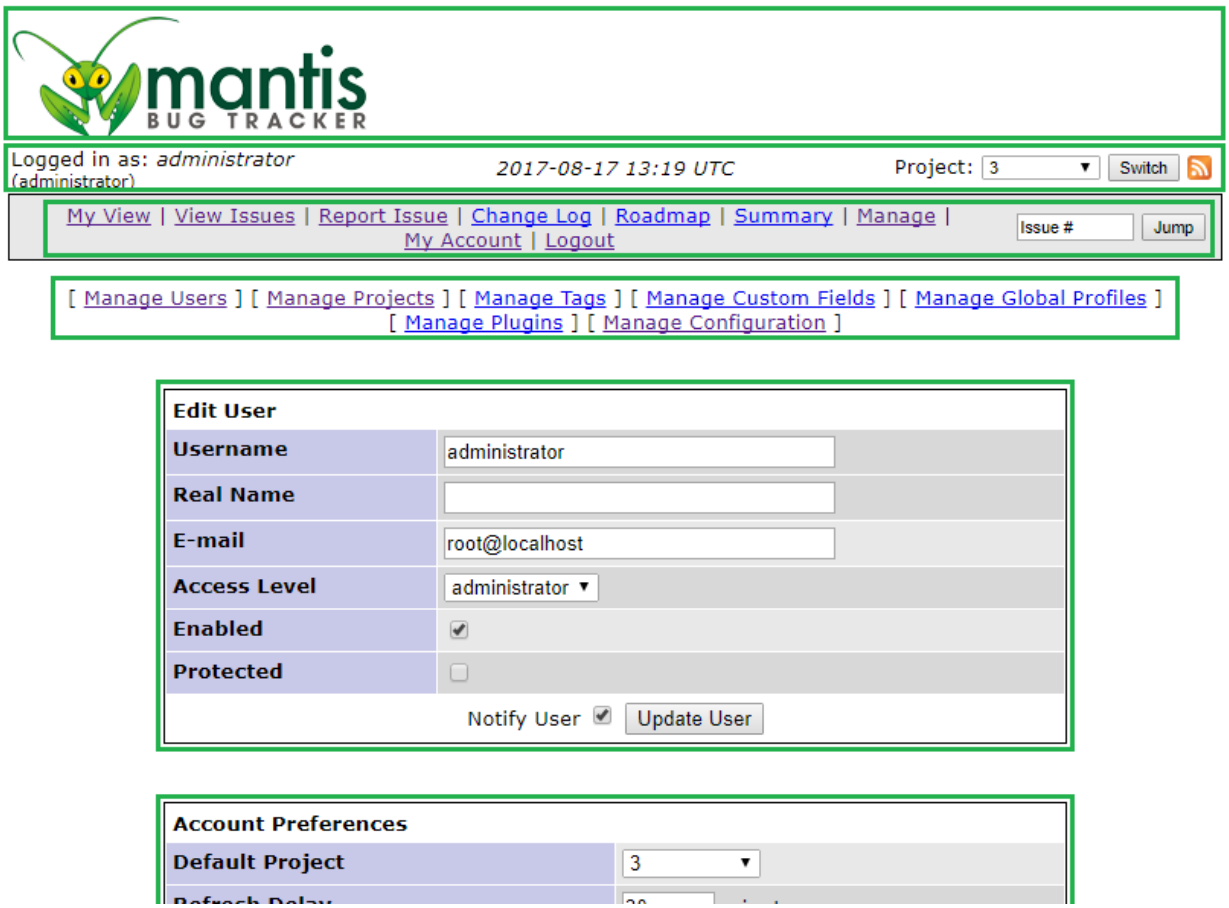


Figure 3.5: Logical grouping of elements on the web page

part of this analysis is pre–processed by the Tapir Framework, the final arrangement of these elements in the SUT model can be refined by the Test Lead. The pseudocode of the page content analyzer main algorithm is presented in Algorithm 1.

3.5.1.2 Tester's Session Recording

Actions performed by the tester t in the SUT in the browser are recorded by the Tapir Browser Extension. A shortened example of a recorded session as displayed in TapirHQ is presented in Fig. 3.6. During the recording of tester's session, the following sequence of steps is performed:

- Tester navigates the browser to an initial page of the SUT, which is part of the configuration of the Tapir Framework.
- When the page requested by tester is not the initial page, the tester is informed by Tapir to navigate to the initial page.

Input: HTML element E
Output: Groups of model elements extracted from DOM tree fragment with E being the root of that fragment
begin
  for ***every*** *element C, C is child of E* **do**
    switch C **do**
      case *<a> with valid value of* ***href*** *attribute* **do**
        extract element properties, add to group; ▷ the reason this element is treated differently compared to action and input elements is, that those elements are nested inside `<form>` element
      case *<form>* **do**
        extract element properties for each nested `<input>`, `<select>` and `<textarea>` element
      case *layout <table>* **do**
        recursively call this for C; ▷ layout table is a table created to arrange elements, for example inside a `<form>` element
      case *data <table>* **do**
        collect data for table headers, cells and rows; ▷ a table displaying data, it can be identified by the presence of table header cells `<th>` or by row elements `<tr>` with certain CSS classes (for example `row-1`, `row-2` for alternating row styles)
      otherwise do
        recursively call this for C;
      end
    end
  end
end

Algorithm 1: Pseudocode of the page content analyzer

  - An API request from the Tapir Tracking Extension to the TapirHQ server is sent, the response indicates whether the site is Tapir–enabled (a configuration exists in the Tapir database).
  - Another request from the Tapir Tracking Extension to the TapirHQ server is made to get the configuration for the site (for example the definition of page conditions).

- Tester's session is started when the tester lands on starting page and from this point, every action is recorded by the Tapir Browser Extension. The following events are recorded:
  - form submitted event,
  - link clicked event,

  - an error was detected by Tapir event,
  - page reload event (the user has refreshed the page using browser command, usually by pressing F5 key),
  - client redirect event caused by JavaScript running in the page or a "refresh" pragma in the page's meta tag[1],
  - server redirect event caused caused by a 3XX HTTP status code sent from the server[1],
  - history navigation event (the user has moved back or forward to a previously visited page, usually by pressing the combination of Alt and Right or Left Arrow keys),
  - address changed event when the user triggers the navigation from the address bar.

- The tester's session is closed when the tester:
  - closes the browser or the browser tab,
  - navigates away from the SUT, or
  - has been inactive for a period of time, defined in the tester's profile configuration.

3.5.1.3 Authentication

For tester's authentication in the Tapir Framework, standardized Google authentication is used. Standard web authentication protocols utilize HTTP features, but Chrome Extensions run inside the application container; they don't load over HTTP and can't perform redirects or set cookies. Thus, with the use of Google Identity API the user can be easily authenticated using Google authentication (and the Google Account the user is currently logged to in the browser)[2] by calling a single method `getAuthToken`. If the user is not authenticated a pop–up window is displayed with the Google authentication dialog. The 3rd party authentication was not implemented, although it is technically possible.

3.5.1.4 Support for Other Browsers

Currently, the browser extension is developed for the Google Chrome browser. We decided to use Google Chrome browser because of its current largest market share (76.7% by W3C statistics up to July 2017[3]). The future road–map of the project includes also the creation

[1] https://developer.mozilla.org/en-US/Add-ons/WebExtensions/API/webNavigation/TransitionQualifier

[2] https://developer.chrome.com/apps/app_identity

[3] https://www.w3schools.com/Browsers/default.asp

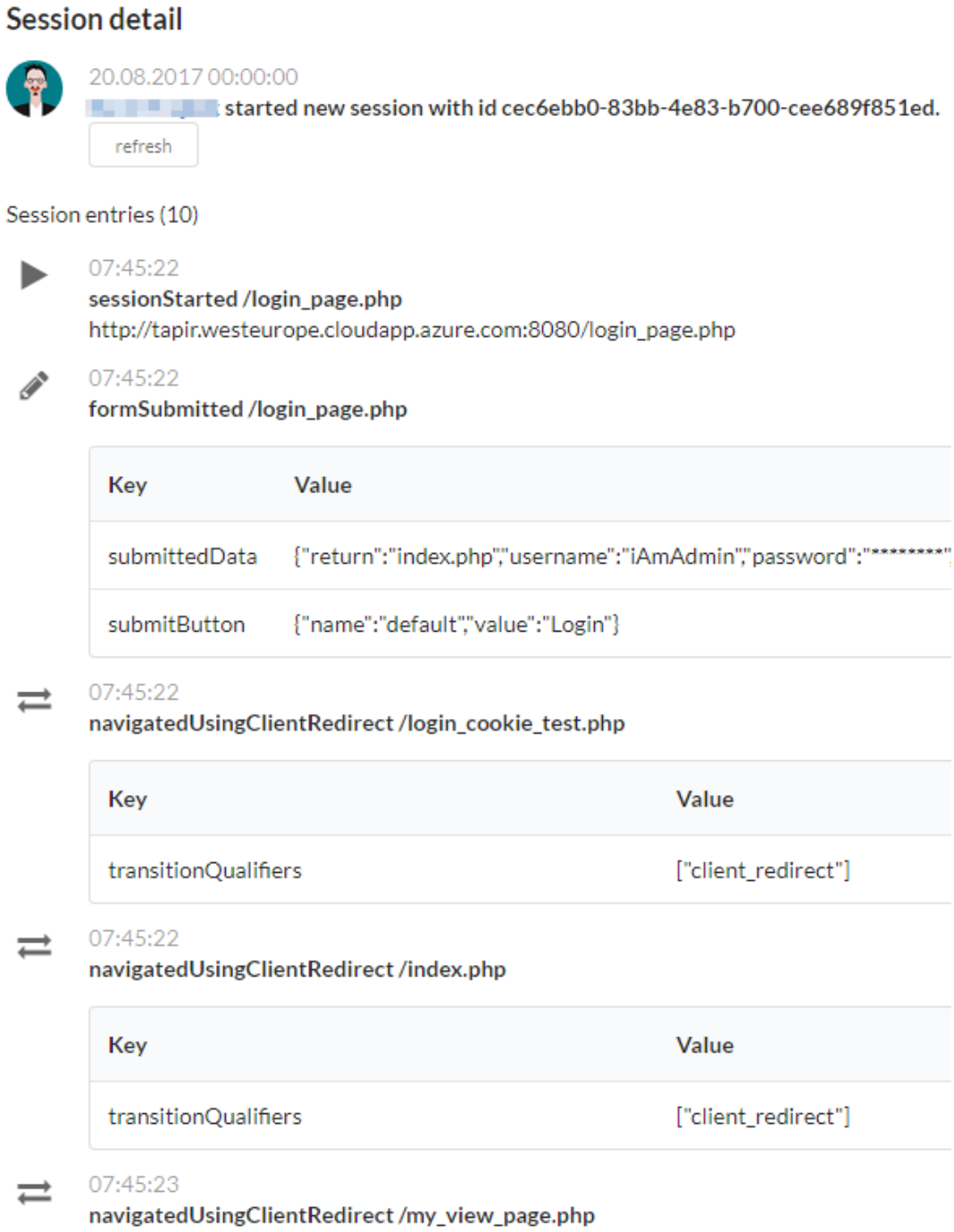


Figure 3.6: Recording of tester's session displayed in TapirHQ module

of Firefox Tapir Browser Extension. Firefox WebExtensions are designed for cross–browser compatibility. The Firefox technology is declared to be compatible with the extension API supported by Google Chrome and Opera. Chrome and Opera extensions shall in most cases run in Firefox with minor changes only[4]. Currently, the porting of the Chrome extension to a Firefox version of the browser extension is in progress.

3.5.2 TapirHQ

TapirHQ represents the core of the Tapir Framework functionality. This module receives the events from the Browser Extensions, constructs the SUT model, constructs the navigational test cases and present them to the tester.

Figure 3.8 depicts tester's navigation support in the SUT. The left side (the larger screenshot) depicts Tapir HQ guidance for the tester. Besides the navigational statistics, the system displays suggested actions to be explored (sorted by ranking functions) by several navigational strategies allowed to the user. Regarding the format of this article, the view is simplified: test data suggestions and other details are not visible in this sample. In Figure 3.8 a corresponding screenshot from SUT is presented. The Tapir Browser Extension highlights the elements to be explored in the next step, together with the value of the ranking function. The Figure 3.9 depicts a sample of recorded details of a SUT form.

The TapirHQ is a client application implemented as a JavaScript single page application using the ReactJS framework. The server back–end part is implemented in .NET C#. When a tester starts testing the SUT – the tester has to land on the staring page (defined as w_0 in 3.2), a real–time bidirectional event–based communication channel (a socket) is opened between TapirHQ back–end service and TapirHQ front–end application in the browser to synchronize the data in real–time. For this communication, the SocketIO[5] library is used.

TapirHQ also contains an open interface to Combination Interaction Testing tool to import preferred test data combinations (ref. to test data strategies DATA_NEW_GENERATED and DATA_NEW_GENERATED_TEAM). The interface is based on upload of CSV files of defined structure, or, alternatively, in a predefined JSON format.

The SUT model is stored in a NoSQL database MongoDB[6]. The document–oriented NoSQL database was selected because the JSON documents are first–class citizens there and can be stored in this database directly.

The data with SUT model stored in the database is shared by both TapirHQ and Tapir Analytics modules. The TapirHQ back–end service exposes the API for indirect access to the database by the individual modules.

[4]https://developer.mozilla.org/en-US/Add-ons/WebExtensions/Porting_a_Google_Chrome_extension
[5]https://socket.io/
[6]https://www.mongodb.com/

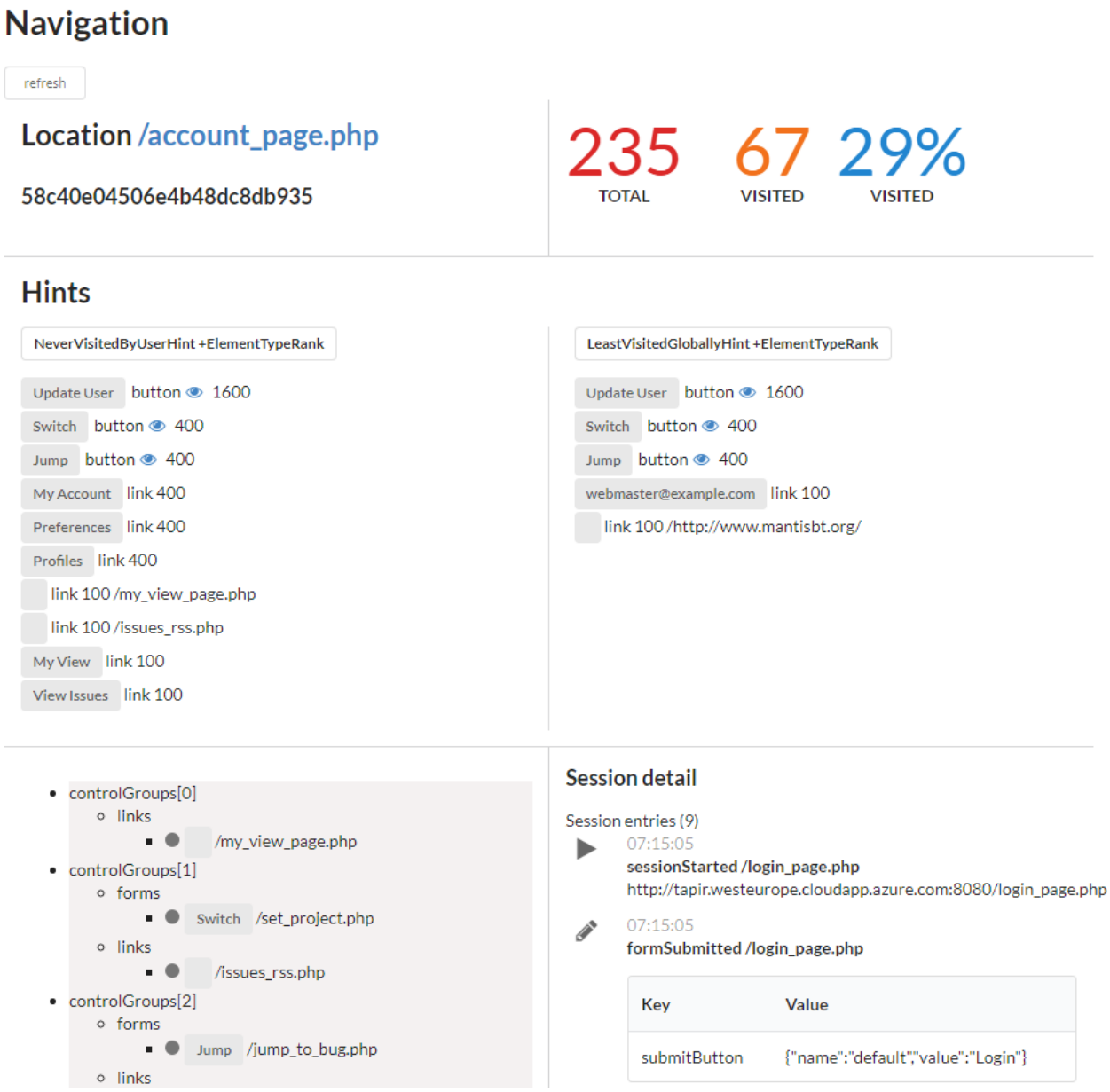


Figure 3.7: A sample of testers' navigational test case (simplified)

