Pós-Graduação em Ciência da Computação

JACINTO FILIPE SILVA REIS

AIDING EXPLORATORY TESTING WITH PRUNED GUI MODELS

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
<http://www.cin.ufpe.br/~posgraduacao>

RECIFE

2017

JACINTO FILIPE SILVA REIS

AIDING EXPLORATORY TESTING WITH PRUNED GUI MODELS

Este trabalho foi apresentado à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

ORIENTADOR: **Prof. Alexandre Mota**

RECIFE

2017

Catálogo na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

R375a Reis, Jacinto Filipe Silva
 Aiding exploratory testing with pruned GUI models / Jacinto Filipe Silva
 Reis. – 2017.
 66 f.: il., fig., tab.

 Orientador: Alexandre Cabral Mota.
 Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn,
 Ciência da Computação, Recife, 2017.
 Inclui referências.

 1. Ciência da computação. 2. Análise estática. I. Mota, Alexandre Cabral
 (orientador). II. Título.

004

CDD (23. ed.)

UFPE- MEI 2017-99

Jacinto Filipe Silva Reis

Aiding Exploratory Testing with Pruned GUI Models

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação

Aprovado em: 22/02/2017.

BANCA EXAMINADORA

Prof. Dr. Juliano Manabu Iyoda
Centro de Informática/UFPE

Profa Dra Roberta de Souza Coelho
Departamento de Informática e Matemática Aplicada/UFRN

Prof. Dr. Alexandre Cabral Mota
Centro de Informática/UFPE
(Orientador)

I dedicate this dissertation to my family and my wife,
who supported me with all necessary to get here.

Acknowledgements

First and foremost, I thank God for everything, without Him I would not be able to carry out this work.

I thank my family, especially to my parents, Ildaci and Anacleto, for the solid educational foundation I received, for the zeal and incentives throughout my life.

Thank you to my lovely wife, Priscila, for her patience, support, attention, fellowship, and incentive. This was fundamental to give me strength to keep working.

I also would like to thank my advisor, Prof Alexandre Mota, who stood by me through the whole process, giving relevant insights that helped me drive this research.

In addition, I thank the Informatics Center of the Federal University of Pernambuco for great support provided to both students and professors. Thank you to all professors I had the opportunity to meet.

I also want to thank the members of my dissertation committee, professors Roberta Coelho and Juliano Iyoda, for accepting the invitation and helping to improve my work.

Finally, for all who directly or indirectly helped me in this journey, my sincere “thank you”. This research would not have been possible without the support of you.

*“Se avexer não. Toda caminhada começa no primeiro passo.
A natureza não tem pressa, segue seu compasso, inexoravelmente chega lá.”
(Accioly Neto)*

Abstract

Exploratory testing is a software testing approach that emphasizes the tester's experience in the attempt to maximize the chances to find bugs and minimize the time effort applied on satisfying such a goal. It is naturally a GUI-oriented testing activity for GUI-based systems. However, in most cases, exploratory testing strategies may not be accurate enough to reach changed code regions. To reduce this gap, in this work, we propose a way of aiding exploratory testing by providing a GUI model of the regions impacted by internal code changes (for example, as result of change requests to fix previous bugs as well as for software improvement). We create such a delimited GUI model by pruning an original GUI model, quickly built by static analysis, using a reachability relation between GUI elements (i.e., windows, buttons, text fields, etc.) and internal source code changes (classes and methods). To illustrate the idea we provide promising data from two experiments, one from the literature and another from our industrial partner.

Keywords: GUI Testing . Static Analysis . Swing Patterns . Exploratory Testing . Change Request . Release Notes

Resumo

Teste exploratório é uma abordagem de teste de software que enfatiza a experiência do testador na tentativa de maximizar as chances de encontrar *bugs* e minimizar o esforço de tempo aplicado na satisfação desse objetivo. É naturalmente uma atividade de testes orientada à GUI aplicada em sistemas que dispõem de GUI. No entanto, na maioria dos casos, as estratégias de testes exploratórios podem não ser suficientemente precisas para alcançar as regiões de código alteradas. Para reduzir esta lacuna, neste trabalho nós propomos uma forma de auxiliar os testes exploratórios, fornecendo um modelo de GUI das regiões impactadas pelas mudanças internas de código (por exemplo, como resultado de solicitações de mudanças para corrigir *bugs* anteriores, bem como, para realização de melhorias do software). Criamos um modelo de GUI delimitado, podando um modelo de GUI original, construído rapidamente através de análise estática, usando uma relação de alcançabilidade entre elementos de GUI (janelas, botões, campos de textos) e alterações de código interno (classes e métodos). Para ilustrar a ideia, nós fornecemos dados promissores de dois experimentos, um da literatura e outro de nosso parceiro industrial.

Palavras-chave: Teste de GUI . Análise Estática . Padrões Swing . Teste Exploratório . Solicitação de Mudança . Notas de Publicação

List of figures

Figure 1 – Pruning a GUI model	16
Figure 2 – Example of a test case in TestLink	20
Figure 3 – Model-Based Testing (MBT) process (extracted from [66])	22
Figure 4 – Example of a containment hierarchy (extracted from [51])	25
Figure 5 – Example of a Graphical User Interface (GUI) hierarchy (extracted from [64])	25
Figure 6 – An overview of the hierarchy and structure of the Swing toolkit	26
Figure 7 – Generic Change Request (CR) life cycle (extracted from [16])	26
Figure 8 – Example of a bug life cycle (extracted from [26])	27
Figure 9 – Example of template used to guide bug reporting process	28
Figure 10 – CFG for Listing 2.1	30
Figure 11 – A BookManager application.	33
Figure 12 – GUI model representation of BookManager application	34
Figure 13 – Visualizing additional information when hovering an edge	34
Figure 14 – Soot phases (extracted from [54])	36
Figure 15 – Pruning GUI model	47
Figure 16 – Rachota GUI model after pruning	53
Figure 17 – Rachota GUI model before pruning	54
Figure 18 – Tooltip indicating how to execute event <i>e130</i> on Rachota	55
Figure 19 – Executing event <i>e130</i> on Rachota	55
Figure 20 – About screen on Rachota	56
Figure 21 – Tooltip indicating how to execute event <i>e242</i> on Rachota	56

List of tables

Table 1 – Evaluation Results	50
Table 2 – Code coverage (related only to changed code) of exploratory testing . .	52

List of abbreviations and acronyms

API	Application Programming Interface p. 24
APK	Android Application Package p. 51
AST	Abstract Syntax Tree pp. 30 , 59
CFG	Control Flow Graph pp. 29 , 30 , 44
COMET	Community Event-based Testing p. 48
CPU	Central Processing Unit p. 49
CR	Change Request pp. 9 , 14 , 25–28 , 44 , 52
DC	Degree of Connectivity pp. 48 , 49 , 51 , 58
ET	Exploratory Testing pp. 18–20
FSM	Finite State Machine pp. 22 , 23
GTK+	GIMP Toolkit p. 58
GUI	Graphical User Interface pp. 9 , 14–17 , 23–25 , 29 , 32–36 , 38–40 , 42 , 44–46 , 48 , 49 , 51–53 , 55 , 56 , 58–60
IDE	Integrated Development Environment pp. 29 , 59
IEC	International Electrotechnical Commission p. 19
IEEE	Institute of Electrical and Electronics Engineers p. 19
ISO	International Organization for Standardization p. 19
JFC	Java Foundation Classes p. 24
JSF	JavaServer Faces p. 24
MBT	Model-Based Testing pp. 9 , 22 , 21–23 , 59
MVC	Model-View-Controller p. 24
NDA	Non-Disclosure Agreement pp. 51 , 58
PC	Personal Computer p. 49
RAM	Random Access Memory p. 49
RHS	Right-Hand Side pp. 30 , 31
SUT	Software Under Test pp. 14 , 15 , 18 , 21–23 , 27
SVG	Scalable Vector Graphics p. 34
SWT	Standard Widget Toolkit p. 24
TR	Transition Relation pp. 15 , 42
UML	Unified Modeling Language pp. 21 , 22

Contents

1	INTRODUCTION	14
1.1	Problem Overview	14
1.2	Proposal	15
1.3	Contributions	16
1.4	Outline	17
2	BACKGROUND	18
2.1	Software Testing	18
2.1.1	Exploratory Testing	18
2.1.2	Scripted Testing	19
2.1.3	Exploratory Testing vs. Scripted Testing	20
2.1.4	Regression Testing	21
2.1.5	Model-Based Testing	22
2.2	Main Tools for Software Development	24
2.2.1	Swing	24
2.2.2	Change Request	26
2.2.3	Release Notes	28
2.3	Static Analysis	29
2.3.1	Control Flow Graph	30
2.3.2	Def-Use Chains	30
3	GUI MODELING	32
3.1	GUI Representation	32
3.2	Static Analysis using Soot	34
3.3	Swing GUI Code Patterns	36
3.3.1	Identifying Windows and Components	37
3.3.1.1	Initialization	37
3.3.1.2	Connection	37
3.3.1.3	Disposer	38
3.3.2	Identifying an Event	38
3.3.2.1	Initialization	38
3.3.2.2	Connection	39
3.3.2.3	Event	39
3.4	Building the GUI model	40
3.4.1	Collecting GUI elements	40
3.4.2	Building the Paths (Transition Relation)	42

4	PRUNING THE GUI MODEL FROM CHANGED CODE	44
4.1	Getting the Changed Code	44
4.2	The Pruning Process	45
4.3	Exemplifying	47
5	EVALUATIONS	48
5.1	First Evaluation - Building Whole GUI Model	48
5.1.1	Discussion	51
5.1.2	Threats to Validity	51
5.2	Second Evaluation - Pruned GUI Model	52
5.2.1	Discussion	53
5.2.2	Exemplifying the GUI model usage	55
5.2.3	Threats to Validity	57
6	CONCLUSION	58
6.1	Related Work	59
6.2	Future Work	60
	REFERENCES	61

1 INTRODUCTION

Nowadays, applications based on a Graphical User Interface (GUI) are ubiquitous. Testing such applications are becoming harder and harder, especially due to the huge state space (possible interactions). Another recurring challenge during the testing stage is to determine which of the testing activities might be performed manually or by automation. With respect to that, many software testing techniques, strategies, and tools have been developed to support activities for creating, selecting, prioritizing and executing tests.

Regarding the creation of automated test cases for GUI-based applications, one can basically have two main approaches: (i) by capture-replay [5, 8, 22, 39, 60] with human aid; and (ii) by traversing some GUI model [7, 12, 36, 37] built automatically.

On the other hand, exploratory testing [68] is seen as one of the most successful software testing approaches for manual testing because it is based on the freedom of experienced testers that try to exercise potentially problematic regions of a system very quickly using their expertise.

1.1 Problem Overview

During an exploratory testing session, experienced testers perform several simultaneous and implicit activities. They design, prioritize and select test scenarios based on their “feelings” and previous experience and information about the Software Under Test (SUT). There is no prefixed test script or test input and both effectiveness and efficacy about achieved results (for instance, defects detected and code coverage) are strictly related to the tester experience [68].

To improve an exploratory testing session, exploratory testing usually focuses on unstable test scenarios by manually examining Change Requests (CRs) related to most recent bug fixes and/or software improvements [13]. However, in most cases, the information gathered from such reports may not be accurate enough to determine which GUI elements (for example, windows, buttons, and text fields) may be exercised to indirectly reach the affected regions. This creates a gap between GUI elements and internally changed elements. Just to give an idea of such a gap in practice, we measured the code coverage (simply based on reached methods) of an exploratory testing session of our industrial partner and we obtained a 6.8% code coverage with respect to the changed regions. This is very low and worrying.

1.2 Proposal

In order to reduce such a gap, we propose a solution that joins the two worlds of software testing approaches (manual and automation), where we create a delimited **GUI** model automatically as a way of aiding exploratory manual testing by providing information of the **GUI** parts impacted by internal code changes (for example, as result of change requests to fix previous bugs as well as for software improvement). Our proposal is structured in two main parts: (i) Building the **GUI** model; and (ii) Pruning this **GUI** model based on internal code changes recently committed.

As the first part of our proposal, we create a **GUI** model automatically. Within this context, one can find two main alternatives: (a) run-time model creation (which inherits some of the capture-replay characteristics) [24, 38, 43]; and (b) static-based model creation [53, 59]. Both are limited in some respect. We have chosen the static alternative because it seems to be more flexible in terms of manipulating the source code since it is not necessary to run **SUT** to perform an analysis, which is needed in a run-time approach. In addition, static analysis seems to be more aligned with respect to the current trend being investigated in academia and industry [6, 7, 41]. By following a static analysis approach, our main concern is on how to identify specific code fragments to collect the windows and events¹ related to widgets (for instance: buttons, combo boxes or text fields) as well as the relationship between them to build the correct set of paths that comprise our **GUI** model, called Transition Relation (**TR**) in our model definition.

As static analysis is sensitive to code writing styles, our approach uses the Soot framework [30], which transforms any Java input code into a uniform intermediate code style, and focuses on the Swing toolkit [34]. We implemented Soot transformers to capture graphical components and event listeners based on some proposed Swing code patterns. Besides, we use def-use chains to make a relationship between the widgets and their corresponding listeners. To measure the efficiency (time to create the **GUI** model) and efficacy (the degree of connectivity—formally defined here—achieved in the transition relation) of our **GUI** model builder, we employed our analyzer to 32 applications found in public repositories.

In the second step of our proposal, we perform the pruning in the **GUI** model based on modified code regions (new and changed methods). This pruned **GUI** model emerges by keeping only those **GUI** elements present in the complete **GUI** model that are related to internally changed elements by means of a transitive closure operation (or reachability analysis). For example, after running some comparison tool in order to obtain the differences between two versions of the same **SUT**, we are able to identify which regions

¹The **GUI** responds to an event by executing a piece of code registered in the event listener (sometimes called “handler methods”) related to that event.

Figure 1 – Pruning a GUI model

of source code were modified.

In Figure 1 we show an illustrative scenario of pruning where internally modified code regions are related to the edge **e15**, circled in green (as described in Section 3.1, nodes represent GUI elements and edges are an abstraction of user actions). Thereby, by applying our pruning algorithm in whole GUI model (left-hand side) we have pruned GUI model in terms of **e15** (right-hand side).

As the reader can see in Section 5.2, our experiments (from the literature and industry) illustrate that our proposed strategy brings promising results. From the literature, we increased the coverage in added and modified code regions (Java methods) from 42.86% to 71.43% and in the industry, with partial application of our approach, we increased from 6.8% to 9.75%. Although there was a small increase in coverage for our industrial experiment, it was enough to reveal 2 bugs that were not found using only the testers experiences.

1.3 Contributions

The main contributions of this work are the following:

- A model that shows all possible event sequences that can be triggered on the GUI;
- The use of Soot to build the proposed GUI model from Swing source code;

- The implementation of a Soot-based tool;
- The definition of a metric in terms of the degree of connectivity of the resulting GUI model;
- An evaluation of our tool in more than 30 GUI-based applications found in the literature and public repositories;
- A pruned GUI model based on changed code.

1.4 Outline

The remainder of this dissertation is organized as follows:

- Chapter 2 provides an overview of essential concepts used for understanding this dissertation;
- Chapter 3 presents our proposed GUI model and how we use the Soot framework in order to construct it. In addition, we show and explain our set of source code patterns for Swing and how they are used in algorithms to build the GUI model;
- Chapter 4 describes how the changed code was reflected in pruned GUI model;
- Chapter 5 details how was the evaluation of two parts of our proposed approach, besides discussing the results and respective threats to validity;
- Chapter 6 summarizes the contributions of this work, discusses related and future work, and presents our conclusions.

2 BACKGROUND

In this chapter we provide some information to better understand the rest of the dissertation. In Section 2.1, we illustrate basic concepts about software testing, in particular, with an emphasis on exploratory testing, scripted testing, regression testing and model-based testing. In Section 2.2 we describe some practices used in software development applied in our work. Lastly, in Section 2.3, we conclude the chapter discussing static analysis.

2.1 Software Testing

Software testing plays a fundamental role during the software development process, increasing the final quality of the implemented software products. Its main objective is to apply a set of techniques, methods, strategies and tools, either manually or automatically, with the objective of detecting failures in a system execution, either in its real environment or a simulated environment. This way, in the next sections we describe some important concepts used as the foundation for our work.

2.1.1 Exploratory Testing

Exploratory Testing (ET) is a testing approach that provides greater freedom for the tester to make decisions about what will be tested. Rather than following a pre-established script, during an exploratory test session, the tester acquires new knowledge about the SUT as test scenarios are exercised and in conjunction with previous experiences and skills new test scenarios emerge. In other words, both test design and test execution are made at the same time [68].

An important characteristic of ET is its flexibility and adaptability in situations where the SUT has no documented requirements or when this documentation is changed frequently. When there is no documentation, ET can be used with a focus on knowing and learning the possible behaviors of the software, as well as a mapping of the main modules and features.