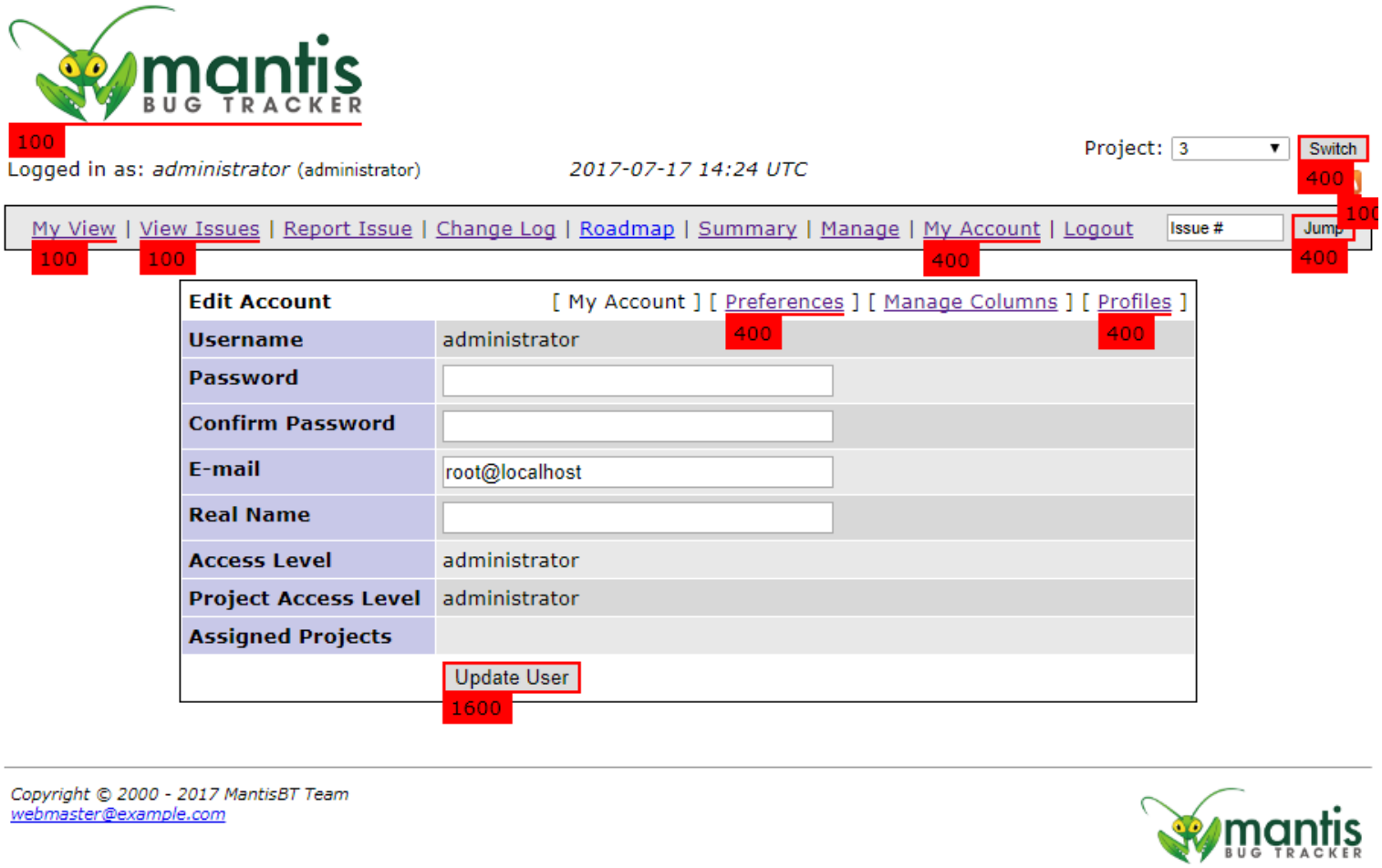
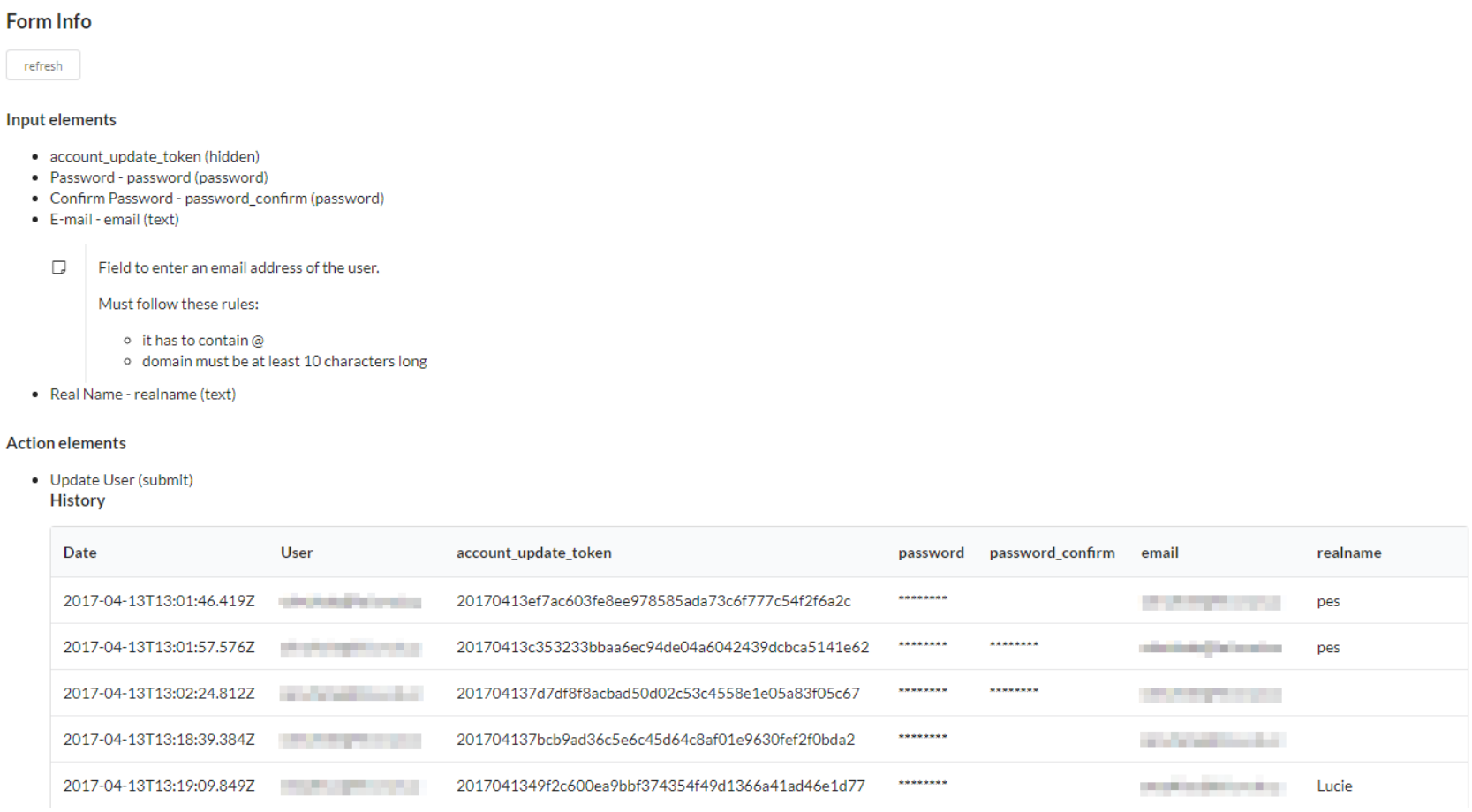


Figure 3.8: A sample of SUT screen with highlighted elements with hints



Form Info

refresh

Input elements

- account_update_token (hidden)
- Password - password (password)
- Confirm Password - password_confirm (password)
- E-mail - email (text)

  Field to enter an email address of the user.

  Must follow these rules:

  - it has to contain @
  - domain must be at least 10 characters long
- Real Name - realname (text)

Action elements

- Update User (submit)

History

Date	User	account_update_token	password	password_confirm	email	realname
2017-04-13T13:01:46.419Z		20170413ef7ac603fe8ee978585ada73c6f777c54f2f6a2c	********			pes
2017-04-13T13:01:57.576Z		20170413c353233bbaa6ec94de04a6042439dcbca5141e62	********	********		pes
2017-04-13T13:02:24.812Z		201704137d7df8f8acbad50d02c53c4558e1e05a83f05c67	********	********		
2017-04-13T13:18:39.384Z		201704137bcb9ad36c5e6c45d64c8af01e9630fef2f0bda2	********			
2017-04-13T13:19:09.849Z		2017041349f2c600ea9bbf374354f49d1366a41ad46e1d77	********			Lucie

Figure 3.9: A sample of recorded details of a SUT form

3.5.3 Tapir Analytics

Tapir Analytics module visualises the current state of SUT model in a textual representation or in a form of directed graph and a particular state of SUT exploration. This part is implemented as a module of TapirHQ, sharing the SUT model with the TapirHQ application. The framework administrator grants access rights to this module. This part is implemented in .NET C#. Visualization of the SUT model is implemented using ReactJS framework.

Figure 3.10 depicts a sample of Test Lead's overview of part of the SUT model — a particular SUT page. Prioritization of SUT pages and elements can be done in this function. Figure 3.11 then depicts a sample from Analytics module — visualisation of SUT pages and possible transitions between the pages.

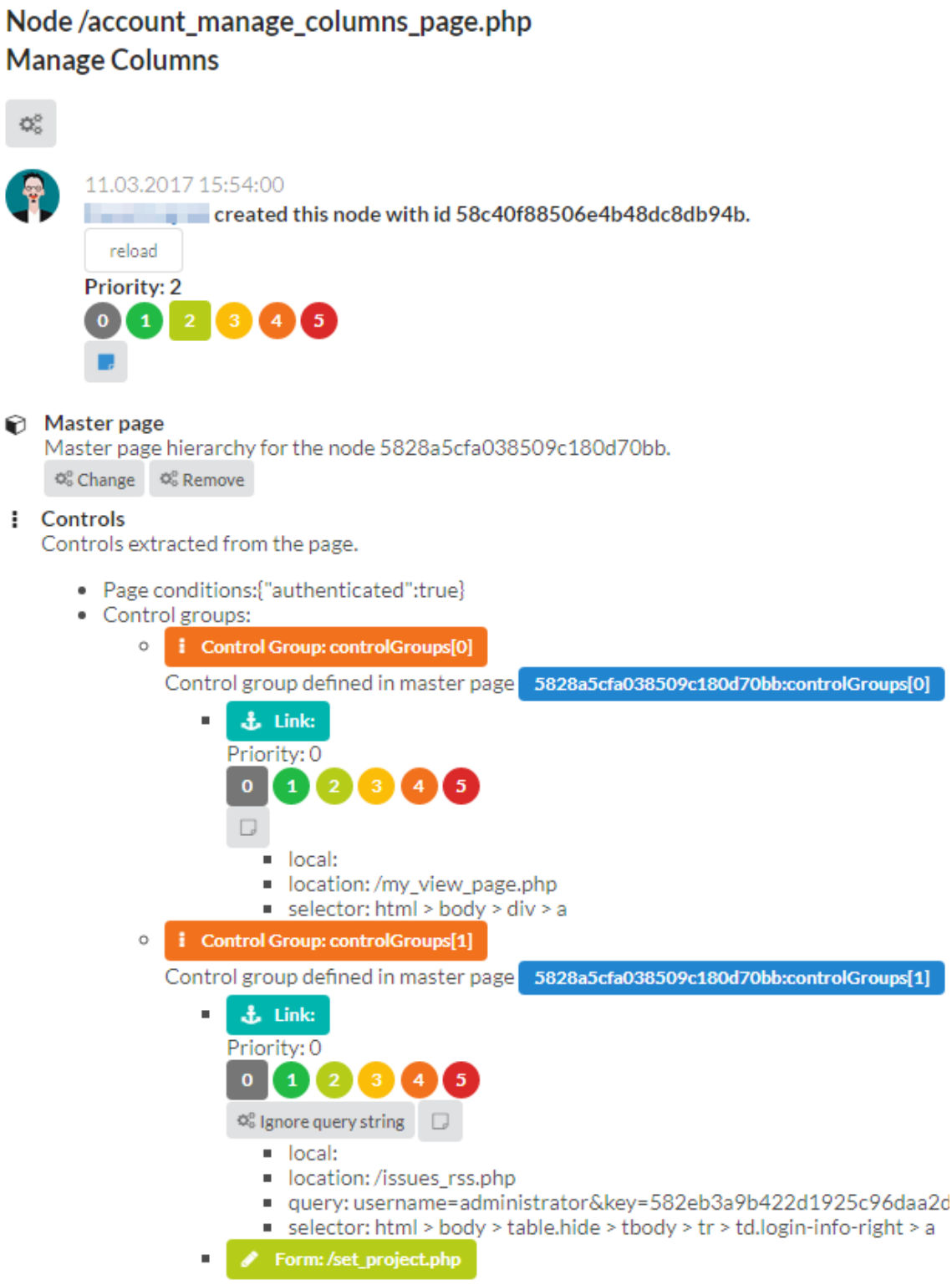


Figure 3.10: A sample of Test Lead's overview of part of the SUT model

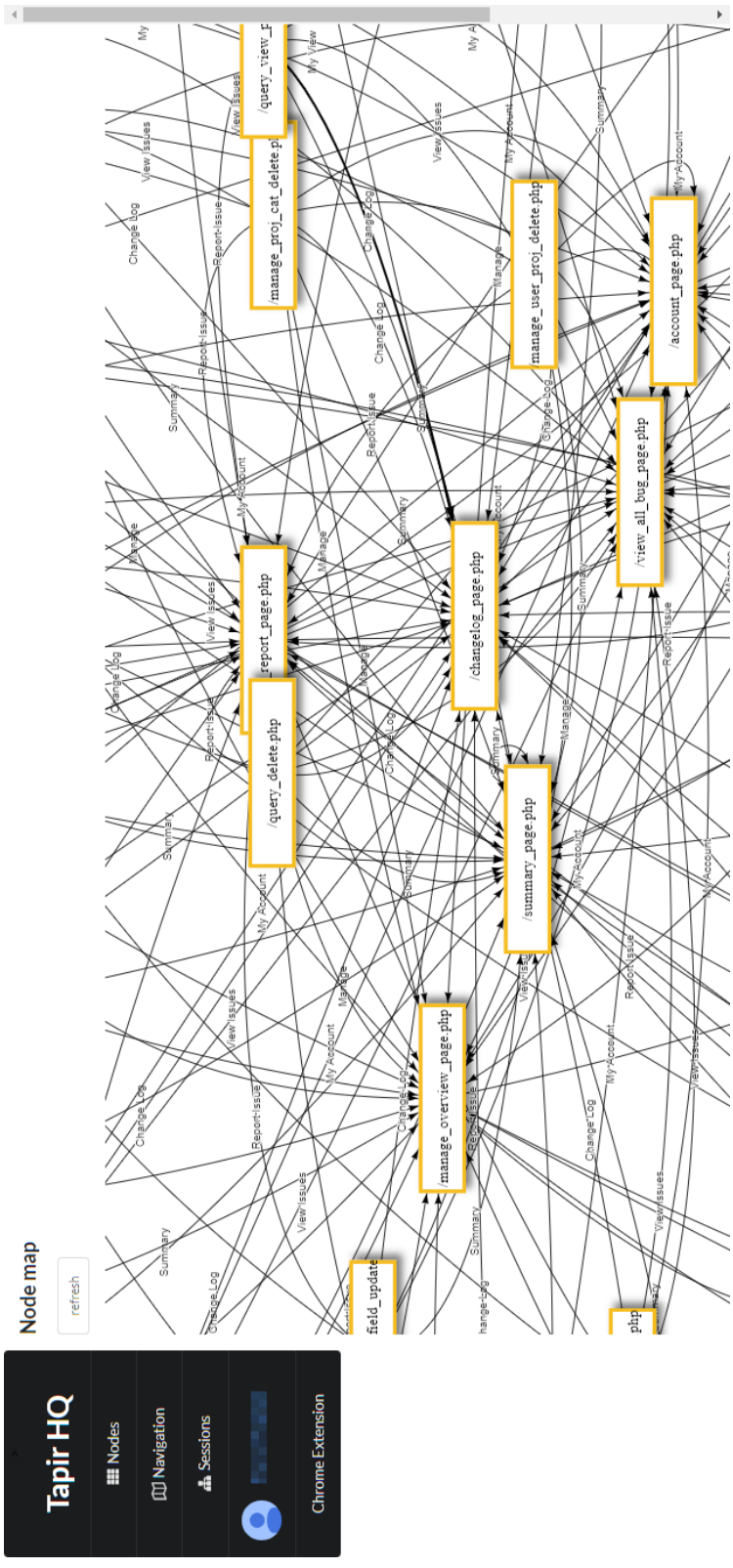


Figure 3.11: A sample from the Tapir Analytics module – SUT pages and possible transitions between them

3.5.4 Handling the Changes in the SUT

The SUT usually does evolve and change even during the testing phase of the software project — the defects are fixed, the end–users require new features, the UI is changed to reflect the new features (the elements were moved on the page, new elements were added, the page was restructured and other changes can be made). These changes shall be reflected in the model of the SUT and the model has to be up–to–date with the application. The changes cannot be incorporated automatically into the model of the SUT since the impact on the whole model cannot be predetermined.

There are several possible scenarios how to deal with these changes, depending on a size of the modification and a fact, if the testing team knows the exact location of the changes:

1. **Major part of the system was affected by the change, including page structure and page flow.**
   If the change in SUT is so significant, that complete re-testing of SUT is needed, administrator archives previous SUT model and the testing team explores the SUT building a new up-to-date model.

2. **Significant part of the system was affected by the change, including page structure and page flow, impact analysis of the changes is not known.**
   Undocumented changes made to the SUT are not desirable in a software development process. This type of changes will affect manual testers as well; the manual test scenarios will become obsolete and not corresponding to an actual state of the SUT. Unfortunately, when a change is not documented, also the Tapir Framework cannot provide an efficient support here; a correct solution is to archive the previous version of the SUT model and start building a new one again.

3. **Significant part of the SUT was affected by the change, and the testing team knows exactly, where these changes have been made.**
   Using the release notes or an impact analysis document the user with the appropriate permission can invalidate affected parts of the model and let the testers to explore those parts first (a “dynamic priority” can be used in this case).

4. **Changes known to the testing team affect only small parts of the SUT (only the content of pages is changed, the page flow is not affected).**
   Again, the release notes or an impact analysis document can be used to invalidate affected parts of the model that describe the content of the page and testing data.

Chapter 4

Experiments

To analyze the practical applicability of the proposed Tapir Framework and its efficiency when used to support Exploratory Testing process, we conducted four case studies, which we present in this chapter.

The aim of Case Studies 1 and 2 was to compare the efficiency of the Tapir Framework with the manually performed Exploratory Testing process. In these case studies, we split the testers to two principal groups. The first group was using the support of the Tapir Framework. The second group was only monitored using the Browser Tracking extension. During this process, the SUT model was constructed, but no navigational support was provided to the testers of this group. In the both case studies, we compared data collected from constructed models. In the Case Study 1, we also collected testers reports on time spent on individual tasks in the testing process. In the Case Study 2, we relied more on time measurement connected to model construction. The Case Study 2 was performed using a more recent prototype of the Tapir Framework and the method also differed in several points which we explain further on.

The aim of Case Study 3 was to assess proposed navigational strategies and suggest the best option to be evolved further on. We compared team versus individual navigational strategies, as well as alternative ranking functions.

Case Studies 1–3 primarily focused to an efficiency of the process of exploration of new SUT functions, as we consider this activity as a main use case of the Tapir Framework.

The aim of Case Study 4 was to assess the applicability of the Tapir Framework to different types of SUTs and also to assess its applicability when used on an SUT with dynamically generated front–end HTML pages, including the structure of the pages and URL format.

The case studies 1–4 were aiming to answer seven research questions following in the section 4.1.

Table 4.1: Research questions and Case Studies and that are answering them

	Research question	Case study
Q1	In which sub-tasks of the Exploratory Testing process does the support provided by the Tapir Framework bring time savings and how significant these savings are?	1
Q2	Is the proposed approach more efficient than manual Exploratory Testing regarding detected defects?	1, 2
Q3	Will exploratory testers supported by the Tapir Framework cover a larger part of the SUT using the proposed approach?	2
Q4	What is an efficiency of ET process supported by the Tapir Framework measured by metrics based on particular key elements of the SUT model compared to this process performed manually?	2
Q5	Are there any aspects, where the Tapir Framework decreases the efficiency of the Exploratory Testing process, compared to its pure manual execution?	1, 2
Q6	Which of the proposed navigational strategies and ranking functions, designed for exploration of new parts of the SUT, are the most efficient?	3
Q7	How applicable to different types of SUTs the Tapir Framework is?	4

4.1 Research Questions

To assess the efficiency and applicability of the proposed solution, we defined seven research questions, which are summarized in Table 4.1. In this table, these research questions are linked to the case studies, which were performed to answer them.

4.2 System Under Test with Injected Defects

As a system under test for Case Studies 1–3, we used open–source MantisBT[1] issue tracker. This system is written in PHP and is using MySQL relational database. We modified the source code of the SUT by insertion of a group artificial defects. To automate reporting of activation of these artificial defects, we accompanied the defective code lines by logging mechanism, reporting each activation of the defective line code. For Case Study 1, we used a group of artificial defects specified in Table 4.2. For Case Studies 2 and 3, we used artificial defects specified in Table 4.3, because, based on the experiment results, we considered the defects injected for the Case Study 1 as rather too easy to detect.

The used version of the MantisBT (1.2.19) composed of 202964 lines of code, 938 application files, the database schema has 31 database tables. Also, we have scanned the

[1]https://www.mantisbt.org/

MantisBT source code for the occurrence of the elements relevant for SUT model. In the version of Mantis BT used for the case studies, the system composed of front–end 73 pages. These pages contained 117 input forms and 292 links in total.

Injected defect ID	Type	SUT function
synt_1	Syntax error	Plugin installation function broken
synt_2	Syntax error	Plugin uninstallation function broken
synt_3	Syntax error	Import issues from XML function broken
synt_4	Syntax error	Adding empty set of users to a project causes system defect of the SUT
synt_5	Syntax error	Setting configuration option with empty value causes system defect of the SUT
synt_6	Syntax error	Configuration option of float type cannot be created
synt_7	Syntax error	Configuration option of complex type cannot be created
synt_8	Syntax error	User cannot change the password for his account
synt_9	Syntax error	Roadmap page has a serious syntax error preventing it from being loaded
synt_10	Syntax error	Selected Tag filter links do throw an error
synt_11	Syntax error	Access level for changing status cannot be changed
synt_12	Syntax error	Cannot show users with global access when editing a project
synt_13	Syntax error	The action “Close” on search issues page to close selected issues throws an error
synt_14	Syntax error	Cannot view the issues in separate window (the “^” link on issue view page)
synt_15	Syntax error	Plugin setting cannot be modified
synt_16	Syntax error	Hiding issues with selected status (and above) is broken
synt_17	Syntax error	Workflow thresholds for selected operations cannot be modified
synt_18	Syntax error	Priority of the issue cannot be changed
synt_19	Syntax error	Jump to bug does not work
synt_20	Syntax error	JavaScript code to show UI to change the reporter of the issue does not work
mc_1	Missing code	Export to CSV is not implemented
mc_2	Missing code	Export to Excel is not implemented

mc_3	Missing code	The action "set sticky" in search issues screen is not implemented
mc_4	Missing code	Printing of the issue details is not implemented
mc_5	Missing code	User cannot be deleted
mc_6	Missing code	File cannot be uploaded and attached to an issue
mc_7	Missing code	New profile cannot be created
mc_8	Missing code	Logout does not work properly
mc_9	Missing code	A link to send an email to administrator is not working
mc_10	Missing code	Cannot filter the issues by OS
cc_1	Change in condition	Issue configuration option value cannot be set in database
cc_2	Change in condition	Issue configuration option value in not loaded properly from database
cc_3	Change in condition	Tag with the name "Tapir" (predefined in the SUT) cannot be deleted
cc_4	Change in condition	Properties of a read–only bug can be modified
cc_5	Change in condition	A project is wrongly considered to be read–only and its properties cannot be modified
var_1	Wrong set of variable	Language in user preferences is always "English" and cannot be changed
var_2	Wrong set of variable	User defined columns in issue list cannot be copied between projects
var_3	Wrong set of variable	When adding new bug note, its status cannot be "private"
var_4	Wrong set of variable	Bug note view status cannot be changed
var_5	Wrong set of variable	Only a university email address can be used when creating new account

Table 4.2: Defects injected to the system under test for the Case Study 1

4.3 Case Study 1: Evaluation of the Tapir Framework Efficiency

To answer the research questions **Q1**, **Q2** and **Q5** we conducted the following case study with a group of exploratory testers. This case study was performed using an initial proto-

Injected defect ID	Type	SUT function
synt_1	Syntax error	Plugin installation function broken
synt_2	Syntax error	Plugin uninstallation function broken
synt_3	Syntax error	Import issues from XML function broken
synt_4	Syntax error	Adding empty set of users to a project causes system defect of the SUT
synt_5	Syntax error	Setting configuration option with empty value causes system defect of the SUT
synt_6	Syntax error	Configuration option of float type cannot be created
synt_7	Syntax error	Configuration option of complex type cannot be created
mc_1	Missing code	Export to CSV is not implemented
mc_2	Missing code	The action "set sticky" in search issues screen is not implemented
mc_3	Missing code	Printing of the issue details is not implemented
mc_4	Missing code	User cannot be deleted
mc_5	Missing code	Bug note cannot be deleted
cc_1	Change in condition	Issue configuration option value cannot be set in database
cc_2	Change in condition	Issue configuration option value in not loaded properly from database
cc_3	Change in condition	Tag with the name "Tapir" (predefined in the SUT) cannot be deleted
var_1	Wrong set of variable	Language in user preferences is always "English" and cannot be changed
var_2	Wrong set of variable	User defined columns in issue list cannot be copied between projects
var_3	Wrong set of variable	When adding new bug note, its status cannot be "private"
var_4	Wrong set of variable	Bug note view status cannot be changed

Table 4.3: Defects injected to the system under test for the Case Study 2 and 3

type of the Tapir Framework.

4.3.1 Method of Case Study

To provide an answer to research questions **Q1** and **Q5**, the Exploratory Testing process was decomposed to a set of sub-tasks (see Table 4.4) and the time spent by performing each sub-task was recorded by individual testers. Fourteen testers carried out the experiment on the selected SUT. The testing group was divided into two groups with seven testers in each to prevent a learning effect. The first group was instructed to test the SUT using the classical manual exploratory testing technique. The second group has been using the proposed solution to aid the ET process. background of the experiment participants in software testing was from 1.5 years to 8 years. The participants were split into the both groups in a way to keep the distribution of this praxis as equal as possible.

The group performing manual exploratory testing was provided with the Tapir Browser Extension only. Thus, this group was performing standard manual exploratory testing with no additional support. Only the SUT model was recorded in the background. The second group has been given the full support of the proposed solution.

The testing environment was set for each tester to keep their work isolated from each other. We have chosen Docker for its deployment simplicity and created a Docker container to host the SUT. Then, we have created a Docker container for the MantisBT application. The container with SUT was then distributed to the testers.

The testing experiment was conducted in three phases. The first phase was a simulation of the first round of exploratory test when the testers are not familiar with the SUT. The second and third phase were simulating two rounds of exploratory tests when the testers already knew the SUT and they were adding new test cases to cover more situations. For each of the phases, the time limit for testing was set to 4 hours. We let the testers to decide when they consider the SUT explored and tested. Data were collected from the experiment participant using the structured questionnaires.

To provide an answer to research question **Q2**, we used a defect injection technique. We modified used SUT by adding a set of configurable artificial defects. When turned on, the artificial defect demonstrated as an artificial malfunction of the SUT. These artificial defects changed processed data or displayed an error message on the screen, but they were not changing the proper flow of the functions and screens in the SUT. When the defect was reached (its code was activated) by the particular tester, this event was logged by the SUT to a special log created for this purpose. Besides that, the testers were reporting the defects in the separate defect tracker.

We prevented the learning effect by the following measure: The majority of the injected artificial defects activated in Phase 1 was deactivated in Phase 2 and this phase, a set of new artificial defects has been activated. The same we applied for the transition between

Phase 2 and Phase 3. By this, we simulated defect fixing and regression effect between the testing phases.

4.3.2 Case Study Results

Tables 4.4, 4.5 and 4.6 display the average times in minutes reported by experiment participants as times needed for completing the ET subtasks in each testing phase. The measured time is averaged for one subtask execution (e.g. average time needed for execution of a test case or average time needed for documentation of test case). Test cases in this experiment were defined as complete End–to–end test case exercising a principal business functionality of the SUT. Table 4.7 displays the average values for all three phases.

The collected data documents that the proposed solution was efficient in subtasks related to documentation of the path that has been explored by the testers and documentation of the parts which have to be explored in the future tests.

Minimal or no improvement was observed in the initial stage of testing - the testers in both groups were exploring the application with no structure prepared beforehand. In the later testing phases, support of the Tapir Framework.

In overall average, the exploratory testing process aided by the framework was 23.54% more resource efficient in terms of reported task execution times than the manual version of this process.

The results of the defect injection experiment are presented in Table 4.8. As the data show, there was a slight increase in an average number of detected defects for the ET aided by the Tapir Framework in comparison to the manually performed ET process.

4.3.3 Evaluation of the Results and Discussion

In this case study, two groups of testers were involved. One group was using manual ET process — only SUT model was reconstructed in the background of the process, but it was not visible for the testers. The second group was testing the SUT with the help of the proposed system.

The results of this study show that considerable effort was saved for the part of subtasks of the ET process when performed with the aid of an automated support (research question **Q1**). The savings have been achieved mostly in the subtasks related to the documentation of the test case (steps which have to be taken) and overall documentation of the explored parts of the SUT. Overall time savings in the case study were 23.54% when the proposed solution was used.

In the subtasks which were not directly supported by the proposed solution (e.g. specification of the test expected result, or defect report), no significant improvement has been achieved. These tasks had to be performed manually by the testers in both experimental groups.

Table 4.4: Time efficiency of manual ET process vs. the proposed approach – phase 1 – the first test

ET process subtask	Manual ET	Aided ET	Savings (%)
Execution of the test case	12.78	12.60	1.41
Documentation of test case: process flow	10.50	8.50	19.05
Documentation of test case: test input data	5.70	5.25	7.89
Documentation of test case: expected test result	7.65	7.42	3.01
Documentation of which part of SUT has been explored	4.75	0.00	100.00
Documentation of which part of SUT to be explored in the next phase	7.15	0.00	100.00
Report of possible defects	9.15	8.67	5.25
Revision of the test cases for next iteration of tests	8.60	7.60	11.63
Time spent by all subtasks	66.28	50.04	24.50

Table 4.5: Time efficiency of manual ET process vs. the proposed approach – phase 2 – the second test round

ET process subtask	Manual ET	Aided ET	Savings (%)
Execution of the test case	10.45	9.40	10.05
Documentation of test case: process flow	7.83	6.90	11.88
Documentation of test case: test input data	3.15	3.05	3.17
Documentation of test case: expected test result	7.40	6.80	8.11
Documentation of which part of SUT has been explored	2.80	0.00	100.00
Documentation of which part of SUT to be explored in the next phase	3.1	0.00	100.00
Report of possible defects	7.71	7.50	2.72
Revision of the test cases for next iteration of tests	7.07	4.85	31.40
Time spent by all subtasks	49.51	38.50	22.24

Table 4.6: Time efficiency of manual ET process vs. the proposed approach – phase 3 – the third test round

ET process subtask	Manual ET	Aided ET	Savings (%)
Execution of the test case	9.80	8.95	8.67
Documentation of test case: process flow	7.40	6.10	17.57
Documentation of test case: test input data	2.90	2.76	4.83
Documentation of test case: expected test result	6.18	5.67	8.25
Documentation of which part of SUT has been explored	3.8	0.00	100.00
Documentation of which part of SUT to be explored in the next phase	1.8	0.00	100.00
Report of possible defects	7.58	7.00	7.65
Revision of the test cases for next iteration of tests	6.45	4.61	28.53
Time spent by all subtasks	45.91	35.09	23.57

Table 4.7: Average time efficiency of manual ET process vs. the proposed approach

ET process subtask	Manual ET	Aided ET	Savings (%)
Execution of the test case	11.01	10.32	6.30
Documentation of test case: process flow	8.58	7.17	16.44
Documentation of test case: test input data	3.92	3.69	5.87
Documentation of test case: expected test result	7.08	6.63	6.31
Documentation of which part of SUT has been explored	3.78	0.00	100.00
Documentation of which part of SUT to be explored in the next phase	4.02	0.00	100.00
Report of possible defects	8.15	7.72	5.20
Revision of the test cases for next iteration of tests	7.37	5.69	22.88
Time spent by all subtasks	53.90	41.21	23.54

Table 4.8: Results of the defect injection experiment in Case Study 1

Exploratory testing phase	Phase 1	Phase 2	Phase 3
Previous artificial defects deactivated	0.0	16.0	12.0
New artificial defects activated	20.0	10.0	8.0
Total defects activated	20.0	14.0	10.0
Defects reached by manual ET testers (average)	17.1	11.2	9.2
Defects reported by manual ET testers (average)	16.1	10.7	9.0
Defects reached by aided ET testers (average)	17.8	12.0	9.8
Defects reported by aided ET testers (average)	17.2	11.4	9.3
Difference between manual and aided ET approaches for reached defects (%)	4.1	7.1	6.5
Difference between manual and aided ET approaches for reported defects (%)	6.8	6.5	3.3

No significant decrease of time efficiency has been observed as a consequence of Tapir Framework support (research question **Q5**).

Further on, there was a slight increase in an average number of detected defects for the aided exploratory testing in comparison to the manually performed ET process by 6% (research question **Q2**).

4.4 Case Study 2: Evaluation of the Tapir Framework Efficiency (Alternative Method)

To get more data about the efficiency of the Tapir Framework and also to verify the more recent version of the framework prototype, we conducted Case Study 2. The aim of this study is to answer research questions **Q2**, **Q3**, **Q4** and **Q5**. Differently to the Case Study 1, this study was aimed at the exploration of the SUT functions in a simulated smoke–tests, primarily interested in the extent of the SUT functions explored in a limited amount of time. The collection of the data was based on machine processing of the SUT models recorded during the experiment. Details are presented in the following subsections.

4.4.1 Method of Case Study

In this case study, ET process performed manually by individual testers was compared with ET process supported by the Tapir Framework. In this case study, we used the following method:

Group of 54 testers performed exploratory testing in the SUT, the MantisBT issue tracker with inserted artificial defects (see Table 4.3). Each of the testers acted individually. The testers were instructed to perform exploratory smoke-test and in this process, to explore the maximal extent of the SUT. Exit criteria were left to individual tester's consideration.

To evaluate the results of this case study, we used data which were available in the SUT model created by the Tapir Framework during the exploratory testing process (for details refer to Section 4.4.3). In difference to Case Study 1, We have not relied on subjective reports by individual testers.

1. Group of 23 testers performed Exploratory Testing process manually. The activity of these testers was recorded by a Tapir Framework tracking extension and the TapirHQ Back-End service. The TapirHQ Front–End application was not available to this group, so no navigational support was provided to its members.

2. Group of 31 testers, disjunctive to the previous one, performed the Exploratory Testing process with support provided by the Tapir Framework. This group used RANK_NEW navigational strategy. Within this strategy, one randomly selected half of this group used *PageComplexityRank* and the second half used *ElementTypeRank*. As test data strategy, DATA_NEW_RANDOM was used, but no equivalence classes were defined by the Team Lead and the testers in this group were explicitly instructed to do not rely on the framework suggestions in terms of test data and to be initiative in determining which test data to enter. This was done to make the conditions of the both groups as equal as possible (the group performing the ET manually

has not received any support regarding the test data). No priorities were set for SUT pages and its elements. The Test Lead was not changing any set–up during the experiment. Values of the $actionElementsWeight$, $inputElementsWeight$, and $linkElementsWeight$ constants were left to their default value 256.

Regarding the experimental groups, we ensured that all participants had received the equivalent initial training regarding software testing techniques: principle of Exploratory Testing, identification of boundary values and equivalence classes, combination of testing data to an input in SUT (condition, decision and condition/decision coverage, pairwise testing and basics of constraint interaction testing) and techniques to explore a SUT workflow (process cycle test). The participants were differing in praxis in software testing from 0,5 to 4 years. Participants were distributed randomly to the particular groups. This applies also to the experimental groups of Case Study 3 described further on.

Intentionally, in this Case Study we have not used team variants of the navigational strategies provided (principally RANK_NEW_TEAM). To perform an experiment in an objective way, equivalent team support shall be also present in case of ET performed manually. We have tried such an initial experiment, nevertheless, an equivalent simulation of the Tapir Framework functionality by a human team leader was very hard to achieve. Thus, we evaluate the team versions of the navigational strategies further on in the Case Study 3.

4.4.2 Metrics Used to Evaluate Case Studies 2 and 3

To evaluate the Case Studies 2 and 3, we used a set of metrics, which are summarized in Table 4.9. In this table, we define the metrics by elements of the SUT model presented in section 3.2. When referring to these metrics in the further text, we use their unified names and codes (Table 4.9, columns "Code" and "Metric name").