When requirements are constantly updated, ET uses an artifact called a **charter**. A **charter** defines the mission of an ET session as well as the areas of concentration that should be focused by the tester. There is no prefixed step-by-step or level of detail that results in a lot of time spent in writing a charter. Thus, when there are changes in the requirements, charters can be adjusted quickly redefining the missions and areas that

should be attacked. There are many ways to describe a charter. An example is described below. This template is generally applied to [ET](#) in an agile context [10]:

Explore *<area, feature, requirement or module>*
With *<resources, conditions, or constraint>*
To discover *<information>*

A good practice when describing a charter is that it should not be so generic, to provide any relevant and applicable information, nor being so specific in such a way that it becomes a test procedure (for example, editing the name field or clicking the OK button). A good example, focusing on security issues, is depicted as follows:

Explore *all input fields in the user registration screen*
With *javascript and sql injections*
To discover *security vulnerabilities*

2.1.2 Scripted Testing

Even with the increasing use of [ETs](#) in recent years, the traditional approach based on test cases, also known as the **scripted testing** approach, is still found in many software development organizations. In general, these companies follow a typical software testing process much more well-structured in terms of defined steps. As exposed in [56], a generic testing process encompasses 5 steps: **(i)** Testing Planning and Control; **(ii)** Testing Analysis and Design; **(iii)** Testing Implementation and Execution; **(iv)** Evaluating of Exit Criteria and Reporting, and **(v)** Test Closure Activities. Although this generic process is illustrated sequentially, the activities in the test process may overlap or happen in parallel. It is usually customized according to the particularities of each project.

The stage **Testing Analysis and Design** contains, among other things, the design of high-level test cases and scenarios that should be exercised during the test execution. However, only in the stage **Testing Implementation and Execution**; it is indeed started the building of concrete test cases [56]. A **test case** is an artifact that consists of “a set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement” [1].

An important part of a test case is its **test procedure**, also known as a **test script**. It contains instructions for carrying out the test cases. According to [ISO/IEC/IEEE 24765:2010](#) a **test procedure** is a set of “detailed instructions for the setup, execution, and evaluation of results for a given test case”. For this reason, by being guided by a pre-established script, this approach based on test cases is widely called **scripted testing**.

For comparison purposes, we show in Figure 2 an example of a scripted testing

The screenshot shows a 'Test Case' window in TestLink. At the top, it says 'gm-1:GmailLogin' with a version '1'. Below this is a 'Summary' section stating the purpose is to test the login functionality of the Gmail page. A 'Preconditions' section lists two requirements: internet connectivity and valid credentials. The main part of the form is a table with four columns: 'Step actions', 'Expected Results', 'Execution', and a status column with red and green icons. It contains four steps: opening the website, entering username and password, and clicking the sign-in button. At the bottom, there are fields for 'Status' (Draft), 'Importance' (Medium), 'Execution type' (Manual), and 'Estimated exec. (min)'. Buttons for 'Create step' and 'Resequene Steps' are also present.

Step actions	Expected Results	Execution
1 Open Gmail Website	The Website should be opened.	Manual
2 Enter username in the username textbox.	Textbox should accept the entered data.	Manual
3 Enter password in the password textbox.	Textbox should accept the entered data.	Manual
4 Click on "signin" button.	Login should success and navigate to the mail box page.	Manual

Figure 2 – Example of a test case in TestLink

commonly used in tools like TestLink [63]. There is a variation with respect to the templates used but in all of them it is possible to notice a considerable amount of required information for each test case, such as: **Identifier** (e.g., *gm-1* in Figure 2), **Title** (e.g., *GmailLogin* in Figure 2), **Summary**, a list of **Preconditions** needed to perform the test case, **Steps** describing the test scenario with their corresponding **Expected Results** (in Figure 2, the pair **Step** and **Expected Result** corresponds to each line in the table, for example, in line 1, “*Open Gmail Website*” is the step and “*The Website should be opened*” is its expected result), **Importance** used to determine the priority of a test case, usually assuming values as: High, Medium and Low, and, **Execution type** typically classified as Manual or Automated.

2.1.3 Exploratory Testing vs. Scripted Testing

When one talks about **ET**, we usually recall scripted testing. It is normal to want to compare the results obtained by these two approaches. But it is important to keep in mind that they have different goals.

Scripted testing is used to confirm that the software behaves as specified, where for each requirement, a set of test cases is created and, as a consequence of the large quantity, some of these tests are usually duplicated. Each test case follows a well-defined structure that contains, among other things, the goal, the preconditions, the steps (actions) and expected results.

On the other hand, the idea in an [ET](#) is to be able to vary the test scenarios, to go beyond the steps defined by the scripts, to explore areas that are not covered by them. The variation is mainly due to the freedom to plan the next steps, the next executions, based on what has been learned about the software.

2.1.4 Regression Testing

During the software development process, as the software evolves, either by adding or changing existing modules, it is an important assignment for the test team to check whether the change conforms to what is expected for the behavior of the system. However, to check only the changed area is not enough to complete the testing process, because these modifications may add unwanted side effects. One way to determine whether the new code breaks anything that worked prior to the change is to perform a testing approach called **Regression Testing**.

Regression Testing is a quality measure to verify if the new code conforms to the behavior accepted for the old code and that the unmodified code is not being affected by the added changes [\[56\]](#). In general, during a regression testing, a set of test cases is executed after each software build or release of a new version in order to verify that the previous features continue to be performed properly. Due to its nature of detecting bugs and side effects caused by changes, a regression testing should be considered at all test levels and applied to both functional and nonfunctional tests [\[56\]](#).

Although regression tests can be executed manually, since this set of tests is constantly executed, they are strong candidates for test automation. Another important point is that the regression test cases need to be carefully selected so that with a minimal set of cases the maximum coverage of a feature is reached. If the test covers only changed or new code parts, it neglects the consequences these modifications can have on unaltered parts [\[56\]](#).

Spillner et al. [\[56\]](#) described some strategies that can be applied in the selection of regression test cases, each one has its own drawbacks. And thus the main challenge is to balance them to optimize the relation between risk and cost. The strategies commonly used are:

- Repeating only the high-priority tests according to the test plan;
- Omitting certain variations (special cases) of functional test;
- Restraining the tests to certain configurations only (e.g., testing only in one kind of operating system or one type of language);
- Restricting the scope of test to certain subsystems or test levels (e.g., unit level, integration level).

Figure 3 – MBT process (extracted from [66])

2.1.5 Model-Based Testing

Model-Based Testing (MBT) is an approach that generates a set of test cases using a formal model that describes some (usually functional) aspects extracted from SUT artifacts (for instance, Unified Modeling Language (UML) or system requirements) [66].

Figure 3 illustrates a usual MBT process. It is structured in 5 steps (Model, Generate, Concretize, Execute and Analyze) [66]. The first step (Model) aims to build an abstract model focused on aspects of the SUT that one wishes to test. This model is commonly represented by Finite State Machines (FSMs), UML state machines, and B abstract machine notation. After writing the model, its consistency is verified using some tools. This check is important to analyze whether the behavior of the model attends to the expected behavior.

The second step (Generate) is responsible for generating a set of abstract tests from the model obtained in the first step. This generation is guided by some test selection criteria that determine which tests should be derived. The test selection criteria are necessary due

to the infinite number of possibilities represented in the model [66]. Besides outputting abstract tests, the test case generator may derive two other artifacts: a requirements traceability matrix (containing the relation between functional requirements and generated abstract tests) and some model coverage reports (showing coverage statistics for operations and transition contained in the model).

In the third step (Concretize), a test script generator converts the set of abstract tests, that are not directly executable, in a set of test scripts (a.k.a. concrete test or executable tests). This transformation involves the use of some templates and mappings to translate abstract operations into the low-level SUT details.

The fourth step (Execute) executes the set of test scripts on the SUT. The way of execution depends on the kind of MBT (on-line or off-line). In an on-line MBT, the test scripts are run as they are generated. On the other hand, in an off-line MBT, the generation of test scripts and their execution are performed in different moments.

In the last step (Analyze), the analysis of results obtained during the test executions is performed. For each detected failure, there is an effort to find out the possible causes. A failure may arise due to either an existing bug on SUT or a fault in test (false positive). This step is responsible for determining what caused each failure.

An MBT provides several benefits such as an easy test suite maintenance, the reduction on testing cost and time, a better test quality due to reducing human faults, a traceability between test cases and model, and most important, it enables the detection of both requirement defect and SUT fault.

Due to the widespread use of GUIs, their importance has been increasingly emphasized. An area of the MBT emerged for this purpose. This emerging area called Model-Based GUI Testing, as the name suggests, has the main focus on deriving tests from a GUI model [11, 19, 44, 46, 69]. The model used in this approach encapsulates information about the behavior of screens that compose the SUT, typically expressed in terms of user actions (for example, enter a text into a field or click on a button), also called events. Normally, these models are built using an FSM, where test cases are created by traversing it.

In Section 3.1 we formally describe our model and how we handle events and widgets present on a GUI. Therefore, although our main purpose is to aid an exploratory tester to reach modified code regions through the application's GUI, the proposed GUI model can also be adapted to be used in conjunction with some model-based techniques. We describe in Section 6.2 the possibility of adapting our GUI model in order to be used in the test case generation.