Code	Metric name	Definition of metric by SUT model elements
	Number of participants	$\|T\|$
PE	Average number of pages explored, pages can repeat	$\frac{\sum_{t \in T} \sum_{w \in W_t} visits(w)_t}{\|T\|}$
UPE	Average number of unique pages explored	$\frac{\sum_{t \in T} \|W_t\|}{\|T\|}$
RUP	Ratio of unique pages explored	$\frac{UPE}{PE} \cdot 100\%$

LE	Average number of links explored, elements can repeat	$\frac{\sum_{t \in T} \sum_{l \in L_t} visits(l)_t}{\lvert T \rvert}$
ULE	Average number of unique links explored	$\frac{\sum_{t \in T} \lvert L_t \rvert}{\lvert T \rvert}$
RUL	Ratio of unique links explored	$\frac{ULE}{LE} \cdot 100\%$
AE	Average number of action elements explored, elements can repeat	$\frac{\sum_{t \in T} \sum_{a \in A_t} visits(a)_t}{\lvert T \rvert}$
UAE	Average number of unique action elements explored	$\frac{\sum_{t \in T} \lvert A_t \rvert}{\lvert T \rvert}$
RUA	Ratio of unique action elements explored	$\frac{UAE}{AE} \cdot 100\%$
TP	Average time spent on page [*seconds*]	$\tau \Big/ \frac{\sum_{t \in T} \sum_{w \in W_t} visits(w)_t}{\lvert T \rvert}$
TUP	Average time spent on unique page [*seconds*]	$\tau \Big/ \frac{\sum_{t \in T} \lvert W_t \rvert}{\lvert T \rvert}$
TL	Average time spent on link element [*seconds*]	$\tau \Big/ \frac{\sum_{t \in T} \sum_{l \in L_t} visits(l)_t}{\lvert T \rvert}$
TUL	Average time spent on unique link element [*seconds*]	$\tau \Big/ \frac{\sum_{t \in T} \lvert L_t \rvert}{\lvert T \rvert}$
TA	Average time spent on action element [*seconds*]	$\tau \Big/ \frac{\sum_{t \in T} \sum_{a \in A_t} visits(a)_t}{\lvert T \rvert}$
TUA	Average time spent on unique action element [*seconds*]	$\tau \Big/ \frac{\sum_{t \in T} \lvert A_t \rvert}{\lvert T \rvert}$

Table 4.9: Metrics used to evaluate the Case Studies 2 and 3

In Table 4.9, τ stands for total time spent by exploratory testing activity, averaged for all testers in the group, given in seconds. In Table 4.9, we have not defined metrics for measurement of the efficiency of defect detection, as information about detected defects in not a part of the SUT model.

4.4.3 Case Study Results

Table 4.10 summarizes the comparison of manual exploratory testing approach with the Tapir Framework, based on the data which we were able to automatically collect from the recorded SUT model. In this overall comparison, the averages of particular results from used navigational strategy RANK_NEW_USER and ranking functions *ElementTypeRank* and *PageComplexityRank* are provided for the Tapir Framework and as test data strategy, DATA_NEW_RANDOM was used. The

$$DIFF = \frac{AUT - MAN}{AUT}$$

in percentage, where AUT stands for value measured in the case of the Tapir Framework manual and MAN stands for value measured in the case of the manual approach. In the statistics, we excluded excessive lengthy steps (tester spent more than 15 minutes on a particular page), caused by leaving the session opened and not testing actually.

Average times spent on the page are measured using the Tapir Framework logging mechanism. Average time to activate a defect is calculated as average total time spent by exploratory testing process divided by the number of activated defects. When a defect is activated, it is reached during the exploratory testing process, thus a tester can notice and report this defect.

Out of 19 inserted artificial defects, 3 were not activated by any of the testers in the group supported by the Tapir Framework, which is 15.8% ratio. In particular, it was defects *synt_6*, *synt_7* and *mc_2*. In the case of ET performed manually, one the defect *mc_2* was activated by one tester from the group.

The efficiency of manual exploratory testing approach compared to exploratory testing supported by the Tapir Framework in terms of potential to detect injected artificial defects in SUT is depicted in Figure 4.1.

For particular injected defects, we depict an average value how many times the defect has been activated by one tester. Value 1 would mean, that all the testers in the team have activated the defect once. For example, value 0,5 would mean, that 50% of the testers in the team has activated the defect once. The injected artificial defects have been introduced in Table 4.3.

Metric	Manual approach	Tapir used	DIFF
Number of participants	23	31	
UPE — Average number of pages explored, pages can repeat	151.8	197.9	23.3%
UPE — Average number of unique pages explored	22.2	37.7	41.0%
RUP — Ratio of unique pages explored	14.6%	19.0%	23.1%
LE — Average number of links explored, links can repeat	64.7	113.2	42.9%
ULE — Average number of unique links explored	21.4	44.0	51.3%
RUL — Ratio of unique links explored	33.1%	38.9%	14.8%
AE — Average number of action elements explored, elements can repeat	24.5	59.0	58.5%
UAE — Average number of unique action elements explored	9.6	28.3	66.2%
RUA — Ratio of unique action elements explored	39.1%	47.9%	18.5%
TP — Average time spent on page [*seconds*]	21.5	20.1	-6.6%
TUP — Average time spent on unique page [*seconds*]	146.7	105.7	-38.7%
TL — Average time spent on link element [*seconds*]	50.4	35.2	-43.2%
TUL — Average time spent on unique link element [*seconds*]	152.3	90.6	-68.1%
TA — Average time spent on action element [*seconds*]	133.1	67.5	-97.3%
TUA — Average time spent on unique action element [*seconds*]	340.7	140.8	-141.9%
Average activated defects logged (defects can repeat)	11.1	16.6	32.8%
Average unique activated defects logged	4.6	6.6	31.1%
Average time to activate 1 defect (when activated defects can repeat) [*seconds*]	292.8	240.5	-21.7%
Average time to activate 1 unique defect [*seconds*]	713.8	601.3	-18.7%

Table 4.10: Comparison of manual exploratory testing approach with the Tapir Framework: data from the SUT model

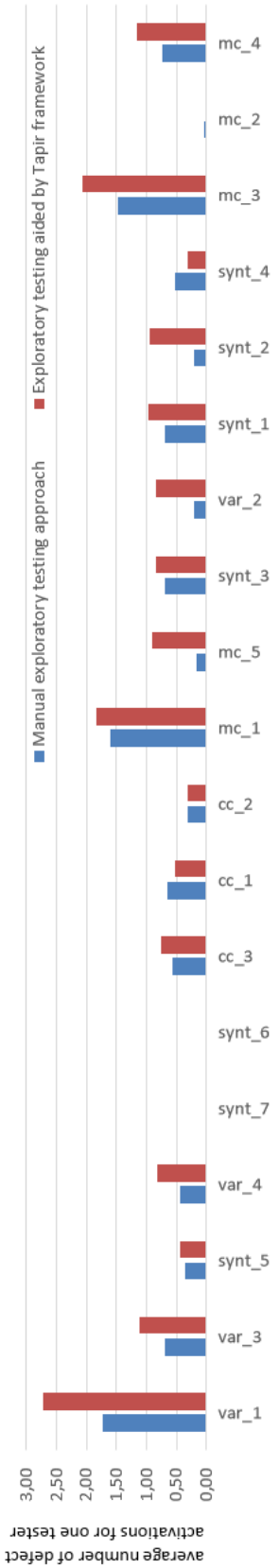


Figure 4.1: Potential of manual exploratory testing and the Tapir Framework approach to detect injected defects in the SUT

Regarding the average times spent on SUT page by testers using manual exploratory testing approach and testers using the Tapir Framework support, the details are provided in Figure 4.2.

Another comparison can be made for unique inserted defects activated during the activity of individual testers in the both groups. Details are presented in Figure 4.3.

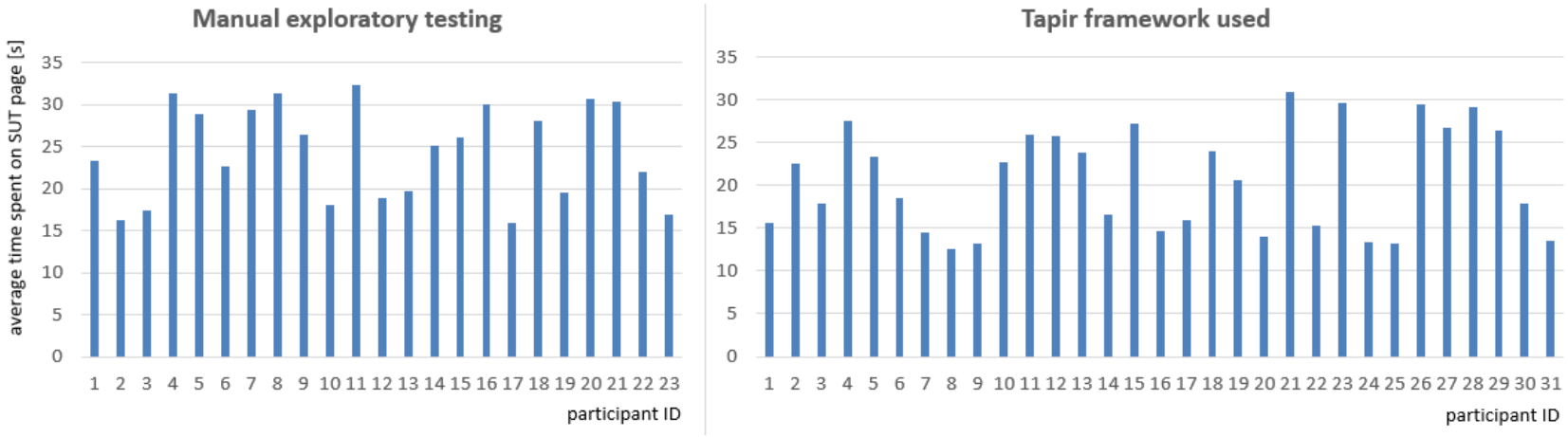


Figure 4.2: Average times spent on SUT pages by testers using manual approach and the Tapir Framework

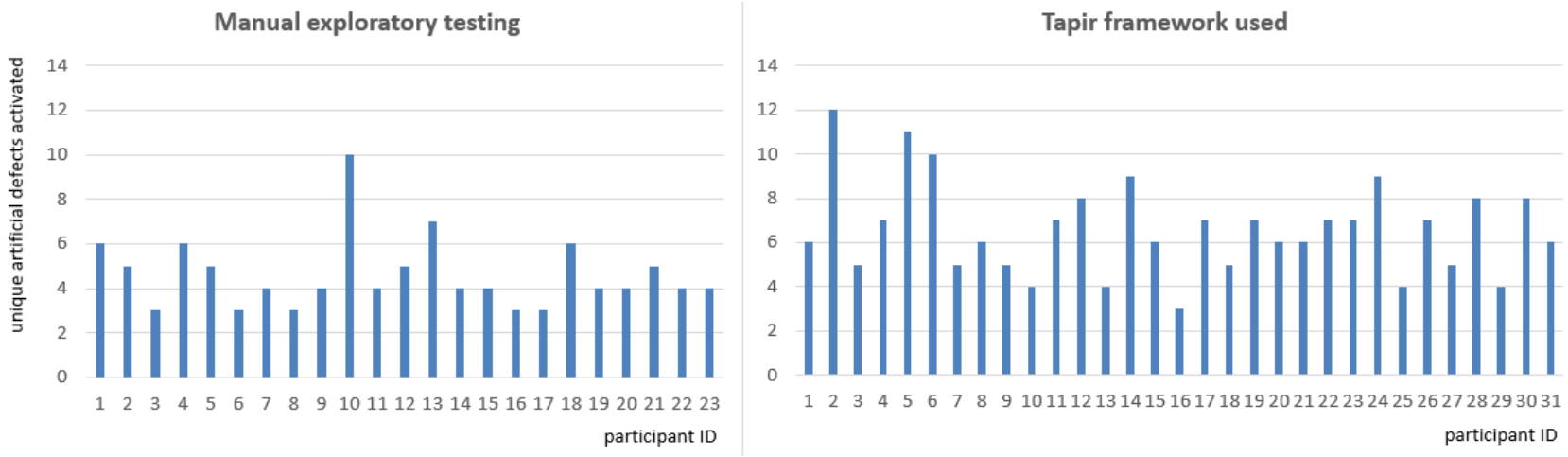


Figure 4.3: Unique inserted defects activated by testers using manual approach and the Tapir Framework

4.4.4 Evaluation of the Results and Discussion

To evaluate the case study, let's analyze the data in Table 4.10. Usage of the Tapir Framework lead the testers to explore larger extents of the SUT, compared to ET performed manually. This effect can be observed for average number of total explored and unique explored SUT pages (values *PE* and *UPE*), where Tapir support leads the testers to explore 23.3% pages in total and 41% more of unique SUT pages. For average number of explored total and unique link elements (values *LE* and *ULE*) and also for total and unique action elements of the pages (values *AE* and *UAE*) the differences are even higher. However, individual times spent by ET process differ, thus, the efficiency of the exploratory testing process aided by the framework has to be examined in more detail, analyzing the data in proper relations. Here, we can analyze three principal indicators:

(1) ratio of repetition of the pages and page elements during the testing process (research question **Q4**),

(2) extent of SUT explored per time unit (research question **Q3**) and

(3) defect detection potential (research question **Q2**).

(1) Ratio of repetition of the pages and page elements during the testing process could indicate an extent of possible unnecessary action done in the SUT during the ET process. In collected data (ref. to Table 4.10) we express this metric as the ratio of unique pages or elements exercised during the tests. Considering the average ratio of explored unique pages explored in the ET process (value *RUP*), for Tapir framework it improves by 4.4% (23.1% in relative difference *DIFF*). The average ratio of explored unique link elements (value *RUL*) improves by 5.8% (14.8% in relative difference*DIFF*) in case of the Tapir Framework support. In case of action elements (value *RUA*), we achieve the largest improvement of 8.8% (18.8% in relative difference *DIFF*). Also, these improvements are significant, but we need to discuss the relevance of these metrics.

When analyzing the data, the first–glance–conclusion is, that the ratio of unique pages is relatively low. For instance, when exploring the SUT in manual ET process, in average, each page was repeated 6.83 times to achieve a new page in the SUT. In the case of the Tapir Framework support, this number lowers to 5.25. Nevertheless, the sound explanation can be quickly provided when analyzing this situation. Pages repeat in the testing flows, as the aim of the process is to exercise action elements and link transitions. What is maybe more surprising is that also the links are repeated relatively frequently in the testing process.

For ET performed manually, participants exercised each link 3.02 times in average to explore one new unique link transition. In the case of the Tapir Framework support, this ratio lowered to 2.57. When imagining navigation in the SUT and repetition of its particular functions with various test data, also this fact matches the overall picture. The

same case is the average repetition of action elements, where each action has been repeated 2.56 times in case of manual ET process and 2.08 in the case of Tapir support.

Our conclusion is that we cannot base the evaluation of the efficiency of the Tapir Framework or ET process in general solely on the ratio of unique elements. Exercising the SUT with more combinations of test data could make these numbers lower and thus making an impression of the inefficiency of the testing process. The fact is, that using more test data combinations (and thus repeating the SUT elements more times during the exploration) can lead to the detection of more defects. Here, it strongly depends on the testing goals and principal types of defects we want to detect. If the testing goal is a rapid smoke test of the SUT, the ratio of unique pages or page elements can be the good indicator of process efficiency. If the goal is to detect more complex structural defects in the SUT, such a metric can be even contra productive. In any case, we need to analyze and consider also other indicators, which are following in this section.

(2) Extent of SUT explored per time unit indicates overall time efficiency when exercising the SUT with exploratory tests. Average time spent on page (value *TP*) improves by 6.6% in case of Tapir support, which can be explained by Tapir handling overhead connected to ET process (documentation of the path taken, decisions what to test next, documentation of the test data). As SUT pages were repeated quite often significance of this result is not major. As detailed data in Figure 4.2 show, there are differences between individual times spent on page by testers — still, this factor is strongly influenced by individual tester's attitude and the efficiency of work.

In individual time spent on SUT pages, we need to distinguish two factors, which contribute to the overall testing time:

(1) overhead related to ET process which, as result shows, is being decreased by the Tapir Framework, and,

(2) time essential to analyze the SUT page, notice and report the defects.

The second part shall remain the same in the both manual and aided exploratory testing. In the first factor, the machine support can lower the time spent on overhead activities. In the provided data, the both parts are mixed together (as it is practically impossible to distinguish these two parts when collecting the data based on monitoring of the events in the SUT user front–end).

Because we are rather interested in exploring SUT functions available on pages, more significant can be a discussion, how much time was needed in average to explore an SUT action element (value *TA*) or link (value *TL*). In the case of action elements, the time changes significantly by 97.3% in case of Tapir support, in the case of links, the difference is also significant (43.2%). The differences are even more striking in case of considering unique pages (value *TUP*, difference 38.7% in favor of the Tapir Framework), unique action

elements (value *TUA*, difference 141.9%) and unique link elements (value *TUL*, difference 68.1%).

From these figures, we can draw a conclusion, that the Tapir Framework leads to more efficient exploring of the SUT functions in relation the spent testing time. However, this optimism can diminish when we discuss the possible various goals of the testing process again. For a rapid smoke or exploratory lightweight testing of SUT, when the primary mission statement is to explore the new SUT parts rapidly and efficiently, it seems, that the Tapir can give promising support. Nevertheless, for more thorough testing, the validity of these metrics shall be revised, as more thorough tests and more extensive variants of test data are used. Thus, also results in this part shall be analyzed together with efficiency in terms of defect detection potential.

(3) Defect detection potential is an alternative to defect detection rate — in this metric we measure if the artificial defect has been activated in the code (which was ensured by the Tapir Framework logging mechanism). When a tester activates a defect, he/she is capable of noticing and reporting it later on. In the case of the Tapir Framework support, testers activated 32.8% more defects in average when we considered all activated defects, including repeating ones (a tester exercised the same functionality with an inserted defect more times) and 31.1% more defects in average when we consider unique defects only. Out of 19 inserted defects, 4.6 unique defects in average were defected in average in manual execution of ET, whereas with Tapir support this amount raised to 6.6. Still, this is approximately one–third of all inserted defects only, which we attribute to a rather difficult character of the inserted artificial defects.

Keeping in mind that the group using the Tapir support exercised the SUT longer than the group without his support, we are rather interested in the time needed to activate a defect. With the support of the Tapir Framework the average time to activate a defect (regardless if activated defects repeat), this time was lower by 21.7%. When we consider unique defects, the difference is 18.7% in favor of the Tapir Framework.

Statistics by individual inserted defects are presented in Figure 4.1. Details to the efficiency of individual participants are given in Figure 4.3. Also here, we see differences between individual testers in terms of their efficiency. When analyzing the data, we have not observed a direct correlation between time spent on page and number of defects which were detected by individual testers in the both groups.

The Case Study 2 provided data to answer research questions **Q2**, **Q3**, **Q4** and **Q5**. Compared to the manual approach, support of the Tapir Framework lead the testers to explore larger extents of the SUT and supports the testers to explore parts of the SUT unreached previously (research question **Q3**). This is also documented by average number of SUT pages explored per time unit, which rises slightly in case of SUT pages in favor of the Tapir Framework (6.6%), but this difference starts to be significant, when we consider unique pages (38.7%), total links (43.2%), unique links (68.1%), total action elements

(97.3%) and unique action elements (141.9%) (research question **Q4**). Nevertheless, we need to keep in mind, that these figures document only the capability of the framework to lead the testers to explore new parts of the SUT time–efficiently. No relation to the intensity of testing is expressed here: more thorough testing involves repetition of the SUT parts, thus, the reliability of this indicator shall be discussed in this case. On the other hand, measured defect detection potential also speaks in favor of the Tapir Framework: with the systematic support (and mainly because the testers were able to explore the larger extent of the SUT), the testers activated 31.1% more unique inserted defects in average. Regarding an average time efficiency to detect a defect, in the case of the Tapir Framework, this indicator was better by 18.7% for unique inserted defects (research question **Q2**).

Regarding the question **Q5**, no indicator documenting an aspect in which efficiency of the Exploratory Testing process supported by the Tapir Framework would decrease was found during the analysis of the results.

Practically, the Case Study 2 confirms the first results acquired by the Case Study 1. In both of the studies, we used different version of the Tapir Framework prototype and alternative method of data collection (in Case Study 1 we let the testers to measure the times spent on individual tasks and report them in prepared forms, in Case Study 2 we relied on automated data collection and gained the data from the recorded SUT model). Also, the experiment instructions have been formulated differently to the participants (complete Exploratory Testing cycle including documentation of the test cases versus smoke–test with primary aim to explore the large extent of the SUT) and the participants were exercising different parts of the SUT, which was triggered by different user roles in the system. However, the results demonstrate benefits of the Tapir Framework in both of the cases.

Group ID	Number of participants	Navigational strategy	Ranking function	Test data strategy
1	13	RANK_NEW_TEAM	*ElementTypeRank*	DATA_NEW_RANDOM_TEAM
2	11	RANK_NEW	*ElementTypeRank*	DATA_NEW_RANDOM
3	12	RANK_NEW_TEAM	*PageComplexityRank*	DATA_NEW_RANDOM_TEAM
4	12	RANK_NEW	*PageComplexityRank*	DATA_NEW_RANDOM

Table 4.11: Participant groups performing the Case Study 3

4.5 Case Study 3: Comparison of Navigational Strategies

In the Case Study 3, we focused to answer the research question **Q6**, related to selection of the most efficient navigational strategy. With independent groups of testers, we compared proposed navigational strategies focused primarily to explore new SUT functions. In this study, a group of 48 testers performed exploratory testing in MantisBT issue tracker with inserted artificial defects (see Table 4.3), all of them using the support of the Tapir Framework. The testers were instructed to explore the maximal extent of the SUT. This group was split to 4 subgroups specified in Table 4.11.

In this case study, no equivalence classes were defined by the Team Lead and the testers in this group were explicitly instructed to do not rely on the framework suggestions in terms of test data and to act actively in determining which test data to enter. No priorities were set for SUT pages and its elements. The Test Lead was not changing any set-up during the experiment. Values of the $actionElementsWeight$, $inputElementsWeight$, and $linkElementsWeight$ constants were left to their default value 256.

Regarding the strategies using prioritization of the page elements and pages (particularly PRIO_NEW and PRIO_NEW_TEAM), this concept adds extra opportunities to make the Exploratory Testing process more efficient. No comparable alternative in proposed navigational strategies is available at this stage of the Tapir Framework development. When equivalent prioritization is done in the manual Exploratory Testing process, we expect the same increase of testing process efficiency. For these reasons, we have decided to not include evaluation of navigational strategies PRIO_NEW and PRIO_NEW_TEAM in the described case study.

Regarding the test data strategies, each of the individual strategies is designed practically for different use case (ref. to Table 3.3). Comparison can be done between individual tester and team version of the strategies, e.g. DATA_NEW_RANDOM versus DATA_NEW_RANDOM_TEAM. As the presented case study primarily focuses on the efficiency of the

process of exploration of new SUT functions, comparison of the strategies DATA_NEW_RANDOM and DATA_NEW_RANDOM_TEAM were included in this Case Study.

4.5.1 Case Study Results

Table 4.12 presents comparison of the different Tapir Framework navigational strategies (ref. to Table 3.1) based on data automatically collected from the SUT model. In Table 4.12 the same metrics as in Table 4.10 are used.

Metric	Group 1	Group 2	Group 3	Group 4
Navigational strategy	RANK_NEW_TEAM	RANK_NEW	RANK_NEW_TEAM	RANK_NEW
Ranking function	*ElementType Rank*	*ElementType Rank*	*Page Complexity Rank*	*Page Complexity Rank*
Test data strategy	DATA_NEW_RANDOM_TEAM	DATA_NEW_RANDOM	DATA_NEW_RANDOM_TEAM	DATA_NEW_RANDOM
Number of participants	13	11	12	12
PE — Average number of pages explored, pages can repeat	224.0	211.7	233.6	206.2
UPE — Average number of unique pages explored	47.1	39.0	51.4	42.2
RUP — Ratio of unique pages explored	21.0%	18.4%	22.0%	20.5%
LE — Average number of links explored, elements can repeat	131.8	104.2	142.5	118
ULA — Average number of unique links explored	54.4	37.9	58.1	41.1
RUL — Ratio of unique links explored	41.3%	36.4%	40.8%	34.8%
AE — Average number of action elements explored, elements can repeat	69.3	57.5	75.1	62.7
UAE — Average number of unique action elements explored	34.8	24.6	38.3	29.6

RUA — Ratio of unique action elements explored	50.2%	42.8%	51.0%	47.2%
TP — Average time spent on page [*seconds*]	17.8	19.1	19.0	21.4
TUP — Average time spent on unique page [*seconds*]	84.6	103.6	86.3	104.7
TL — Average time spent on link element [*seconds*]	30.2	38.8	31.1	37.4
TUL — Average time spent on unique link element [*seconds*]	73.2	106.6	76.3	107.5
TA — Average time spent on action element [*seconds*]	57.5	70.3	59.0	70.4
TUA — Average time spent on unique action element [*seconds*]	114.5	164.3	115.8	149.2
Average activated defects logged (defects can repeat)	19.7	16.8	20.3	17.3
Average unique activated defects logged	8.6	6.9	9.1	7.4
Average time to activate 1 defect (when activated defects can repeat) [*seconds*]	202.2	240.6	218.4	255.3
Average time to activate 1 unique defect [*seconds*]	463.3	585.8	487.2	596.9

Table 4.12: Comparison of Tapir navigational strategies based data from SUT model

Relative differences between results of Case Study 2 groups are presented in Table 4.13.

Relative difference formula (α stands for a metric from Table 4.12)	$\frac{\alpha_{Group1}-\alpha_{Group2}}{\alpha_{Group1}}$	$\frac{\alpha_{Group3}-\alpha_{Group4}}{\alpha_{Group4}}$	$\frac{\alpha_{Group2}-\alpha_{Group4}}{\alpha_{Group2}}$	$\frac{\alpha_{Group1}-\alpha_{Group3}}{\alpha_{Group1}}$

Comment	RANK_NEW vs. RANK_NEW_TEAM for *ElementTypeRank*	RANK_NEW vs. RANK_NEW_TEAM for *PageComplexityRank*	*ElementTypeRank* vs. *PageComplexityRank* for RANK_NEW	*ElementTypeRank* vs. *PageComplexityRank* for RANK_NEW_TEAM
PE — Average number of pages explored, pages can repeat	5.5%	11.7%	2.6%	-4.1%
UPE — Average number of unique pages explored	17.2%	17.9%	-8.2%	-8.4%
RUP — Ratio of unique pages explored	12.4%	7.0%	-11.1%	-4.4%
LE — Average number of links explored, elements can repeat	20.9%	17.2%	-13.2%	-7.5%
ULE — Average number of unique links explored	30.3%	29.3%	-8.4%	-6.4%
RUL — Ratio of unique links explored	11.9%	14.6%	4.2%	1.2%
AE — Average number of action elements explored, elements can repeat	17.0%	16.5%	-9.0%	-7.7%
UAE — Average number of unique action elements explored	29.3%	22.7%	-20.3%	-9.1%
RUA — Ratio of unique action elements explored	14.8%	7.4%	-10.3%	-1.5%
TP — Average time spent on page [*seconds*]	-7.4%	-12.9%	-12.2%	-6.3%
TUP — Average time spent on unique page [*seconds*]	-22.5%	-21.3%	-1.0%	-1.9%
TL — Average time spent on link element [*seconds*]	-28.3%	-20.3%	3.5%	-2.8%

TUL — Average time spent on unique link element [*seconds*]	-45.6%	-40.8%	-0.8%	-4.0%
TA — Average time spent on action element [*seconds*]	-22.3%	-19.3%	-0.2%	-2.6%
TUA — Average time spent on unique action element [*seconds*]	-43.5%	-28.9%	9.2%	-1.1%
Average activated defects logged (defects can repeat)	14.7%	14.8%	-3.0%	-3.0%
Average unique activated defects logged	19.8%	18.7%	-7.2%	-5.5%
Average time to activate 1 defect (when activated defects can repeat) [*seconds*]	-19.0%	-16.9%	-6.1%	-7.4%
Average time to activate 1 unique defect [*seconds*]	-26.5%	-22.5%	-1.9%	-4.9%

Table 4.13: Relative differences between results of Case Study 3 groups

4.5.2 Evaluation of the Results and Discussion

In the Case Study 3, we compared the efficiency of the individual strategies provided by the framework with aim to answer the research question **Q6**. In our analysis of the data, we will refer to Table 4.13. Let's start with **comparison of individual testing strategy RANK_NEW with team strategy RANK_NEW_TEAM** (columns $\frac{\alpha_{Group1}-\alpha_{Group2}}{\alpha_{Group1}} \cdot 100\%$ and $\frac{\alpha_{Group3}-\alpha_{Group4}}{\alpha_{Group4}} \cdot 100\%$). Results differ by ranking function used (*ElementTypeRank* vs. *PageComplexityRank*), but still, there are general trends, which can be observed in the data. Team navigational strategy RANK_NEW_TEAM increased the ratio of unique explored pages (value *RUP*) by 12.4% for *ElementTypeRank*, resp. by 7.0% for *PageComplexityRank*.

In case of ratio of unique link elements (value *RUL*), improvement is 11.9% for *ElementTypeRank* and 14.6% for *PageComplexityRank*. Similar trend can be observed for unique action elements (value *RUA*), where the improvement is 14.8% for *ElementTypeRank* and 7.4% for *PageComplexityRank* used as a ranking function.

More relevant can be statistics related to time efficiency of the testing process: here, the RANK_NEW_TEAM performs also better. Average time spent on unique page (value *TUP*) was decreased by 22.5% for *ElementTypeRank* and 21.3% for *PageComplexityR-*

ank. Average time spent on unique link element (value *TUL*) was decreased by 45,6% for *ElementTypeRank* and 40.8% for *PageComplexityRank*. Average time spent on unique action element (value *TUA*) was decreased by 43.5% for *ElementTypeRank* and 28.9% for *PageComplexityRank*. These numbers are speaking in favour of team strategy again.

Regarding the average unique activated defects logged, RANK_NEW_TEAM raises the result by 19.8% for *ElementTypeRank* and 18.7% for *PageComplexityRank*. Also average time to detect one unique defect drops by 26.5% in case of *ElementTypeRank* and by 22.5% in case of *PageComplexityRank*. These results correspond to explored extent of SUT functions, which is higher in case of RANK_NEW_TEAM navigational strategy.

From these relative differences, it might seem, that *ElementTypeRank* performs better and in case of team version of the navigational strategy, it gains better improvement. That would not be the right conclusion; to assess the **efficiency of *ElementTypeRank* and *PageComplexityRank***, we need to analyze the data independently.

Generally, *PageComplexityRank* leads to exploration of slightly larger extent of SUT (Table 4.13). The ratio of unique pages explored is 11.1% higher for RANK_NEW and 4.4% higher for RANK_NEW_TEAM navigational strategy. Also, the ratio of unique forms explored is 10.3% higher for RANK_NEW navigational strategy.

Regarding the average times spent on SUT page, link and action elements, the only significant difference is average time spent on page (value *TP*), which is 12.2% in favour of *ElementTypeRank* in the case of RANK_NEW and 6.3% in case of RANK_NEW_TEAM navigational strategy. *PageComplexityRank* leads to the exploration of more complex pages, which takes more time in the process, resulting in higher time spent on SUT page. As the more complex pages usually aggregate more unexplored links and action elements, the extent of explored SUT parts is higher than in the case of *ElementTypeRank*, as already discussed above.

What is interesting, *PageComplexityRank* leads to exploration of more action elements than *ElementTypeRank* (Table 4.12, value *UAE*). The explanation lays in the navigational strategy algorithm: with *PageComplexityRank*, the Tapir Framework mechanism scans the possible following pages and prefers the more complex pages (usually containing more action elements to explore).

PageComplexityRank also slightly increased the number of average unique activated defects by 7.2% for RANK_NEW and 5.5% for RANK_NEW_TEAM strategies. In average time to detect these defects, no significant difference was found.

From all analyzed data, the most efficient combination of navigational strategy and ranking function are RANK_NEW_TEAM with *PageComplexityRank* regarding the number of activated inserted defects and extent of the SUT explored, but when we consider time efficiency to explore the new SUT functions, RANK_NEW_TEAM with *ElementTypeRank* seems to be a better candidate.

This case study provided data to answer research questions **Q6** to identify the most

efficient navigational strategy. Regarding the comparison of RANK_NEW and RANK_NEW_TEAM navigational strategies, the team navigational strategy performs better in all measured aspects. For ranking function *PageComplexityRank* average ratio of unique explored pages increased by 7.0%, average ratio of unique link elements by 14,6% and the average ratio of unique action elements by 7.4%. Average time spent on unique page improved by 21.3%, average time spent on unique link, resp. action elements improved by 40.8%, resp. 28.9%. Also, total unique activated defects logged improved by 18.7% and average time to detect one unique defect improved by 22.5%. Analyzed separately, ranking function *PageComplexityRank* performs better than *ElementTypeRank* in terms of extent of explored SUT functions, and, thus, slightly higher number of activated unique defects.

No clear favorite is obvious from the combination of navigational strategy RANK_NEW_TEAM with ranking functions *ElementTypeRank* and *PageComplexityRank*. RANK_NEW_TEAM with *PageComplexityRank* performed slightly better in terms of extent of explored SUT and detected defects, on the contrary RANK_NEW_TEAM with *ElementTypeRank* was slightly more time–efficient.

4.6 Case Study 4: Applicability of the Tapir Framework to Various SUTs

The goal of Case Study 4 was to answer the research question **Q7** related to applicability of the Tapir Framework to different types of SUT, especially focusing on an SUT with strongly dynamically generated HTML pages. For this case study, we selected three another system under test: **JTrac** with HTML front-end pages generated dynamically and **OFBiz** and **Moodle** with front-end pages with more fixed elements and their identificators, including the URL format of the SUT pages.

4.6.1 JTrac

As a part of the experiments, we have tried to use the Tapir Framework with JTrac[2]. system as SUT. JTrac is an open–source highly customizable issue tracker written in Java (a sample screen is presented in Fig. 4.5). The code base is not as large as the code base of MantisBT, but in terms of system functions, JTrac provides a number of functions related to issue tracking processes (full–text search, customizable fields, advanced filters, etc.). The main reason for selection of JTrac, was a dynamic way, by which the SUT front-end HTML pages are generated, including the URL of the individual pages of the SUT.

Unfortunately, the application showed to be unsuitable for use with the Tapir Framework. The JTrac application is built using component oriented Java web application framework Wicket[3], which has a consequence: URLs of pages in this system are generated dynamically.

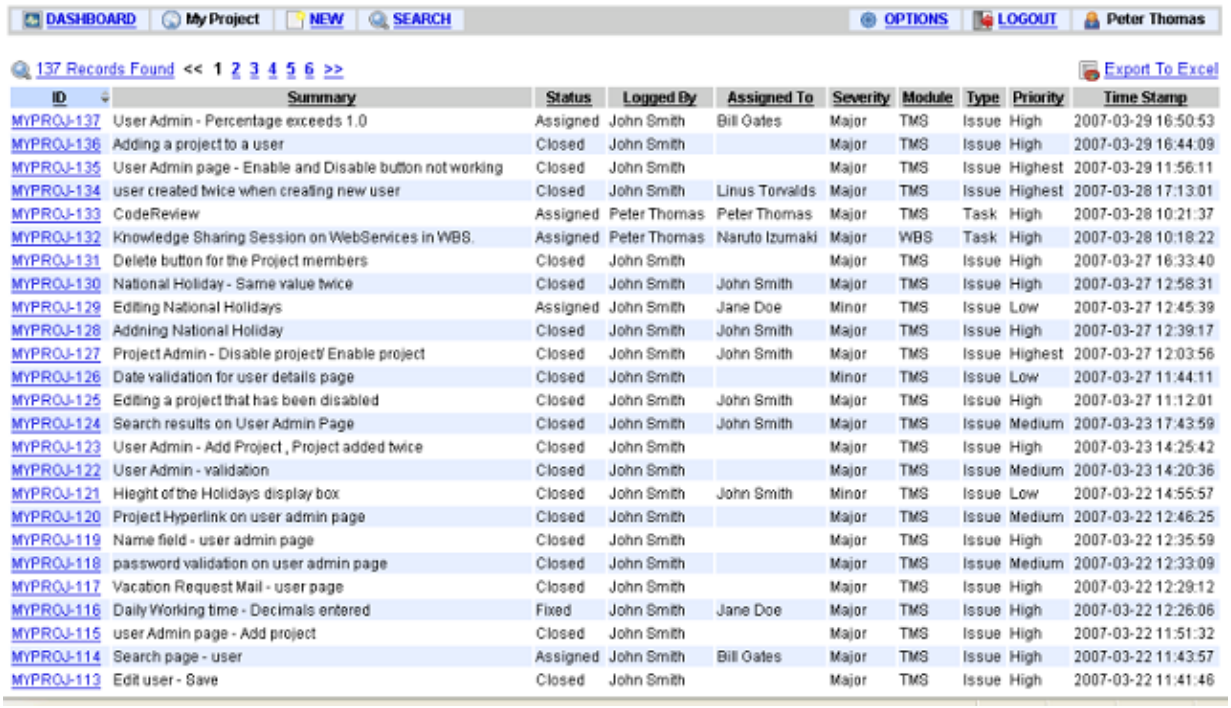
As mentioned, Tapir tracking extension matches the current page the user is viewing with the node of the model of the SUT using the page URL. The JTrac pages are changing their URLs with every post–back or redirect making it problematic to match to the corresponding page node in the SUT model. In particular, this URL format was `http://.../jtrac/app/?wicket:interface=:<incrementally changing number>::::`.