2.2 Main Tools for Software Development

During the software development process, some tools and artifacts are used and generated with the aim of improving tools commonly used to support software development. The following sections describe some of the key concepts we use to develop our proposed solution.

2.2.1 Swing

Nowadays, most applications provide a graphical interface, also known as a **GUI**, as a means to exchange information between the user and the application's program. This ubiquity of **GUIs** makes the main programming languages provide some **GUI** toolkits that enable the creation of user interfaces. With respect to the Java programming language, there are several **GUI** frameworks designed for the most diverse contexts, for example, in web context, the most used are Spring **MVC** [58] and **JSF** [28], in mobile context, the largest used is Android [3], and, in desktop context, the most popular are Swing [61], **SWT** [62], and JavaFX [27].

Swing is the main **GUI** toolkit based on Java programming language that includes graphical components for developing desktop applications [51]. It is a part of the Java Foundation Classes (**JFC**), which is an Application Programming Interface (**API**) for building **GUIs** and adding rich graphics functionality and interactivity to Java applications.

In Swing toolkit, each graphical element that composes a **GUI**, such as a window, a button or a checkbox, is represented by a Swing component. The components that allow users interact with the user interface are called *widgets* (for example, a button or a text field). In addition, there are some Swing components that can hold other components inside it. They are known as *containers* (for instance, a menu bar or a panel).

During the implementation of a Swing application, each component must be held inside a container. Thus, as a consequence of the relationship between components, a **GUI** application creates a tree of components that is known as a *containment hierarchy*. The container at the highest level, in other words, the root of the tree of components, is called a *top-level container* (for instance, a frame or a dialog). Each program that uses Swing components has at least one top-level container. Figure 4 illustrates an example of a containment hierarchy for a Swing application.

Swing toolkit has some definitions regarding structuring the containment hierarchy, such as:

- Each Swing component can be added in a container only once. If it is added in more than one container at one time, only the last container will show the component.

Figure 4 – Example of a containment hierarchy (extracted from [51])

- The first component below the top-level container in the containment hierarchy is the *root pane*. It is a lightweight container used behind the scenes to manage the content pane and the menu bar.
- Following the containment hierarchy, below the root pane there are two components called *layered pane* and *glass pane*. The glass pane is a transparent component that is placed on top of everything. It acts like a boundary that can intercept input events for the root pane. On the other hand, a layered pane acts as the main container of a root pane. It has a component, called *content pane*, that holds all Swing components that compose a GUI in different layers. In addition, the layered pane can hold a *menu bar*, that, by default, is an optional component.

To consolidate how a GUI is usually structured, Figure 5 depicts the assembly of a GUI in terms of frame (top-level container), root pane, glass pane, layered pane, content pane and menu bar.

Due to the variety of Swing components which can compose a GUI, it is necessary a deeper study of the documentation¹ in order to understand how such components relate. Figure 6 depicts an overview of the hierarchy and structure of the Swing toolkit. This hierarchy view was used as one reference to build the static analyzer proposed in this work.

Figure 5 – Example of a GUI hierarchy (extracted from [64])

¹<https://docs.oracle.com/javase/8/docs/api/javax/swing/package-summary.html>

Figure 6 – An overview of the hierarchy and structure of the Swing toolkit

2.2.2 Change Request

A Change Request (**CR**), also called issue, is a textual document that describes a defect to be fixed or an enhancement to be developed in a software system. Some tracking tools, like Mantis [35], Bugzilla [14] and Redmine [50], are used to support the **CR** management. They enable the stakeholders to deal with various activities related to **CRs**, such as registering, assignment and tracking.

In general, a **CR** can be opened by developers, testers, or even a special group of users. Each company determines the life cycle, also known as a workflow, that **CRs** will follow after they are opened. In Figure 7, we have an example of a generic workflow commonly applied in **CR** management.

Figure 7 – Generic **CR** life cycle (extracted from [16])

The screenshot shows the 'Issues' page of the Android Open Source Project Issue Tracker. The 'Template' dropdown is set to 'Android Emulator Bug'. The 'Description' field contains the following text:

```
Please Read:
http://tools.android.com/filing-bugs/emulator

Emulator Version (Emulator--> Extended Controls--> Emulator Version):
Android SDK Tools:
Host Operating System:

Steps to Reproduce Bug:
1.
2.
3.

Expected Behavior:

Observed Behavior:
```

On the right side, there are two informational boxes:

- Tip:** Please search for an existing issue before reporting a problem as a new issue.
- Remember:** This report will be publicly visible. So, don't include passwords or other confidential information.

At the bottom, there is an 'Attach a file' button, a 'Reporter:' field with a redacted name, and 'Submit issue' and 'Discard' buttons.

Figure 9 – Example of template used to guide bug reporting process

It is expected that a **CR** contains the minimum amount of information to assist developers in resolving a failure. Thinking about that, some tracking tools provide templates to guide the user during bugs reporting. This attempts to standardize the process of creating a bug report and, consequently, facilitates the understanding of the **CR** by the team involved in its resolution. Figure 9 shows an example of such a template available on issue tracker from Android project [4].

2.2.3 Release Notes

Each software product delivering usually brings with it release notes. The release notes is a document that provides high-level descriptions of enhancements and new features integrated into the delivered release. Depending on the level of detail, release notes may contain information about which **CRs** are incorporated in the release. Release notes can be generated manually or automatically.

An analysis performed in [42] has found that most frequent items included in release notes are related to fixed bugs (present in 90% of release notes), followed by information about new (with 46%) and modified (with 40%) features and components. Given this, it is notorious the importance of release notes as another source of information in the testing activities, especially in exploratory testing sessions, where a system specification may not sometimes exist. However, it is worth pointing out that release notes only list the most important issues about a software release, having the main objective in providing quick and general information, as reported in [2].

2.3 Static Analysis

Increasing the quality during software development gains more relevance than ever nowadays. Quality brings with it several benefits, among which: user satisfaction and cost reduction. The problem of detecting bugs is strictly related to cost reduction because the sooner a problem is found in the software, the cheaper is its resolution.

There are many ways to anticipate the detection of failures in a software. Such practices can be classified into two types of verification: the dynamic analysis [12, 24, 38] and static analysis [53, 59].

Dynamic analysis requires the target object of the analysis (program codes) to be executed so that the checks take place. Software testing techniques make use of this approach, as we can see in Section 2.1.

On the other hand, the static analysis has as the main characteristic to perform an analysis and checking of program codes without the need to execute them [33]. During source code processing, static analysis transforms it into some intermediate model, a kind of abstract representation, so that such a model is used for matching some code patterns they recognize. A static analysis can handle either the source code or intermediate code as bytecode [32].

In addition, a static analysis can also perform some kind of data-flow analysis [29] (e.g., liveness analysis, reaching definitions, def-use chains). This technique is widely used for gathering information about the possible values that variables might have at various points in a program. It makes possible to identify in the program code, for instance, inaccessible parts or non-initialized variables.

Static analysis is used for many different purposes, such as helping in identifying potential software quality issues (e.g., type checking, style checking), detecting potentially vulnerable code (e.g., malware detection), or also bug finding. Usually, some static analysis tools are used during the implementation phase. They are incorporated in the programmer's Integrated Development Environment (IDE) through plug-ins such as FindBugs [21], Checkstyle [17] and PMD [47].

There are also some frameworks that allow the use of static analysis for generating, transforming and analyzing program codes, for example, ASM [9] and Soot [54]. In our approach, we use Soot for the implementation of the static analysis to build the proposed GUI model (see Chapter 3).

2.3.1 Control Flow Graph

The flow of a computer program is basically structured through chaining of functions. To deal with these sequence of operations, a static analysis tool usually makes use of an intermediate representation called Control Flow Graph (CFG).

A CFG is a way to represent a code fragment using a directed graph notation, thereby easing certain kinds of analysis (e.g., data-flow analysis). A CFG is usually built on top of the Abstract Syntax Tree (AST) or another intermediate program representation.

The instructions (e.g., assignment or function call) present in a code fragment are represented by nodes in CFG. Each node is known as a *basic block* and may group a sequence of instructions with no branches. The possible flow of control among basic blocks are represented by *directed edges*. In Figure 10, we show the CFG generated for program depicted in Listing 2.1. As there is an *if-else* statement, the CFG has two possible paths exiting from B_1 node. The T denotes the path when the condition $z > 0$ holds. And the F denotes the complementary situation, that is, when $z > 0$ is false.

Each sequence of basic blocks that defines a path through the code is called a *trace*. Thus, there are two possible traces in Figure 10: $[B_1, B_2, B_4]$ and $[B_1, B_3, B_4]$.