As a result, the Tapir Framework was running partially, but the model reconstruction was inaccurate. The conclusion from this experiment can be generalized — the SUTs with dynamically generated URLs, which change with every post–back or redirect call are not suitable to use with the Tapir Framework.

4.6.2 OFBiz

Apache OFBiz[4], which was the next subject of experimental application of the Tapir Framework, is an open–source product for the automation of enterprise processes that includes

[2] http://jtrac.info/
[3] https://wicket.apache.org/
[4] https://ofbiz.apache.org/



DASHBOARD | My Project | NEW | SEARCH | OPTIONS | LOGOUT | Peter Thomas

137 Records Found << 1 2 3 4 5 6 >> Export To Excel

ID	Summary	Status	Logged By	Assigned To	Severity	Module	Type	Priority	Time Stamp
MYPROJ-137	User Admin - Percentage exceeds 1.0	Assigned	John Smith	Bill Gates	Major	TMS	Issue	High	2007-03-29 16:50:53
MYPROJ-136	Adding a project to a user	Closed	John Smith		Major	TMS	Issue	High	2007-03-29 16:44:09
MYPROJ-135	User Admin page - Enable and Disable button not working	Closed	John Smith		Major	TMS	Issue	Highest	2007-03-29 11:56:11
MYPROJ-134	user created twice when creating new user	Closed	John Smith	Linus Torvalds	Major	TMS	Issue	Highest	2007-03-28 17:13:01
MYPROJ-133	CodeReview	Assigned	Peter Thomas	Peter Thomas	Major	TMS	Task	High	2007-03-28 10:21:37
MYPROJ-132	Knowledge Sharing Session on WebServices in WBS.	Assigned	Peter Thomas	Naruto Izumaki	Major	WBS	Task	High	2007-03-28 10:18:22
MYPROJ-131	Delete button for the Project members	Closed	John Smith		Major	TMS	Issue	High	2007-03-27 16:33:40
MYPROJ-130	National Holiday - Same value twice	Closed	John Smith	John Smith	Major	TMS	Issue	High	2007-03-27 12:58:31
MYPROJ-129	Editing National Holidays	Assigned	John Smith	Jane Doe	Minor	TMS	Issue	Low	2007-03-27 12:45:39
MYPROJ-128	Addning National Holiday	Closed	John Smith	John Smith	Major	TMS	Issue	High	2007-03-27 12:39:17
MYPROJ-127	Project Admin - Disable project/ Enable project	Closed	John Smith	John Smith	Major	TMS	Issue	Highest	2007-03-27 12:03:56
MYPROJ-126	Date validation for user details page	Closed	John Smith		Minor	TMS	Issue	Low	2007-03-27 11:44:11
MYPROJ-125	Editing a project that has been disabled	Closed	John Smith	John Smith	Major	TMS	Issue	High	2007-03-27 11:12:01
MYPROJ-124	Search results on User Admin Page	Closed	John Smith	John Smith	Major	TMS	Issue	Medium	2007-03-23 17:43:59
MYPROJ-123	User Admin - Add Project , Project added twice	Closed	John Smith		Major	TMS	Issue	High	2007-03-23 14:25:42
MYPROJ-122	User Admin - validation	Closed	John Smith		Major	TMS	Issue	Medium	2007-03-23 14:20:36
MYPROJ-121	Hieght of the Holidays display box	Closed	John Smith	John Smith	Minor	TMS	Issue	Low	2007-03-22 14:55:57
MYPROJ-120	Project Hyperlink on user admin page	Closed	John Smith		Major	TMS	Issue	Medium	2007-03-22 12:46:25
MYPROJ-119	Name field - user admin page	Closed	John Smith		Major	TMS	Issue	High	2007-03-22 12:35:59
MYPROJ-118	password validation on user admin page	Closed	John Smith		Major	TMS	Issue	Medium	2007-03-22 12:33:09
MYPROJ-117	Vacation Request Mail - user page	Closed	John Smith		Major	TMS	Issue	High	2007-03-22 12:29:12
MYPROJ-116	Daily Working time - Decimals entered	Fixed	John Smith	Jane Doe	Major	TMS	Issue	High	2007-03-22 12:26:06
MYPROJ-115	user Admin page - Add project	Closed	John Smith		Major	TMS	Issue	High	2007-03-22 11:51:32
MYPROJ-114	Search page - user	Assigned	John Smith	Bill Gates	Major	TMS	Issue	High	2007-03-22 11:43:57
MYPROJ-113	Edit user - Save	Closed	John Smith		Major	TMS	Issue	High	2007-03-22 11:41:46

Figure 4.4: A sample of the JTrac application — list of issues

framework components and business applications for ERP (Enterprise Resource Planning), CRM (Customer Relationship Management), E–Business/E–Commerce, SCM (Supply Chain Management), MRP (Manufacturing Resource Planning), MMS/EAM (Maintenance Management System/Enterprise Asset Management).

The HTML pages of the OFBiz website are well structured, which made it easier to create the page analyzer easier. Although the forms on this web site can be overflowing with elements, still the structure is logical and consistent across the web pages — the naming and identification of elements, application of CSS classes to rows in tables displaying data. This was not the case of MantisBT where similar patterns were implemented in different ways and the UI was inconsistent in a number of parts of this system. What was challenging during the adoption of the Tapir Tracking Extension for the OFBiz web pages was the usage of JavaScript those pages that are used to improve the UX. For example, the input elements for date values are enriched with date–picker controls — this resulted in a more detailed inspection of the created model of the page and adjusting the analyzer accordingly.

Despite these technical obstacles, the adoption of Tapir Tracking Extension to this SUT was successful. The framework was running without defects and reconstructed SUT model was accurate.

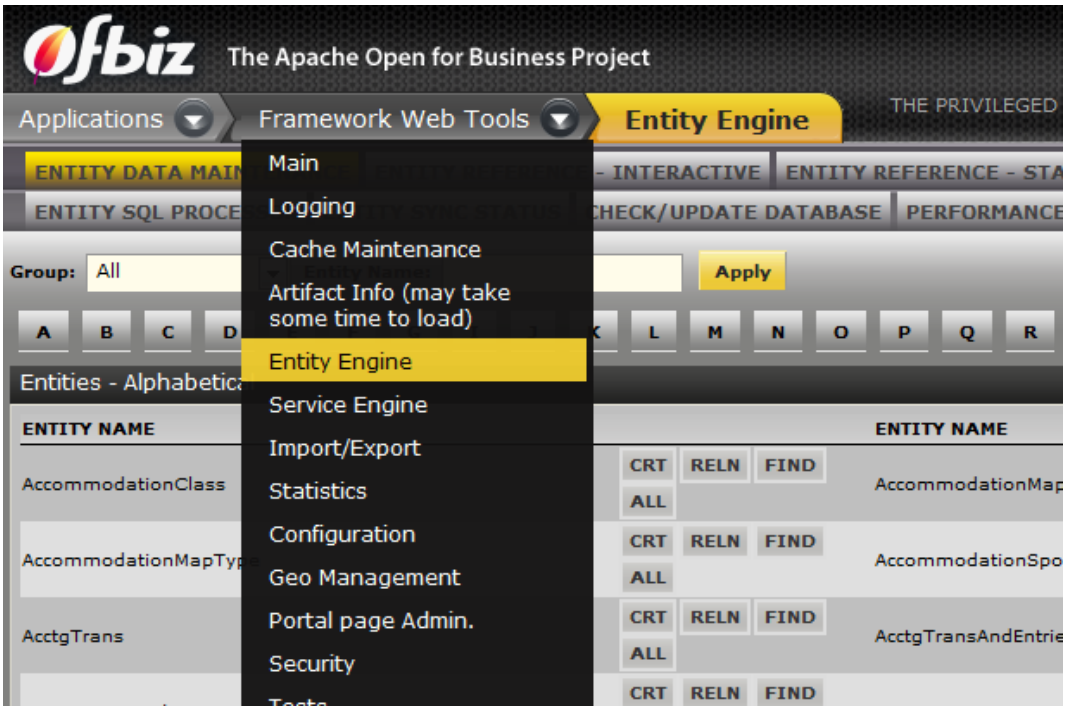


Figure 4.5: A sample of the OFBiz application — system dashboard

4.6.3 Moodle

Moodle[5] is an open–source, on–line Learning Management system enabling educators to create their own private website filled with dynamic courses that extend learning[6]. The application has a number of customizable management features (create a course with lessons, upload learning material add quiz questions, badges, ...) and widgets (student progress monitor, event calendar, ...) to help the educators achieve their goal. The web pages of Moodle are well structured. The structure of the page–master page relationship is well defined with reasonable hierarchy.

The naming of HTML elements, as well as application of CSS classes, is consistent making the identification of the elements to create the model of the SUT straightforward. This was not the case of MantisBT and the code of the analyser had to be adapted. The application uses modern YUI[7] library to create the UI, similar library was not used by neither of the previously tested applications and the analyser had to be adapted accordingly. The YUI library, for example, uses HTML `<i>` elements with icons inside HTML `<a>` elements as action elements.

The Tapir Framework functionalities have been verified during a set of functional tests

[5]Modular Object–Oriented Dynamic Learning Environment
[6]https://moodle.org/
[7]Yahoo User Interface library, https://yuilibrary.com/

also with this SUT with positive results – the SUT model was being reconstructed correctly and the generation of navigational test cases was running by the specification.

4.7 Threats to Validity

In performed case studies, we tried to set the conditions of the compared experimental groups as equal as possible and, principally, to compare only comparable alternatives, when fixing other conditions the same for all the participant groups. Nevertheless, several concerns can be raised regarding the validity of the data, which we discuss in this section.

Regarding the Case Study 1, during the experiments, both groups of the testers were recording the duration of the individual steps during the testing. This recording probably prolonged the measured the duration of the subtasks. Also, when measured, it is probable, that testers have been doing their individual tasks more precisely, compared to an unstructured or non-controlled way. This also could have an impact on measured times. Nevertheless, the measurement of spent time was performed for the both groups of the testers, so its impact should be equal for the both groups.

Next concern could be raised about the influence effect of testers‘ skills, experience and general gift for exploratory testing for on the individual results. We have minimised this effect by dividing the testers into the groups based on their level of experience and seniority to keep the groups more or less equal. Nevertheless, this fact must be still considered.

Another point can be raised regarding the method of defect injection. The artificial defects were not changing the proper flow of the functions and screens in the SUT. When implemented so, the possibility to explore the SUT would be limited. This can be considered as a limit of the experiment. Nevertheless, when analysing the situation, such defect injection would impact the both groups (and their results) equally. Thus, our conclusion is that we can accept this limitation.

Considering the impact of the learning effect in each group, this effect shall be minimised, as the both groups have been learning the explored application concurrently. Only the exploratory testing process method (purely manual or aided by the proposed solution) was different. Still, there is an issue deserving discussion discuss: six of the testers were already familiar with the MantisBT application, which gave them an advantage in the exploratory testing. However, we minimised this effect by:

1. Defect injection as described above. The testers were not informed about particular parts of the SUT, where the injected defects were activated during the testing.

2. Equal distribution of these testers in the both experimental groups.

Regarding the Case Studies 2 and 3, we did a discussion of the relevance of used metrics already in section 4.5.2. We hope this discussion was conducted in an objective way and for each of the principal metrics proper disclaimers and possible limiting conditions were described.

In the Case Studies 2 and 3 we used defect activation concept instead of defect detection. The defect activation expresses a chance of the tester even to notice the defect, when

executed. The Tapir Framework logging mechanism exactly logged the fact that a defect is activated, so the collected data are accurate at this point. In praxis, we can expect real defect reporting ratio lower, as some of the activated defects will not be noticed and reported by testers. Nevertheless, as this metric was used in all comparisons, our opinion is, it can be utilized for measurement of a trend alternatively to defect detection ratio (which can be, on the other hand, biased by individual flaws in defect reporting by experimental team members).

Idle times of tester can influence measured times during the session (this is a likely scenario when analyzing measured data, see graph in Figure 4.2). Unfortunately, in the experiments, we had not a better option how to measure times spent during the testing process: initially we experimented with subjective tester's report of time spent on individual testing tasks, but this proved as less reliable method than automated collection of timestamps related to tester's actions in the GUI, which was used in the case studies. We tried to minimise this problem by excluding excessively long steps (tester spent more than 15 minutes on a particular page), caused by leaving the session opened and not testing actually.

In Case Study 2 (comparison of manual exploratory testing approach with the Tapir Framework), the group using the Tapir Framework is larger (31 versus 23 in the group performing ET without support). Nevertheless the size of the group was large enough to mitigate this risk; moreover, all testers acted individually, so team synergy did not play a role in this case study and all analysed data were averages for a particular group.

Previous experience in exploratory testing of the experiment participants and a natural gift of an individual to perform this type of testing efficiently can differ among the participants, and this fact can be raised as another issue. Nevertheless, in the experimental group, there was no tester expertly specialised to exploratory testing, which could favorise one of the groups. Moreover, case study participants were distributed to the groups randomly, which shall mitigate this issue.

Regarding the size of the used SUT, Mantis BT tracker workflow sand screenflow model is extensive enough to draw conclusions regarding research questions Q1–Q6. The used version of the mantis BT (1.2.19) composes of 202964 lines of code, 938 application files, the database schema has 31 database tables.

Finally, a concern can be raised about the selection of the SUTs for all of the case studies: already existing open–source SUTs were used for experiments, which potentially do not reflect specifics of real software development project. During the experiments, we tried to find such a project, nevertheless, in the experimental phase of the Tapir Framework, we have not managed to secure such an experiment in the competitive software development environment. However, when considering the purpose of the case studies, this shall not have an effect which could significantly invalidate the results.

In Case Studies 1–3, we simulated a defective behaviour of the SUT by a defect injection

technique. In Case Study 1 we simulated a process of evolution of the SUT, including defect fixing and regression by the phases of the experiment.

In Case Study 4 we tested the applicability of the Tapir Framework to different SUTs from the viewpoint of technical feasibility to connect the Tapir Browser Extension to the SUT front–end. In this point, the technical style of the front–end structure and implementation details were the major influencers.

4.8 Other Applications of the Tapir Framework

The Tapir Framework allows to monitor software testers during their job and record the explored path and entered testing data in the model. For these properties, the framework can be used alternatively for situations, in which we are interested in measuring efficiency of testing process. In this section, we give two examples of these situations.

4.8.1 Monitoring of Testers to Evaluate Efficiency of Static Testing

Static testing is an efficient method detecting software defects in a phase, where the defect fixing is rather inexpensive when compared to the later project phases. It can detect design errors or inconsistencies related to handling of business data objects of the SUT (e.g. missing functions, wrong assignment of SUT functions to the business data objects, suboptimal design of particular business data objects) and proper static testing can lead to the design of more consistent and efficient dynamic test cases designed to verify consistency of data objects in tested system.

Various concepts and methods exist in this area. Authors of [21] focused on static testing related to consistency of business data objects in the Enterprise Information Systems (EIS). Usually, data–flow based techniques apply to data consistency in EIS. On the conceptual level, the Data Cycle Test (DCyT) [58, 43] is considered as a template for the data consistency tests. The DCyT bases on a concept of CRUD matrix[8] and proposes fundamental methods of static testing using such CRUD matrices.

To investigate the effectiveness of the static testing in this context, an experiment was conducted simulating a situation in which an incomplete and inconsistent test basis was used as an input to the creation of DCyT test cases. Issue tracking system MantisBT was used as experimental SUT. We let several groups of test designers create DCyT test cases, part of them were using the proposed static testing, part of them not. Next, we evaluated the DCyT test cases produced by the experiment participants.

For this evaluation, we used the Tapir Framework, connected to MantisBT. By this, we had available SUT model, defined by the same elements as the test basis model given to the experiment participants. The Tapir Framework reconstructed the workflow model based

[8] http://www.tmap.net/wiki/crud

on the high–level states (pages of the SUT) and transitions between them (SUT functions). The exact workflow model of this SUT was therefore available for further analysis of the data consistency test cases produced during the experiment.

As a part of an evaluation process, artificial data consistency defects were added to the SUT model in the Tapir Framework to evaluate the potential of the DCyT test cases to detect these defects.

Experiment participants were using predefined Microsoft Excel template to record the produced test cases. This allowed automated processing of the data. The test cases were loaded to the Tapir Framework connected to MantisBT defect tracker of the same version as we used for the experiment (1.2.19).

The results of this research have been published in the Cluster Computing journal [A.2].

4.8.2 Evaluation of Test Coverage

Another possible use case of the Tapir Framework is to let the framework monitor the testers performing tests by a set of test cases prepared before the test execution and evaluate a coverage of these test cases. Principally two types of useful information can be acquired by this process:

(1) Parts of the SUT or transitions between SUT pages not exercised by any tests. If any business important functionality will be present in these parts, more tests shall be added to the analysed test set.

(2) Parts of SUT repeatedly executed during the tests. Repetition of particular function in the executed tests does not necessarily indicate an inefficient duplicity, as the function can be repeated with various combinations of test data. Nevertheless, when we analyze a set of smoke tests or priority regression tests, a repetition of the same scenarios can be an indication for further optimisation of the test set.

Such an analysis can be a useful complement to coverage analysis based on the test basis (requirements coverage for instance). As passive monitoring of the testers by the tracking extension and automated reconstruction of the SUT model is performed automatically and does not add an overhead to the testing process, the cost of this analysis is only an evaluation of the recorded model and its comparison to the used test cases.

Chapter 5

Conclusions

In this chapter, we summarise the results of this Dissertation Thesis, its contributions and we discuss the future work on the Tapir project, which is planned for the next two years horizon.

5.1 Summary

The Exploratory Testing technique represents a sound testing alternative for software development projects, in which test basis (design documentation used to design the test cases) is not available, or is significantly obsolete or inconsistent. A number of issues can influence the efficiency of this technique. When testers actions in the SUT are not systematically documented during the testing process, it usually leads to repetitive test cases, including repetitive data combinations. In such situations, it is generally difficult to assess, whether particular SUT feature was already tested. This makes decisions what to test in the next steps difficult, especially, for more junior members of the testing team. Also, this situation usually requires a strong managerial presence of a Test Lead, which has to organize the testing in an efficient manner. Also, problems when reporting defects can be experienced, as testers often do not remember the explored path exactly. This impacts the quality of defect reports, which adds additional overhead to the development and testing part of the project (more information is required by the development team to reproduce these defects). Besides the extra costs, this effect also prolongs the defect fixing by the developers. Consequently, the quality of testing strongly depends on the experience and skills of the testers.

With the aim to minimise these problems, we designed and experimentally implemented a framework for automated support of the Exploratory Testing process. This framework is designed for the systems under tests to web–based applications and information systems, providing an HTML–based user interface. Browser tracking extension records actions in

the SUT front–end performed by the testers and based on this information, the SUT model is created and dynamically updated during testers' exploration of the SUT. During this exploration, navigational test cases are automatically created and presented to the testers by guideline application running side-by-side with the SUT.

The SUT model includes pages of the SUT front–end, input, link and action elements of these pages, as well as structures for management of test data used in the exploratory testing process. History of SUT exploration by individual testers is also recorded in the model.

To evaluate the efficiency and applicability of the proposed solution, we conducted four Case Studies, which we presented in this Dissertation Thesis.

The first two Case Studies were performed to compare the efficiency of exploratory testing supported by the proposed framework with exploratory testing performed manually without such support. The results of these Case Studies are promising: In the Case Study 1, significant effort was saved for the part of subtasks of the ET process when performed with the aid of an automated support. The savings were achieved mostly in the subtasks related to the documentation of the test case (steps which have to be taken) and overall documentation of the explored parts of the SUT. Overall time savings in the case study were 23.54% when the proposed solution was used. In the subtasks which were not directly supported by the used version of Tapir Framework (e.g. specification of the test expected result, or defect report), no significant improvement was achieved. Further on, there was a slight increase in an average number of detected defects for the aided exploratory testing in comparison to the manually performed ET process by 6%.

As the Case Study 2 documented, ET supported by the Tapir Framework lead to exploration of larger extents of the SUT, and, together with that, lead the testers to explore more parts of the SUT unreached previously. This effect was also confirmed by the measured average number of SUT pages explored per time unit, which was better by 6.6% in case of the Tapir Framework. In the case of other elements of the SUT model, this metric improved significantly, in particular, 38.7% for unique pages, 43.2% for total links, 68.1% for unique links, 97.3% for total action elements and even 141.9% for unique action elements. These metrics document capability of the framework to lead the testers to explore new parts of the SUT time–efficiently. However, these metrics are the most relevant in the case of rapid light–weight exploratory testing, in which the testers' goal is to explore new SUT functions time–efficiently and to cover the most previously unexplored functions as possible. In the case of more thorough testing, the relevance of these metrics shall be revised. By principle, SUT pages and elements would repeat the tests when exercised more times by more extensive input test data combinations. Nevertheless, another metrics also document benefits of the Tapir Framework: with its support, testers were able to reach and activate 31.1% more inserted artificial defects in average. Regarding the average time needed to activate one unique defect, this indicator improved by 18.7%.

In the Case Study 3, we compared selected navigational strategies and ranking functions to select the best option to be used in the framework.

This comparison clearly documented, that team–based navigational strategies are more efficient than individual navigational strategies. For *PageComplexityRank* ranking function used in the comparison of the strategies, the average ratio of unique explored pages increased by 7.0% for team navigational strategy. Further on, the average ratio of unique link elements by 14.6% and the ratio of unique action elements by 7.4%. Average time spent on unique page improved by 21.3%, average time spent on a unique link, resp. an action elements, improved by 40.8%, resp. 28.9%. Also, total unique activated defects logged improved by 18.7% and average time to detect one unique defect improved by 22.5%.

Regarding the ranking functions, *PageComplexityRank* performed slightly better than *ElementTypeRank* in several aspects. Nevertheless, we are currently doing more experiments to find an optimal ranking function.

Finally, the Case Study 4 explored applicability of the Tapir Framework to different styles of SUT front–end coding. In the case of OFBiz and Moodle systems, customization of Browser Extension was entirely feasible, and the Tapir Framework was working without technical limitations. Our concern was an application with strongly dynamically generated front–end pages, where only a little stable elements and element IDs are present. As an example of this application, we selected JTrac. As customization of browser tracking extension shown, dynamically generated URL of the front–end pages represented a problem to reconstruct the SUT model; framework was running partially, but the model reconstruction was inaccurate. This represents an application limit of the proposed framework. It is worth to mention, that for an automated testing based on the front–end, such dynamic generation of HTML content also represents a significant obstacle. Such dynamically–generated applications are considered as anti–pattern by test automation community. JTrac was a difficult example in this case; in another experiment, we performed recently [20], preparation of Selenium scripts for JTrac has shown as a technical challenge, for the very similar reasons it was for adoption of tracking extension for the Tapir Framework.

Results of manual ET process can be influenced by a person of Test Lead. Even for a manual process, a systematic approach can be effective. Nevertheless, the systematic guidance of the testers including proper documentation of explored SUT parts is demanding task and shall be performed all time during the testing process. Thus, the proposed Tapir Framework can provide efficient support taking over this administrative overhead and letting Test Lead focus on the analysis of the state, strategic decisions during the testing process and motivation of the testing team more intensely. Effect of such support would be even stronger in the case of extensive SUTs and exploratory software testing in business domains the testers are not entirely familiar with. The results from the presented Case Studies documented viability of the proposed concept and motivate us to develop the framework further.

5.2 Contributions of the Dissertation Thesis

The contributions of this Dissertation Thesis can be summarised as the following:

- Design of the framework, which contributes to conduction of Exploratory Testing process in more efficient way regarding spent resources, extent of explored SUT and found defects.
- Combination of Reverse–Engineering, Model–Based Testing and Exploratory Testing, which we consider innovative (during extensive literature study on related topic, we have not identified a research or software industrial project, which addressed the problem in the way similar to our approach).
- Formal model of the underlying system under test, which can be (apart from being used as a basis for the Tapir Framework processes) further used for modeling of Exploratory Testing process or for measurement of efficiency of software testing process in general.
- Design of the initial navigational strategies, ranking functions and test data strategies, which can be further explored and more efficient variants can be found. This includes the focus on the team work and efficient usage of the individual tester's resources.
- Practical applicability of the proposed framework to industrial software development and testing projects.
- Possible alternative usages of the Tapir Framework, as measurement of the testing process efficiency or assessment of the efficiency of particular set of test cases.

5.3 Future Work

As case studies demonstrated, the concept of the presented Tapir Framework is promising in terms of its industrial application. Our further research plans include the further evolution of the framework. The development road–map of the Tapir Framework for the following two–year horizon covers the following areas:

- Improve usability aspects of the Tapir HQ application, guiding the exploratory tester through the SUT.
- Create Tapir Browser Extension for the Firefox browser, covering the browser with the second market share, and thus, to cover 90% of the browser market share (by the W3C browser usage statistic up to July 2017[1]).

[1] https://www.w3schools.com/Browsers/default.asp

- If technically feasible, also focus on the testers actions, which are being performed in the web browser environment only (without requesting the server for the load of the next SUT page). This would allow us to give higher granularity to the recorded tests.
- Find more efficient navigational strategies, ranking functions and test data strategies.
- Let the system collect more immediate feedback from the Exploratory Testing process and let it adapt the navigational process dynamically to changing conditions during the process (for instance by adjusting weights in the ranking functions in a feedback loop).
- Explore further possibilities of connecting of the framework to a Combinational Interaction Testing (CIT) module for generation of efficient data combinations to be entered in page inputs (typically various forms on the SUT pages).
- Implement the semi–automatic model update. The Browser Extension should compare the structure of the currently explored page with the model of that page. The difference between the page and its model might be caused by (1) an error or (2) a change made to the system. The operator will tell the Tapir Framework how to deal with this change and it will update the model in the case of a wanted change. The testers will be guided to test these parts of the SUT first.

The further development of the Tapir Framework will also be partially driven by feedback from industrial applications. We plan to conduct more case studies providing us with the feedback to improve the framework further on. Currently, we are in negotiations with Czech branches of two international companies, who expressed interest in participating in proof of concept of the Tapir Framework application in their software testing processes.

Bibliography

[1] M. Albert, J. Cabot, C. Gòmez, and V. Pelechano. Automatic generation of basic behavior schemas from uml class diagrams. *Software & Systems Modeling*, 9(1):47–67, 2010. ISSN 1619-1366. doi: 10.1007/s10270-008-0108-x. URL `http://dx.doi.org/10.1007/s10270-008-0108-x`.

[2] D. Amalfitano, A. R. Fasolino, and P. Tramontana. A gui crawling-based technique for android mobile application testing. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 252–261. IEEE, 2011.

[3] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon. Mobiguitar: Automated model-based testing of mobile apps. *Software, IEEE*, 32(5): 53–59, 2015.

[4] S. Arora and M. P. Rao. Probabilistic model checking of incomplete models. In *International Symposium on Leveraging Applications of Formal Methods*, pages 62–76. Springer, 2016.

[5] A. Bandyopadhyay and S. Ghosh. Test input generation using uml sequence and state machines models. In *Software Testing Verification and Validation, 2009. ICST '09. International Conference on*, pages 121–130, April 2009. doi: 10.1109/ICST.2009.23.

[6] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5): 507–525, May 2015. ISSN 0098-5589. doi: 10.1109/TSE.2014.2372785.

[7] F. Bellucci, G. Ghiani, F. Paternò, and C. Porta. Automatic reverse engineering of interactive dynamic web applications to support adaptation across platforms. In *Proceedings of the 2012 ACM international conference on Intelligent User Interfaces*, pages 217–226. ACM, 2012.

[8] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen. *Systems and software verification: model-checking techniques and tools*. Springer Science & Business Media, 2013.

[9] K. Bhatti and A. N. Ghazi. Effectiveness of exploratory testing, an empirical scrutiny of the challenges and factors affecting the defect detection efficiency, 2010.

[10] L. Bozzelli, A. Molinari, A. Montanari, A. Peron, and P. Sala. Interval temporal logic model checking: the border between good and bad hs fragments. In *International Joint Conference on Automated Reasoning*, pages 389–405. Springer, 2016.

[11] M. Brambilla and P. Fraternali. *Interaction Flow Modeling Language: Model-Driven UI Engineering of Web and Mobile Apps with IFML*. Morgan Kaufmann, 2014.

[12] M. Brambilla and P. Fraternali. Large-scale model-driven engineering of web user interaction: The webml and webratio experience. *Science of Computer Programming*, 89:71–87, 2014.

[13] M. Brambilla, P. Fraternali, et al. The interaction flow modeling language (ifml). Technical report, version 1.0. Technical report, Object Management Group (OMG), http://www.ifml.org, 2014.

[14] M. Brambilla, A. Mauri, and E. Umuhoza. Extending the interaction flow modeling language (ifml) for model driven development of mobile applications front end. In *Mobile Web Information Systems*, volume 8640 of *Lecture Notes in Computer Science*, pages 176–191. Springer International Publishing, 2014. ISBN 978-3-319-10358-7. doi: 10.1007/978-3-319-10359-4_15. URL `http://dx.doi.org/10.1007/978-3-319-10359-4_15`.

[15] R. Bruni and U. Montanari. Temporal logic and the μ-calculus. In *Models of Computation*, pages 271–286. Springer, 2017.

[16] K. Bubna and S. K. Chakrabarti. Act (abstract to concrete tests)-a tool for generating concrete test cases from formal specification of web applications. In *ModSym+ SAAAS@ ISEC*, pages 16–22, 2016.

[17] M. Bures. Automated testing in the czech republic: The current situation and issues. In *ACM International Conference Proceeding Series*, volume 883, pages 294–301, 2014. doi: 10.1145/2659532.2659605.

[18] M. Bures. Metrics for automated testability of web applications. In *ACM International Conference Proceeding Series*, volume 1008, pages 83–89, 2015. doi: 10.1145/2812428.2812458.

[19] M. Bures. Framework for assessment of web application automated testability. In *Proceedings of the 2015 Conference on research in adaptive and convergent systems*, pages 512–514. ACM, 2015.

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

[21] M. Bures and T. Cerny. *Static Testing Using Different Types of CRUD Matrices*, pages 594–602. Springer Singapore, Singapore, 2017. ISBN 978-981-10-4154-9. doi: 10.1007/978-981-10-4154-9_68. URL `https://doi.org/10.1007/978-981-10-4154-9_68`.

[22] S. Ceri, P. Fraternali, and A. Bongio. Web modeling language (WebML): a modeling language for designing web sites. *Computer Networks*, 33(1-6):137–157, jun 2000. doi: 10.1016/s1389-1286(00)00040-2. URL `https://doi.org/10.1016/s1389-1286(00)00040-2`.

[23] C. Ş. Gebizli and H. Sözer. Improving models for model-based testing based on exploratory testing. In *2014 IEEE 38th International Computer Software and Applications Conference Workshops*, pages 656–661, July 2014. doi: 10.1109/COMPSACW.2014.110.

[24] C. Ş. Gebizli and H. Sözer. Impact of education and experience level on the effectiveness of exploratory testing: An industrial case study. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 23–28, March 2017. doi: 10.1109/ICSTW.2017.8.

[25] V. Dallmeier, M. Burger, T. Orth, and A. Zeller. *WebMate: Generating Test Cases for Web 2.0*, pages 55–69. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-35702-2. doi: 10.1007/978-3-642-35702-2_5. URL `http://dx.doi.org/10.1007/978-3-642-35702-2_5`.

[26] R. Degiovanni, P. Ponzio, N. Aguirre, and M. Frias. *Abstraction Based Automated Test Generation from Formal Tabular Requirements Specifications*, pages 84–101. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-21768-5. doi: 10.1007/978-3-642-21768-5_8. URL `http://dx.doi.org/10.1007/978-3-642-21768-5_8`.

[27] A. Delgado, D. Calegari, and A. Arrigoni. Towards a generic bpms user portal definition for the execution of business processes. *Electronic Notes in Theoretical Computer Science*, 329:39–59, 2016.

[28] J. den Haan. Model Driven Engineering tools compared on user activities. http://www.theenterprisearchitect.eu/archive/2009/02/18/model-driven-engineering-tools-compared-on-user-activities, May 2017.

[29] A. Deutsch, L. Sui, and V. Vianu. Specification and verification of data-driven web applications. *Journal of Computer and System Sciences*, 73(3):442–474, 2007.

[30] G. A. Di Lucca and A. R. Fasolino. *Web Application Testing*, pages 219–260. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. ISBN 978-3-540-28218-1. doi: 10.1007/3-540-28218-1_7. URL http://dx.doi.org/10.1007/3-540-28218-1_7.

[31] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos. A survey on model-based testing approaches: a systematic review. In *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*, pages 31–36. ACM, 2007.

[32] L. H. do Nascimento and P. D. Machado. An experimental evaluation of approaches to feature testing in the mobile phone applications domain. In *Workshop on Domain specific approaches to software test automation: in conjunction with the 6th ESEC/FSE joint meeting*, pages 27–33. ACM, 2007.

[33] A. Domingues, E. M. Rodrigues, and M. Bernardino. Autofun. In *Proceedings of the 1st Brazilian Symposium on Systematic and Automated Software Testing - SAST*. ACM Press, 2016. doi: 10.1145/2993288.2993298. URL https://doi.org/10.1145/2993288.2993298.

[34] B. Dorninger, J. Pichler, and A. Kern. Using static analysis for knowledge extraction from industrial user interfaces. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 497–500, Sept 2015. doi: 10.1109/ICSM.2015.7332501.

[35] S. Eldh, H. Hansson, S. Punnekkat, A. Pettersson, and D. Sundmark. A framework for comparing efficiency, effectiveness and applicability of software testing techniques. In *Testing: Academic and Industrial Conference - Practice And Research Techniques, 2006. TAIC PART 2006. Proceedings*, pages 159–170, Aug 2006. doi: 10.1109/TAICPART.2006.1.

[36] E. P. Enoiu, A. Čaušević, T. J. Ostrand, E. J. Weyuker, D. Sundmark, and P. Pettersson. Automated test generation using model checking: an industrial evaluation. *International Journal on Software Tools for Technology Transfer*, 18(3):335–353, 2016.

[37] V. Entin, M. Winder, B. Zhang, and S. Christmann. Combining model-based and capture-replay testing techniques of graphical user interfaces: An industrial approach. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 572–577, March 2011. doi: 10.1109/ICSTW.2011.13.

[38] F. Ferri. *Visual Languages for Interactive Computing: Definitions and Formalizations*. Premier reference source. Information Science Reference, 2008. ISBN 9781599045368. URL `https://books.google.co.uk/books?id=LNOSq-q7wfoC`.

[39] M. Frappier, B. Fraikin, R. Chossart, R. Chane-Yack-Fa, and M. Ouenzar. *Comparison of Model Checking Tools for Information Systems*, pages 581–596. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-16901-4. doi: 10.1007/978-3-642-16901-4_38. URL `https://doi.org/10.1007/978-3-642-16901-4_38`.

[40] P. Fraternali and M. Tisi. Multi-level Tests for Model Driven Web Applications. In B. Benatallah, F. Casati, G. Kappel, and G. Rossi, editors, *Web Engineering*, volume 6189 of *Lecture Notes in Computer Science*, pages 158–172. Springer Berlin / Heidelberg, 2010. ISBN 978-3-642-13910-9.

[41] A. N. Ghazi, R. P. Garigapati, and K. Petersen. *Checklists to Support Test Charter Design in Exploratory Testing*, pages 251–258. Springer International Publishing, Cham, 2017. ISBN 978-3-319-57633-6. doi: 10.1007/978-3-319-57633-6_17. URL `https://doi.org/10.1007/978-3-319-57633-6_17`.

[42] M. Gogolla, F. Hilken, P. Niemann, and R. Wille. Formulating model verification tasks prover-independently as uml diagrams. In *European Conference on Modelling Foundations and Applications*, pages 232–247. Springer, 2017.

[43] D.-J. d. Grood. *TestGoal: Result-Driven Testing*. Springer Publishing Company, Incorporated, 1 edition, 2008. ISBN 354078828X, 9783540788287.

[44] M. Hammoudi, G. Rothermel, and P. Tonella. Why do record/replay tests of web applications break? In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 180–190, April 2016. doi: 10.1109/ICST.2016.16.

[45] E. Hendrickson. *Explore it!: reduce risk and increase confidence with exploratory testing*. The Pragmatic Programmers, 2014.

[46] B. Homès. *Fundamentals of Software Testing*. John Wiley & Sons, 2013.

[47] J. Hutchinson, J. Whittle, and M. Rouncefield. Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure. *Science of Computer Programming*, 89:144–161, 2014.

[48] J. Itkonen and M. V. Mäntylä. Are test cases needed? replicated comparison between exploratory and test-case-based software testing. *Empirical Software Engineering*, 19 (2):303–342, 2014. ISSN 1573-7616. doi: 10.1007/s10664-013-9266-8. URL `http://dx.doi.org/10.1007/s10664-013-9266-8`.