2.3.2 Def-Use Chains

A definition-use chain (or simply def-use chain) is a data structure used in data-flow analysis mainly for the purpose of optimizing compilers by providing definition-use relations of program variables [29].

A **definition (def)** of a variable x is a statement that assigns, or may assign, a value to x . On the other hand, a **use** corresponds to an appearance of a variable x as

Figure 10 – CFG for Listing 2.1

a Right-Hand Side (RHS) operand that results in reading its value. Therefore, for each variable definition, a def-use chain consists of a list of the places in the program that uses that variable.

In Listing 2.1, we exemplify in each line (as a comment), which is the def-use chain for each variable definition. For example, in line 4, we noticed that variable `z` located in RHS of assignment statement it is also included in list of uses of variable as well as the use in line 7. This data-flow analysis was widely used in our implementation, because Soot provides an easy way of working with it. Normally, during a Soot execution the def-use chains are automatically computed internally and they can be accessed via classes `SimpleLocalDefs`² and `SimpleLocalUses`³.

Listing 2.1 – Example of def-use chains

```
1  x = 0          // Def-Use chain = {6,7}
2  y = 1          // Def-Use chain = {4,6}
3  if (z > 0)      // Def-Use chain = {}
4      z = z + y  // Def-Use chain = {4,7}
5  else           // Def-Use chain = {}
6      y = y - x  // Def-Use chain = {6}
7  z = x * z      // Def-Use chain = {7}
```

²<https://ssebuild.cased.de/nightly/soot/javadoc/index.html?soot/toolkits/scalar/SimpleLocalDefs.html>

³<https://ssebuild.cased.de/nightly/soot/javadoc/index.html?soot/toolkits/scalar/SimpleLocalUses.html>

3 GUI MODELING

In this chapter we present the main contributions obtained in our work towards GUI modeling. In Section 3.1, we start describing how GUIs are modeled in our approach, besides detailing its formalization and showing each one of elements that compose it. Section 3.2 explains how our static analyzer was implemented using the Soot framework. Section 3.3 shows which source code patterns are collected during the static analysis to generate the GUI model. In Section 3.4, we explain the algorithms developed to build the complete GUI model. These algorithms use the patterns depicted in Section 3.3.

3.1 GUI Representation

In the literature, there are different ways for representing a GUI application in terms of a mathematical model, [38, 40, 52]. In our work, we build the GUI model in terms of small parts, such as: *Component*, *Window* and *Event*. A *Component* is any internal graphical element that the user visualizes, but which does not necessarily interact with. A *Component* that can hold or store a set of components (for instance, *JPanel* or *JMenuBar*), we classify as a *Container*. The remaining *Components* that the user can visualize, and maybe interact, (for example, *JLabel*, *JButton* or *JTextField*), we classify as *Widgets*. The most high-level graphical element that includes the whole GUI structure, generally in a tree (for instance, *JFrame* or *JDialog*), we consider as a *Window*. All available user actions (that is, key presses or mouse clicks) in a GUI application we model as *Event* and they can be associated to a *Component* or a *Window*. In this work, we handle GUIs that contain discrete and deterministic events which are triggered by a single user.

In short, our GUI elements are described as follows¹:

- An *Event* is just an action, like “press a button”, or “click the mouse”;
- A *Component* is either a *Container* or a *Widget*. We abstract away from the internal details of *Container*’s and *Widget*’s.

$$Component ::= Widget \mid Container \ll \mathbb{P} Component \gg$$

- A *Window* is a set of components, or formally $Window \subseteq \mathbb{P} Component$.

¹In this chapter the formal part uses the syntax and semantics of the formal specification language Z [57]

After showing our basic elements, we illustrate how they are grouped to build the model that represents a GUI application. In the following, we present the definition of our proposed GUI model.

Definition 3.1 (GUI Model). *A GUI Model G is a 4-tuple (W, E, SW, TR) (a directed graph), such that:*

1. W is a finite set of Window elements;
2. E is a finite set of Event elements, such that $E = \{e | (w_S, e, w_T) \in TR\}$;
3. SW is a set of starting windows ($SW \subseteq W$);
4. $TR \subseteq Window \times Event \times Window$ is a transition relation.

We exemplify our proposed GUI model using the simple application illustrated in Figure 11. In this example, the **Book Manager** application allows users to perform usual actions of adding and removing books. The main window (Figure 11, left-hand side window) has a table, which stores registered books, and two buttons, one for adding new books and another for removing selected books from the table. By pressing the **Add Book** button, a dialog window (Figure 11, right-hand side window) is displayed. This dialog is used to provide the necessary information to register a new book.

Figure 12 shows a directed graph. It illustrates the obtained GUI model for **Book Manager**. Each displayed *Window* is represented as a node, where some of these windows are starting windows (*BookManagerWindow* node in this example is a starting window). As described in Definition 3.1, each *Event* is a user action and corresponds to an edge of the graph. Event $e1$ expresses a click on the **Add Book** button. This action is available in the main window (node $w1$), and after it is triggered it opens the **Add Book** dialog, captured by the node $w2$. The events $e3$ and $e4$, which are exit events of this dialog window, represent the possibilities of clicking on buttons **OK** and **Cancel**, respectively. They are connected to the main window because after they are fired, the dialog is closed and the focus returns to the **BookManager** window ($w1$). The remaining event ($e2$) encodes the

Figure 11 – A BookManager application.

Figure 12 – GUI model representation of `BookManager` application

Figure 13 – Visualizing additional information when hovering an edge

action of removing books. As it does not open another window, both source and target windows are the same ($w1$).

In addition to the mathematical model, we build a visual representation of GUI model as shown in Figure 12. This GUI model is output in a Scalable Vector Graphics (SVG) file. It allows us to add complementary information that can be accessed interactively. In our example, by hovering the mouse cursor over each edge or edge label, an additional information about how to trigger that event is displayed as a tooltip, as illustrated in Figure 13.

3.2 Static Analysis using Soot

To build the aforementioned GUI model, we perform a static analysis, implemented using the Soot framework, in the Java bytecode [32] of the application under analysis.

The main motivation of using Soot to construct the GUI model, instead of the ASM framework [9], is that it takes as input a Java bytecode or source code and can generate specific intermediate representations (that is, Jimple [67] or Shimple [65]), each

one with a different level of abstraction depending on the analysis purpose.

As intermediate code we use Jimple, which is a three-address representation of the corresponding Java bytecode. A benefit of using Jimple is that the analysis only has to deal with specific combinations of 15 types of statements instead of more than 200 different types of statements available in the Java bytecode. Listing 3.1 shows a Java snippet of a main method and Listing 3.2 shows its Jimple corresponding code.

Listing 3.1 – Java snippet code of a main method

```
public static void main(String[] args) {
    Foo f = new Foo();
    int a = 7;
    System.out.println(f.bar(a));
}
```

Listing 3.2 – Jimple code generated for the main method

```
public static void main(java.lang.String[]) {
    java.lang.String[] args;
    Foo $r0;
    java.io.PrintStream $r1;
    int $i0;

    args := @parameter0: java.lang.String[];
    $r0 = new Foo;
    specialinvoke $r0.<Foo: void <init>()>();
    $r1 = <java.lang.System: java.io.PrintStream out>;
    $i0 = virtualinvoke $r0.<Foo: int bar(int)>(7);
    virtualinvoke $r1.<java.io.PrintStream: void println(int)>($i0);
    return;
}
```

As depicted in Figure 14, Soot’s execution is structured in a sequence of phases, where each phase is implemented by a **Pack**. During execution cycle, each packaged plays a different role, for example, **cg** pack constructs a call graph for whole program analysis and **wjtp** pack performs the Jimple transformation for the whole program.

Each phase contains sub-phases that are actually responsible for applying any kind of manipulation under intermediate representations. Each sub-phase is known as a Soot transformer and it can be implemented as a **SceneTransformer** or a **BodyTransformer**. The **SceneTransformer** allows the handling of the entire program at once because has access to all classes of the application that is being analyzed. On the other hand, a **BodyTransformer** is most appropriate for an intraprocedural analysis, because it is invoked in all methods of the application.

Figure 14 – Soot phases (extracted from [54])

In our solution, we implement some transformers, each one extending a `SceneTransformer`. We present the main ones (Algorithm 1 and Algorithm 2) that build the complete GUI model by traversing the source code of an application and applying the proposed patterns (presented in Section 3.4).

3.3 Swing GUI Code Patterns

Recall from Definition 3.1 that our GUI model is mainly composed of windows and events, and naturally their relationships. In this section, we present how we identify these elements using Swing Source Code patterns related to GUI components and event listeners. These patterns are related to method calls, classified into six groups. In our approach, all patterns are applied to both Swing types (i.e, `JFrame` or `JButton`) and sub-types derived from Swing types (user-defined types). To be self-contained, we present all proposed patterns in Java as well as in Jimple. We use these patterns as input to build our proposed GUI Model.

In Section 3.4, we present our algorithm responsible for traversing the application's code and building the GUI model. This algorithm tries to identify the patterns repetitively.

To improve readability and presentation, our material follows a top-down structure, where in Section 3.3.1 we identify Windows (used to fill variable W of Definition 3.1) and in Section 3.3.2 we identify Events and their relationships to Windows (filling both E and TR elements of Definition 3.1).

3.3.1 Identifying Windows and Components

To identify Windows and internal components, we use three kinds of Swing Code Patterns: (i) Initialization, (ii) Connection, and (iii) Disposer.

3.3.1.1 Initialization

Pattern 1 contains a generic pattern, which is instantiated for several different Swing elements, that initializes a window and its internal components.

Pattern 1 (InitGuiElementPattern).

1. **Java:** *GuiType* *var* = **new** *GuiType*(*parVars*)
2. **Jimple:** *specialinvoke* *var*:<*GuiType*: **void** <*init*>(ParTypes)>(parVars)

where *GuiType* $\in \{JFrame, JPanel, JButton \dots\}$, *var* and *parVars* are variable identifiers, *ParTypes* are different kinds of parameters dependent on the constructor being used.

The algorithm in Section 3.4 applies Pattern 1 by checking whether *GuiType* matches the expected types. If a matching is found, either a window or a component, such an element is collected and stored in the sets *W* (Window) or *C* (Component), respectively.

Recall from Definition 3.1 that a window has a set of constituent components. To identify which component belongs to which window we apply the next pattern.

3.3.1.2 Connection

We use Pattern 2 to create the associations between components and windows. By looking at the Swing documentation, we noticed that there are several ways of making this relationship in terms of source code. Our proposed pattern tries to capture such generic and specific situations.

Pattern 2 (EmbedGuiElementPattern).

1. **Java:** *var.mthType*(*parVars*)
2. **Jimple:** *virtualinvoke* *var*:<*GuiType* *Ret mthType*(ParTypes)>(parVars)

where *GuiType* $\in \{JFrame, JPanel, JMenuBar \dots\}$, *mth* $\in \{add, insert, set\}$, *Type* $\in \{\emptyset, TopComponent, BottomComponent, RightComponent, LeftComponent, ViewportView\}$,

Ret is the return type which is dependent on the *methType*, *var* and *parVars* are variable identifiers, *ParTypes* are different kinds of parameters dependent on the *methType*.

During our analysis, when Pattern 2 matches, we create a connection between elements in sets *W* or *C*, by using the elements *var* and *parVars*. We use def-use chains to identify the place in the code where these elements were initialized.

3.3.1.3 Disposer

Pattern 3 is not directly related to Window and Component sets themselves but it contributes to the transition relation between Windows and Events. During our analysis, Windows visibility change according to the user interaction. When a Window is hidden, a fact captured by Pattern 3, we can create a potential association between an Event triggering and the corresponding Window. By abstracting the possible visibility situations to only hiding a Window, we can get unconnected Windows. This is discussed in detail in Section 5.1.2 and the main reason we have formalized the kind of graphs we can get from the *TR* element of a GUI model and how to calculate the coverage our algorithm can obtain for an application.

Pattern 3 (DisposerPattern).

1. **Java:** *var.mth(parVars)*
2. **Jimple:** *virtualinvoke var.<GuiType> void mth(ParTypes)>(parVars)*

where *GuiType* $\in \{JFrame, JDialog\}$, *meth* $\in \{setVisible, dispose\}$, *var* and *parVars* are variable identifiers, *ParTypes* are different kinds of parameters dependent on the *meth*.

Like in Pattern 2, Pattern 3 also uses def-use chains to identify the GUI elements.

3.3.2 Identifying an Event

To identify Events, we use three kinds of Swing Code Patterns: (i) Initialization, (ii) Connection and (iii) the Event itself.

3.3.2.1 Initialization

Similarly to Pattern 1, Pattern 4 contains patterns that initialize an event listener. Each event listener stores a set of available events for a kind of GUI component. This fills an auxiliary structure prior to store definitive Events in the set *E*.

Pattern 4 (InitListenerPattern).

1. **Java:** *ListType* *var* = **new** *ListType*(*parVars*)
2. **Jimple:** *specialinvoke* *var* <*ListType*: **void** <*init*>(ParTypes)>(parVars)

where *ListType* $\in \{\text{MouseListener}, \text{ActionListener}, \text{KeyListener} \dots\}$, *var* and *parVars* are variable identifiers, *ParTypes* are different kinds of parameters dependent on the constructor being used.

The algorithm in Section 3.4 applies Pattern 4 by checking whether *ListType* matches the expected listener types. If a matching is found, such an event listener is collected and stored in the set of listeners. Elements of this set are lately used to fill the set *E*.

Although not present in Definition 3.1, a listener determines which events will be triggered after a user action. Each GUI element, which has to handle an event triggered by the user, may use an event listener.

3.3.2.2 Connection

Pattern 5 contains patterns that associate an event listener to a GUI element.

Pattern 5 (AddListenerPattern).

1. **Java:** *var.addListType*(*par*)
2. **Jimple:** *virtualinvoke* *var* <*GuiType* **void** *addListType*(*ListType*)>(par)

where *GuiType* $\in \{\text{JFrame}, \text{JPanel}, \text{JButton} \dots\}$, *ListType* $\in \{\text{MouseListener}, \text{ActionListener}, \text{KeyListener} \dots\}$, *var* and *par* are variable identifiers.

When we identify an association between a GUI element (a Window or Component) and an event listener, we store this relationship. We also use def-use chains to identify the place in the code where these elements were initialized. These associations are lately used to fill the transition relation *TR*.

3.3.2.3 Event

Pattern 6 contains patterns concerning the identification of methods that represent actual events. These methods are not called explicitly. They act as responses from user actions.

Pattern 6 (EventPattern).

1. **Java:** *void mthName(ParType)*
2. **Jimple:** $\langle \text{ListType: } \text{void mthName(ParType)} \rangle$

where $\text{ListType} \in \{\text{MouseListener}, \text{ActionListener}, \text{KeyListener} \dots\}$, $\text{mthName} \in \{\text{mouseClicked}, \text{actionPerformed}, \text{keyPressed} \dots\}$, ParType are different kinds of parameters dependent on the mthName being used.

Each event listener has a set of available events. For example, in `KeyListener`, there are three methods (`keyPressed`, `keyReleased`, `keyTyped`), each one representing an event callback.

During analysis, when Pattern 6 matches, we look at event method to detect whether a certain window has started or if the currently visible window is hidden in its scope. For each match, we store the corresponding event in the set E . And with the window that was created and hidden in each event method, we use to fill the transition relation TR .

3.4 Building the GUI model

This section explains the main algorithms used to build the proposed GUI model. By traversing an application's source code, we apply the patterns detailed in Section 3.3 to fill the elements present in the model as described in Definition 3.1. In order to facilitate the understanding, we describe our GUI model building process in two separated algorithms.

3.4.1 Collecting GUI elements

Algorithm 1 encompasses all the process responsible for the GUI model's construction. As the first step, we initialize some sets (E , W , and SW) that are part of the GUI model, as defined in Definition 3.1. The C set stores the other GUI elements (buttons, panels, etc) used to build the connection between windows and components. Then, the algorithm starts the analysis at the method level. From line 4 to 24, it traverses all statements trying to match the set of patterns we proposed in Section 3.4. In line 5, the algorithm tries to match the first group of patterns stated in Pattern 1. If a match is found, we create the corresponding GUI element (G), invoking function *CreateGuiElement*, and insert G in its respective set (see lines 8 and 13). If G is a window element, it is stored in W set. On the other hand, the other elements are stored in C set. Besides that, from line 9 to 11, we check whether the currently analyzed method is the main method, it means,

Algorithm 1 Build GUI Model

Input: The GUI Application source code SC **Output:** A GUI Model, $GUIModel = (W, E, SW, TR)$

```

1: function BUILDGUIMODEL
2:    $C, W, E, SW \leftarrow \emptyset, \emptyset, \emptyset, \emptyset$ 
3:   for each method  $M_i$  : Program Methods from  $SC$  do
4:     for each statement  $S_i$  : Statements from  $M_i$  do
5:       if  $S_i$  matches InitGuiElementPattern then
6:          $G \leftarrow \text{CREATEGUIELEMENT}(S_i)$ 
7:         if  $G$  is a window then
8:            $W \leftarrow W \cup \{G\}$ 
9:           if  $M_i$  is main method then
10:             $SW \leftarrow SW \cup \{G\}$ 
11:          end if
12:         else
13:            $C \leftarrow C \cup \{G\}$ 
14:         end if
15:       else if  $S_i$  matches EmbedGuiElementPattern then
16:          $\text{CREATECONNECTION}(S_i)$ 
17:       else if  $S_i$  matches DisposerPattern then
18:          $\text{STOREHIDDENWINDOW}(S_i)$ 
19:       else if  $S_i$  matches InitListenerPattern then
20:          $\text{STORELISTENER}(S_i)$ 
21:       else if  $S_i$  matches AddListenerPattern then
22:          $\text{ASSOCIATELISTENERTOGUIELEMENT}(S_i)$ 
23:       end if
24:     end for
25:   if  $M_i$  matches EventPattern then
26:      $E \leftarrow E \cup \{\text{CREATEEVENT}(M_i)\}$ 
27:   end if
28: end for
29: return  $(W, E, SW, \text{BUILDTR}(SC))$ 
30: end function

```

a method that starts the application, not necessarily a Java main method. If it is, **GUI** element (G) is also inserted in starting windows set (SW).