[49] J. Itkonen and K. Rautiainen. Exploratory testing: a multiple case study. In *Empirical Software Engineering, 2005. 2005 International Symposium on*, pages 10–pp. IEEE, 2005.

[50] J. Itkonen, M. V. Mäntylä, and C. Lassenius. Defect detection efficiency: Test case based vs. exploratory testing. In *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, ESEM '07, pages 61–70, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2886-4. doi: 10.1109/ESEM.2007.38. URL `http://dx.doi.org/10.1109/ESEM.2007.38`.

[51] J. Itkonen et al. Empirical studies on exploratory software testing. 2011.

[52] A. K. Jena, S. K. Swain, and D. P. Mohapatra. A novel approach for test case generation from uml activity diagram. In *2014 International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT)*, pages 621–629, Feb 2014. doi: 10.1109/ICICICT.2014.6781352.

[53] P. C. Jorgensen. *The Craft of Model-Based Testing*. CRC Press, 2017.

[54] C. Kaner, J. Bach, and B. Pettichord. *Lessons learned in software testing*. John Wiley & Sons, 2008.

[55] S. Kansomkeat, P. Thiket, and J. Offutt. Generating test cases from uml activity diagrams using the condition-classification tree method. In *Software Technology and Engineering (ICSTE), 2010 2nd International Conference on*, volume 1, pages V1–62–V1–66, Oct 2010. doi: 10.1109/ICSTE.2010.5608913.

[56] M. Kessentini, H. Sahraoui, and M. Boukadoum. Example-based model-transformation testing. *Automated Software Engineering*, 18(2):199–224, 2011. ISSN 1573-7535. doi: 10.1007/s10515-010-0079-3. URL `http://dx.doi.org/10.1007/s10515-010-0079-3`.

[57] D.-K. Kim and L.-S. Lee. Reverse engineering from exploratory testing to specification-based testing. *International Journal of Software Engineering and Its Applications*, 8(11):197–208, 2014.

[58] T. Koomen, L. v. d. Aalst, B. Broekman, and M. Vroon. *TMap Next, for Result-driven Testing*. UTN Publishers, 2013.

[59] B. Kumar and K. Singh. Testing uml designs using class, sequence and activity diagrams. *International Journal for Innovative Research in Science and Technology*, 2(3):71–81, 2015.

[60] D. Kundu, D. Samanta, and R. Mall. Automatic code generation from unified modelling language sequence diagrams. *Software, IET*, 7(1):12–28, February 2013. ISSN 1751-8806. doi: 10.1049/iet-sen.2011.0080.

[61] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella. Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 272–281, Oct 2013. doi: 10.1109/WCRE.2013.6671302.

[62] N. Li and J. Offutt. Test oracle strategies for model-based testing. *IEEE Transactions on Software Engineering*, 43(4):372–395, April 2017. ISSN 0098-5589. doi: 10.1109/TSE.2016.2597136.

[63] R. Lipka, T. Potuzak, P. Brada, P. Hnetynka, and J. Vinarek. A method for semi–automated generation of test scenarios based on use cases. In *Software Engineering and Advanced Applications (SEAA), 2015 41st Euromicro Conference on*, pages 241–244. IEEE, 2015.

[64] C. H. Liu, C. Y. Lu, S. J. Cheng, K. Y. Chang, Y. C. Hsiao, and W. M. Chu. Capture-replay testing for android applications. In *Computer, Consumer and Control (IS3C), 2014 International Symposium on*, pages 1129–1132, June 2014. doi: 10.1109/IS3C.2014.293.

[65] M. Lochau, S. Peldszus, M. Kowal, and I. Schaefer. Model-based testing. In *Advanced Lectures of the 14th International School on Formal Methods for Executable Software Models - Volume 8483*, pages 310–342, New York, NY, USA, 2014. Springer-Verlag New York, Inc. ISBN 978-3-319-07316-3. doi: 10.1007/978-3-319-07317-0_8. URL `http://dx.doi.org/10.1007/978-3-319-07317-0_8`.

[66] A. Lomuscio, H. Qu, and F. Raimondi. Mcmas: An open-source model checker for the verification of multi-agent systems. *International Journal on Software Tools for Technology Transfer*, 19(1):9–30, 2017.

[67] A. Mehmood and D. N. Jawawi. Aspect-oriented model-driven code generation: A systematic mapping study. *Information and Software Technology*, 55(2):395 – 411, 2013. ISSN 0950-5849. doi: http://dx.doi.org/10.1016/j.infsof.2012.09.003. URL

http://www.sciencedirect.com/science/article/pii/S0950584912001863. Special Section: Component-Based Software Engineering (CBSE), 2011.

[68] A. M. Memon. Automatically repairing event sequence-based gui test suites for regression testing. *ACM Trans. Softw. Eng. Methodol.*, 18(2):4:1–4:36, Nov. 2008. ISSN 1049-331X. doi: 10.1145/1416563.1416564. URL http://doi.acm.org/10.1145/1416563.1416564.

[69] L. Meng. Self-description and regeneration of test cases based on error-guessing method for safety-critical software automatic testing. *Journal of Tongji University (Nature Science)*, 32(8), 2004.

[70] A. Mesbah, A. Van Deursen, and S. Lenselink. Crawling ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web (TWEB)*, 6(1):3, 2012.

[71] M. Micallef, C. Porter, and A. Borg. Do exploratory testers need formal training? an investigation using hci techniques. In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 305–314, April 2016. doi: 10.1109/ICSTW.2016.31.

[72] J. W. Mickens, J. Elson, and J. Howell. Mugshot: Deterministic capture and replay for javascript applications. In *NSDI*, volume 10, pages 159–174, 2010.

[73] R. M. L. M. Moreira, A. C. R. Paiva, and A. Memon. A pattern-based approach for gui modeling and testing. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pages 288–297, Nov 2013. doi: 10.1109/ISSRE.2013.6698881.

[74] N. Moreno, P. Fraternali, and A. Vallecillo. Webml modelling in uml. *Software, IET*, 1(3):67–80, June 2007. ISSN 1751-8806.

[75] I. C. Morgado and A. C. R. Paiva. The impact tool: Testing ui patterns on mobile applications. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 876–881, Nov 2015. doi: 10.1109/ASE.2015.96.

[76] I. C. Morgado and A. C. R. Paiva. Testing approach for mobile applications through reverse engineering of ui patterns. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, pages 42–49, Nov 2015. doi: 10.1109/ASEW.2015.11.

[77] M. Nabuco and A. C. Paiva. Model-based test case generation for web applications. In *Proceedings of the 14th International Conference on Computational Science and Its Applications — ICCSA 2014 - Volume 8584*, pages 248–262, New York, NY,

USA, 2014. Springer-Verlag New York, Inc. ISBN 978-3-319-09152-5. doi: 10.1007/978-3-319-09153-2_19. URL http://dx.doi.org/10.1007/978-3-319-09153-2_19.

[78] T. Neil. *Mobile design pattern gallery: UI patterns for smartphone apps.* " O'Reilly Media, Inc.", 2014.

[79] B. N. Nguyen, B. Robbins, I. Banerjee, and A. Memon. Guitar: an innovative tool for automated testing of gui-driven software. *Automated Software Engineering*, 21(1):65–105, Mar 2014. ISSN 1573-7535. doi: 10.1007/s10515-013-0128-9. URL https://doi.org/10.1007/s10515-013-0128-9.

[80] R. F. Paige, S. Kokaly, B. Cheng, F. Bordeleau, H. Storrle, J. Whittle, and S. Abrahao. User experience for model-driven engineering: Challenges and future directions. In *ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems*. Institute of Electrical and Electronics Engineers Inc., 2017.

[81] A. C. Paiva, J. C. Faria, and P. M. Mendes. Reverse engineered formal models for gui testing. In *International Workshop on Formal Methods for Industrial Critical Systems*, pages 218–233. Springer, 2007.

[82] B. Peischl, M. Weiglhofer, and F. Wotawa. Executing abstract test cases. In *GI Jahrestagung*, 2007.

[83] D. Pfahl, H. Yin, M. V. Mäntylä, and J. Münch. How is exploratory testing used? a state-of-the-practice survey. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, page 5. ACM, 2014.

[84] J. Polpong and S. Kansomkeat. Syntax-based test case generation for web application. In *2015 International Conference on Computer, Communications, and Control Technology (I4CT)*, pages 389–393, April 2015. doi: 10.1109/I4CT.2015.7219604.

[85] T. Potuzak and R. Lipka. Interface-based semi-automated generation of scenarios for simulation testing of software components. In *SIMUL 2014*, pages 35–42. IARIA, 2014. ISBN 978-1-61208-371-1.

[86] P. Raappana, S. Saukkoriipi, I. Tervonen, and M. V. Mntyl. The effect of team exploratory testing – experience report from f-secure. In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 295–304, April 2016. doi: 10.1109/ICSTW.2016.13.

[87] D. Rafi, K. Moses, K. Petersen, and M. Mntyl. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. pages 36–42, 2012. doi: 10.1109/IWAST.2012.6228988. URL https:

//www.scopus.com/inward/record.uri?eid=2-s2.0-84864258235&doi=10.1109%2fIWAST.2012.6228988&partnerID=40&md5=594002cd294acc00723ef672b330211b.

[88] E. M. Rodrigues, R. S. Saad, F. M. Oliveira, L. T. Costa, M. Bernardino, and A. F. Zorzo. Evaluating capture and replay and model-based performance testing tools. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement - ESEM'14*. ACM Press, 2014. doi: 10.1145/2652524.2652587. URL https://doi.org/10.1145/2652524.2652587.

[89] R. Rodriguez-Echeverria, V. M. Pavón, F. Macías, J. M. Conejero, P. J. Clemente, and F. Sánchez-Figueroa. Ifml-based model-driven front-end modernization. 2014.

[90] G. Rossi. Web modeling languages strike back. *IEEE Internet Computing*, 17(4): 4–6, July 2013. ISSN 1089-7801. doi: 10.1109/MIC.2013.78.

[91] C. Sacramento and A. C. Paiva. Web application model generation through reverse engineering and ui pattern inferring. In *Quality of Information and Communications Technology (QUATIC), 2014 9th International Conference on the*, pages 105–115. IEEE, 2014.

[92] D. C. Schmidt. Model-driven engineering. *COMPUTER-IEEE COMPUTER SOCIETY-*, 39(2):25, 2006.

[93] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Corts. A survey on metamorphic testing. *IEEE Transactions on Software Engineering*, 42(9):805–824, Sept 2016. ISSN 0098-5589. doi: 10.1109/TSE.2016.2532875.

[94] S. M. A. Shah, C. Gencel, U. S. Alvi, and K. Petersen. Towards a hybrid testing process unifying exploratory testing and scripted testing. *Journal of Software: Evolution and Process*, 26(2):220–250, 2014.

[95] H. K. SHARMA, S. K. SINGH, and P. AHLAWAT. Model-Based Testing: The New Revolution in Software Testing. *Database Systems Journal*, 5(1):26–31, May 2014. URL https://ideas.repec.org/a/aes/dbjour/v5y2014i1p26-31.html.

[96] M. Shirole and R. Kumar. Uml behavioral model based test case generation: A survey. *SIGSOFT Softw. Eng. Notes*, 38(4):1–13, July 2013. ISSN 0163-5948. doi: 10.1145/2492248.2492274. URL http://doi.acm.org/10.1145/2492248.2492274.

[97] I. Shufer, A. Ledenev, and Y. Burg. System and method for monitoring exploratory testing by a plurality of testers, Oct. 8 2013. US Patent 8,555,253.

[98] J. C. Silva, J. Saraiva, and J. C. Campos. A generic library for gui reasoning and testing. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 121–128. ACM, 2009.

[99] B. Song, H. Miao, and S. Chen. Considering web frameset and browser interactions in modeling and testing of web applications. In *2009 International Conference on Computational Intelligence and Software Engineering*, pages 1–4, Dec 2009. doi: 10.1109/CISE.2009.5363453.

[100] A. Stavrou and G. Papadopoulos. Automatic generation of executable code from software architecture models. In C. Barry, M. Lang, W. Wojtkowski, K. Conboy, and G. Wojtkowski, editors, *Information Systems Development*, pages 1047–1058. Springer US, 2009. ISBN 978-0-387-78577-6. doi: 10.1007/978-0-387-78578-3_36. URL http://dx.doi.org/10.1007/978-0-387-78578-3_36.

[101] B. Suranto. Exploratory software testing in agile project. In *2015 International Conference on Computer, Communications, and Control Technology (I4CT)*, pages 280–283, April 2015. doi: 10.1109/I4CT.2015.7219581.

[102] H. Tanida, M. R. Prasad, S. P. Rajan, and M. Fujita. *Automated System Testing of Dynamic Web Applications*, pages 181–196. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-36177-7. doi: 10.1007/978-3-642-36177-7_12. URL http://dx.doi.org/10.1007/978-3-642-36177-7_12.

[103] H. Tanno and X. Zhang. Test script generation based on design documents for web application testing. In *2015 IEEE 39th Annual Computer Software and Applications Conference*, volume 3, pages 672–673, July 2015. doi: 10.1109/COMPSAC.2015.74.

[104] H. Tanno, X. Zhang, T. Hoshino, and K. Sen. Tesma and catg: Automated test generation tools for models of enterprise applications. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages 717–720, Piscataway, NJ, USA, 2015. IEEE Press. URL http://dl.acm.org/citation.cfm?id=2819009.2819147.

[105] G. Tassey. The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standards and Technology, 2002.

[106] M. Utting. *The Role of Model-Based Testing*, pages 510–517. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-69149-5. doi: 10.1007/978-3-540-69149-5_56. URL http://dx.doi.org/10.1007/978-3-540-69149-5_56.

[107] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007. ISBN 0123725011, 9780080466484.

[108] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson. Model-based testing of object-oriented reactive systems with spec explorer. *Formal methods and testing*, pages 39–76, 2008.

[109] R. S. Wazlawick. *Object-oriented analysis and design for information systems: Modeling with UML, OCL, and IFML*. Elsevier, 2014.

[110] J. Whittle, J. Hutchinson, and M. Rouncefield. The state of practice in model-driven engineering. *IEEE software*, 31(3):79–85, 2014.

[111] T. Yue, S. Ali, and L. Briand. Automated transition from use cases to uml state machines to support state-based testing. In *Modelling Foundations and Applications*, pages 115–131. Springer, 2011.

Publications of the Author

Articles in Impacted Journals

[A.1] K. Frajták, M. Bureš, and I. Jelínek. Exploratory testing supported by automated reengineering of model of the system under test. *Cluster Computing*, 20(1):855–865, 2017. ISSN 1386-7857. doi: 10.1007/s10586-017-0773-z. URL http://link.springer.com/article/10.1007/s10586-017-0773-z. [33%]

[A.2] M. Bureš, T. Černý, K. Frajták, and B. Ahmed. Testing the consistency of business data objects using extended static testing of crud matrices. *Cluster Computing*, Aug 2017. ISSN 1573-7543. doi: 10.1007/s10586-017-1118-7. URL https://link.springer.com/article/10.1007%2Fs10586-017-1118-7. [25%]

Publications Indexed in ISI Web of Science

[A.3] K. Frajták, M. Bureš, and I. Jelínek. Modelbased testing and exploratory testing: Is synergy possible? In *Proceedings of the 6th International Conference on IT Convergence and Security (ICITCS 2016)*, pages 329–334, Red Hook, US, 2016. ISBN 978-1-5090-3765-0. doi: 10.1109/ICITCS.2016.7740354. URL http://ieeexplore.ieee.org/document/7740354/. [33%]

[A.4] K. Frajták, M. Bureš, and I. Jelínek. Formal specification to support advanced model based testing. In *Federated Conference on Computer Science and Information Systems (FedCSIS 2012)*, pages 1311–1314, New York, US, 2012. ISBN 978-1-4673-0708-6. URL http://fedcsis.org/proceedings/fedcsis2012/pliks/90.pdf. [33%]

Publications Indexed in Elsevier Scopus

[A.5] K. Frajták, M. Bureš, and I. Jelínek. Manual testing of web software systems supported by direct guidance of the tester based on design model. *World*

Academy of Science, engineering and Technology, 80(0):243–246, August 2011. ISSN 2010-376X. URL http://www.scopus.com/inward/record.url?eid=2-s2.0-80052142179&partnerID=40&md5=b88ef067ea6c7dad229641221647c1c7. [33%]

[A.6] K. Frajták, M. Bureš, and I. Jelínek. Pex extension for generating user input validation code for web applications. In *Proceedings of the 9th International Conference on Software Engineering and Applications*, pages 315–320, Setúbal, PT, 2014. ISBN 978-989-758-036-9. doi: 10.5220/0004994103150320. URL http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0004994103150320. [33%]

[A.7] K. Frajták, M. Bureš, and I. Jelínek. Reducing user input validation code in web applications using pex extension. In *ACM International Conference Proceeding Series, Volume 883*, ACM International Conference Proceeding Series, pages 302–308, Rousse, BG, 2014. ISBN 978-1-4503-2753-4. doi: 10.1145/2659532.2659633. URL http://dl.acm.org/citation.cfm?id=2659532.2659633. [33%]

[A.8] K. Frajták, M. Bureš, and I. Jelínek. Using the interaction flow modelling language for generation of automated frontend tests. In *Position Papers of the 2015 Federated Conference on Computer Science and Information Systems*, Annals of Computer Science and Information Systems, pages 117–122, Warsaw, PL, 2015. ISBN 978-83-60810-77-4. doi: 10.15439/2015F392. URL https://fedcsis.org/proceedings/2015/pliks/392.pdf. [33%]

[A.9] K. Frajták, M. Bureš, and I. Jelínek. Transformation of ifml schemas to automated tests. In *Proceeding of the 2015 Research in Adaptive and Convergent Systems (RACS 2015)*, pages 509–511, New York, US, 2015. ISBN 978-1-4503-3738-0. doi: 10.1145/2811411.2811556. [33%]

Other Publications

[A.10] K. Frajták, M. Bureš, and I. Jelínek. Web software systems testing supported by model-based direct guidance of the tester. *Proceedings of International Conference on Information Technologies*, 2012(26):45–52, 2012. ISSN 1314-1023. [33%]

Manuscripts Submitted to Impacted Journals Currently Under Review

[A.11] M. Bureš, K. Frajták, and B. Ahmed. Automation Support of Exploratory Testing Using Model Reconstruction of the System Under Test. *Submitted to IEEE Transactions on Reliability, Special Section on Software Testing and Program Analysis.* [33%]