In line 15, we try to match Pattern 2. If it matches, function *CreateConnection* extracts involved **GUI** elements and creates a connection between them. This method makes use of sets W and C . In line 17, if the current statement matches Pattern 3, function *StoreHiddenWindow* collects and stores information about which windows may be disposed. This step pays attention in statements like `w.setVisible(false)`. In line 19, we try to match Pattern 4. If successful, the function *StoreListener* creates and stores a listener element. In line 21, if Pattern 5 matches, it makes an association between a **GUI** element and a listener². The function *AssociateListenerToGuiElement* treats statements like `btn.addFocusListener(fcsList)`, where `btn` is a `JButton` element and `fcsList` is an instance of `FocusListener` class, which in turn, provides two methods (`focusGained` and `focusLost`) to treat events related to focus.

Finally, the last part of the Algorithm 1 (lines 25–27) checks whether method M_i matches Pattern 6, and then creates an event element (via function *CreateEvent*) and insert it in the set E , in other words, it identifies if the currently analyzed method corresponds to a method that will be called when an event occurs, like `focusGained` and `focusLost` exemplified previously.

To complete our **GUI** model is necessary to build the **TR**. The TR is responsible for the navigation between nodes (*Windows*) in our proposed model. In order to give a better understanding how **TR** is filled, we describe this step in Algorithm 2 and it is invoked via function *BuildTR*.

3.4.2 Building the Paths (Transition Relation)

Algorithm 2 is responsible for filling the transition relation TR of Definition 3.1. The TR causes the model to be navigable. It details which events should be triggered so that other windows are displayed or disposed, what can be observed by the composition of each TR entry, that is a 3-tuple of (*Window*, *Event*, *Window*).

Firstly, in Algorithm 2, we iterate over the associations between **GUI** elements (G_i) and event listeners (L_i). They were collected via function *AssociateListenerToGuiElement* (line 22) described in Algorithm 1. For each iteration, we call the function *GetWindow* to return the window that the **GUI** element G_i belongs to. This window represents the source window (W_S) of a transition relation entry. After that, in line 5, we iterate over all event methods associated with the listener L_i . As mentioned before, an event listener stores a set of available events, each are represented by methods, such as `focusGained` and `focusLost`.

²A listener includes the methods that will be called whenever an event occurs

Algorithm 2 Build Transition Relation

Input: The GUI Application source code SC **Output:** The transition relation TR

```

1: function BUILDTR
2:    $TR \leftarrow \emptyset$ 
3:   for each pair  $(G_i, L_i)$  : GUI-Element/Listener associations do
4:      $W_S \leftarrow \text{GETWINDOW}(G_i)$ 
5:     for each event  $E_i$  :  $\text{GETEVENTS}(L_i)$  do
6:        $W_{TS} \leftarrow \text{GETOPENEDWINDOWS}(E_i)$ 
7:       if  $W_{TS} = \emptyset$  then
8:         if  $\text{CONTAINSDISPOSERCALL}(E_i)$  then
9:            $TR \leftarrow TR \cup \{(W_S, E_i, \text{PREVIOUSWINDOW}(W_S))\}$ 
10:        else
11:           $TR \leftarrow TR \cup \{(W_S, E_i, W_S)\}$ 
12:        end if
13:      else
14:        for each window  $W_T : W_{TS}$  do
15:           $TR \leftarrow TR \cup \{(W_S, E_i, W_T)\}$ 
16:        end for
17:      end if
18:    end for
19:  end for
20:  return  $TR$ 
21: end function

```

For each event E_i , from the set of events associated with the event listener L_i , we call the function *GetOpenedWindows* to access the set of target windows (W_{TS}) that were opened after the user triggers the event E_i . An empty set, at this point, means that after launching event E_i , no window opens and it can result in two possibilities: (i) The current window is disposed, or (ii) the event does not affect the window visibility. To check the first possibility, we use the function *ContainsDisposerCall* (line 8). If it occurs, we insert a new entry in the set TR as depicted in line 9, where function *PreviousWindow* returns the window that holds the focus after closing the current window. If there is no disposer call, we insert an entry where both source and target windows are the same (see line 11). Finally, if an event opens other windows ($W_{TS} \neq \emptyset$), we iterate over these windows and—for each one—we insert a new entry indicating that W_S goes to W_T through event E_i (see line 15). Upon completion, transition relation TR has all transitions.

4 PRUNING THE GUI MODEL FROM CHANGED CODE

This chapter presents how we can focus on relevant parts of the software. That is, we only pay attention to added and modified code regions because they are the most unstable in general and thus more susceptible to bug introduction. By using the application's source code we prune the full GUI model created as depicted in Chapter 3.

4.1 Getting the Changed Code

Industry usually uses CRs reports and release notes as sources of information to exploratory testers. But in this chapter we show that we can use an even more accurate artifact, the source-code itself, to prune the GUI model keeping just the regions where the exploratory testers may act over.

These regions are acquired by accessing the application source code as well as the repository that contains the several versions of that application. From this, we calculate the code difference between any two versions of the application. This is indeed the hardest part of our analysis because this is not just getting file differences as provided by Git diffs. By using Soot, we perform static analysis to detect new and modified methods, without worrying about source code line locations, spacing, etc. For the purpose of this work, we always consider the current version of the application and the version related to the newest CR in a release notes. This allows getting all changed code in the right time period. Algorithm 3 details the steps used to collect the changed code¹.

In our analysis, we focus on changes in terms of methods. The algorithm takes as input two applications (current version and previous one) and returns a set with all changed code. We start initializing the output set (CM), and then, by invoking the function *GetMethods* we obtain the sets with all methods present in each application. A method entry is uniquely identified by its method signature. To find out the new methods we simply perform the difference of two sets (line 6), in this case, we subtract the set containing methods of the current version ($MethsApp_1$) from the other set ($MethsApp_2$). With this, we have the information about newly added methods, that is a part of changed code. They are added in the set of changed methods (line 9).

In order to detect which methods were modified, we first have to know which ones were preserved among the versions. Similarly, we perform an operation between sets,

¹In this chapter, we use the term “changed code” to refer to both new added and modified methods

but now we apply intersection. Having the set of methods kept between versions it is possible to find out if their body was modified by comparing their CFGs. A CFG is a graph representation of computation and control flow in the program. As a CFG includes all possible paths of a method execution, then it also encodes the method behavior. This comparison is depicted in algorithm from line 10 to 16. After obtaining two CFGs (lines 11 and 12) for the same method in both versions of the application, their equality is verified. If they are not equal, it means that the body of the method has been modified and, therefore, this method should be stored in the set of changed methods (line 14). After identifying all methods that were modified, we will finally have the set with all changes between the versions. This resulting set is used as the input to the pruning process described in Algorithm 4.

Algorithm 3 Collecting the diffs

Input: Current Application App_1 and Previous Application App_2

Output: List of Changed Methods CM

```

1: function GETTINGDIFFS
2:    $CM \leftarrow \emptyset$ 
3:    $MethsApp_1 \leftarrow \text{GETMETHODS}(App_1)$ 
4:    $MethsApp_2 \leftarrow \text{GETMETHODS}(App_2)$ 
5:
6:    $NewMeths \leftarrow MethsApp_1 \setminus MethsApp_2$ 
7:    $CM \leftarrow CM \cup NewMeths$ 
8:    $PreservedMeths \leftarrow MethsApp_1 \cap MethsApp_2$ 
9:
10:  for each preserved method  $PM_i : PreservedMeths$  do
11:     $CFG_1 \leftarrow \text{GETCFG}(App_1, PM_i)$ 
12:     $CFG_2 \leftarrow \text{GETCFG}(App_2, PM_i)$ 
13:    if not EQUALCFG( $CFG_1, CFG_2$ ) then
14:       $CM \leftarrow CM \cup PM_i$ 
15:    end if
16:  end for
17:
18:  return  $CM$ 
19: end function

```

4.2 The Pruning Process

Algorithm 4 describes the required steps to perform the pruning in GUI model. It starts initializing the model variable ($PrunGM$) that will be filled when traversing the algorithm. The pruning process is indeed started in line 4, by iterating over all events present in the complete model (as described in Section 3.4). For each event E_i , we invoke the function *GetEventMethod* to get the method associated to the event (for

instance, `mouseClicked` and `keyPressed`). After obtaining method M_i , the algorithm calls *TransitiveTargets*, responsible for returning a set ($Targs_i$) with all reached methods from the method M_i . This function is another ease provided by Soot framework, more details are present in Soot survivor's guide [20].

By following the algorithm, in the next step, we traverse the list of changed methods CM , that is a result of Algorithm 3. For each changed method CM_j , we check whether it belongs to the set $Targs_i$. If it belongs to that set, it means that method CM_j can be reached from event E_i and we need to collect all the paths that arrive at this event E_i from starting windows, in other words, it gets all entries in TR which are part of paths that reach E_i starting from any starting window (SW). Method *GetPaths* brings this information in a 4-tuple containing elements that compose a GUI model, that is, (W, E, SW, TR) , as formally described in Definition 3.1. As the last step, the algorithm stores all paths found in the resulting model $PrunGM$. Upon completion of the process listed from line 4 to 14, we have a pruned GUI model.

Algorithm 4 Pruning GUI Model

Input: GUI Model $GM = (W, E, SW, TR)$ and List of Changed Methods CM

Output: Pruned GUI Model $PrunGM$

```

1: function PRUNINGGUIMODEL
2:    $PrunGM \leftarrow (\emptyset, \emptyset, \emptyset, \emptyset)$ 
3:
4:   for each event  $E_i$  : set of events  $E$  from  $GM$  do
5:      $Meth_i \leftarrow \text{GETEVENTMETHOD}(E_i)$ 
6:      $Targs_i \leftarrow \text{TRANSITIVETARGETS}(Meth_i)$ 
7:
8:     for each changed method  $CM_j$  :  $CM$  do
9:       if  $CM_j \in Targs_i$  then
10:         $Paths \leftarrow \text{GETPATHS}(GM, E_i)$ 
11:         $\text{STORESPATHS}(PrunGM, Paths)$ 
12:       end if
13:     end for
14:   end for
15:
16:   return  $PrunGM$ 
17: end function

```

Figure 15 – Pruning GUI model

4.3 Exemplifying

Figure 15 depicts a GUI model before and after the pruning process. In this illustrative example, the left-hand side shows the whole GUI model generated for one GUI application, as stated in Definition 3.1. By applying Algorithm 4 on this model in conjunction with the list of changed code (obtained after using Algorithm 3), it was identified that all modified code regions are related to the event **e15** (circled in green). Thus, by getting all paths related to **e15** we have the pruned GUI model presented on the right-hand side.

5 EVALUATIONS

In this chapter, we present and discuss the experiments, obtained results and threats to the validity of two parts of our proposed approach. In Section 5.1 we show our first experiment related to how effective are our proposed algorithms and patterns to build a whole GUI model. In Section 5.2, we show the results of our second evaluation: a comparison between applying exploratory testing on original and pruned GUI model of two applications, one from the literature and another from our industrial partner.

5.1 First Evaluation - Building Whole GUI Model

In this section, we illustrate the application of our proposed building process using our GUI model builder on 32 applications found in public repositories, most of them are in SourceForge [55], with the exception of *TerpWord* that is available from the Community Event-based Testing (COMET) [18]. Our goal in choosing these applications is to check the Degree of Connectivity (DC) (related to the GUI model) our GUI model builder achieves on analyzing different code writing styles, the abstractions assumed in Section 3.3 and application sizes. The degree of connectivity (a numerical measure) is directly related to the effectiveness of our proposed algorithms and patterns. For this purpose, in this section, we present formally what we mean by the degree of connectivity. It is based on the notion of a disconnected component of a graph. That is, for an unconnected graph, the largest (in the sense defined here as well) constituent graph is the disconnected component.

To define the disconnected component of our GUI models, we need first to define the notion of a connected graph. From now on assume that for a GUI model G , if $tr \in TR$ then tr_S is the source window and tr_T is the target window.

Definition 5.1 (Connected graph). *Let TR be a transition relation of G . Then, G is a connected graph (represented by $\overleftrightarrow{TR}$) iff $\forall tr^A, tr^B : TR \bullet tr_S^A \rightsquigarrow tr_T^B \vee tr_S^B \rightsquigarrow tr_T^A$, where $\rightsquigarrow$ means a transitive closure modified to compose triples (w_S, e, w_T) , disregarding the event e .*

To define¹ the disconnected component of our GUI models, we need first to define the notion of the size of a connected graph.

Definition 5.2 (Size of a connected graph). *Let $\overleftrightarrow{TR}$ be a connected graph. Its size is given by $Size(\overleftrightarrow{TR}) = \# \overleftrightarrow{TR} + \#\{tr_S, tr_T \mid tr \in \overleftrightarrow{TR}\}$*

¹Like in Chapter 3, in this chapter the formal definitions are given in the Z language.

Definition 5.2 states that the size of a connected graph is given by the number of transitions ($\# \overset{\leftrightarrow}{TR}$) plus the number of distinct source and target windows ($\#\{tr_S, tr_T \mid tr \in \overset{\leftrightarrow}{TR}\}$).

Definition 5.3 (Disconnected graph). *Let $\overset{\leftrightarrow}{TR}_1$ and $\overset{\leftrightarrow}{TR}_2$ be two connected graphs, such that $\overset{\leftrightarrow}{TR}_1$ is disconnected from $\overset{\leftrightarrow}{TR}_2$ (formally represented by $\overset{\leftrightarrow}{TR}_1 \parallel \overset{\leftrightarrow}{TR}_2$) iff $\forall tr^1 : \overset{\leftrightarrow}{TR}_1, tr^2 : \overset{\leftrightarrow}{TR}_2 \bullet tr_S^1 \not\rightsquigarrow tr_T^2 \wedge tr_S^2 \not\rightsquigarrow tr_T^1$.*

Definition 5.4 states that the disconnected component is the one connected graph with the largest size. Obviously, if the transition relation is fully connected this definition reduces to the unique connected graph lying in the GUI model.

Definition 5.4 (Disconnected component). *Let $G = (W, E, SW, TR)$ be a GUI model. Its disconnected component is given by:*

$$DC(G) = \overset{\leftrightarrow}{TR}^D \iff \exists \overset{\leftrightarrow}{TR}^D \subseteq TR \bullet \forall \overset{\leftrightarrow}{TR}_m \subseteq TR \setminus \overset{\leftrightarrow}{TR}^D \\ \mid \overset{\leftrightarrow}{TR}^D \parallel \overset{\leftrightarrow}{TR}_m \bullet Size(\overset{\leftrightarrow}{TR}^D) \geq Size(\overset{\leftrightarrow}{TR}_m)$$

Finally, we can present the definition of the degree of connectivity (DC) of a GUI model in Definition 5.5.

Definition 5.5 (Connectivity). *Let $G = (W, E, SW, TR)$ be a GUI model. Its degree of connectivity is given by*

$$\mathcal{C}(G) = \frac{Size(DC(G))}{(\#TR + \#W)}$$

Note the above $\mathcal{C}(G)$ becomes 100% if the disconnected component equals the transition relation TR . In this case, the expression $\#\{tr_S, tr_T \mid tr \in \overset{\leftrightarrow}{TR}\}$ reduces to $\#W$. That is, our GUI model builder did not generate any disconnected graph by successfully recognizing all code patterns needed.

The obtained results are depicted in Table 1 where for each application, we show the application category, the average ($\bar{x}(\text{Gen})$) and standard deviation ($\sigma(\text{Gen})$) time for generating the GUI model (Gen), the number of detected transition relations ($\#TR$), the number of detected windows ($\#W$) and the DC of captured GUI model. All time values are in second (s).

The evaluation was performed using a PC with an Intel Core i7-4550U CPU and 8GB RAM. The evaluation machine runs a 64-bit Windows 7 and the Oracle Java Virtual Machine version 1.7.0_69. Also, for each analyzed application, we run ten times in order to reach the precise average execution time.

Table 1 – Evaluation Results

Application	Category	$\bar{x}(\text{Gen})$	$\sigma(\text{Gen})$	#TR	#W	DC
Rachota	Timetracker	31.22	4.45	325	19	94.47%
TerpWord	Word Processor	44.58	2.61	567	27	99.83%
CrispySyncNotes	File Manager	8.81	0.77	61	12	89.04%
StreamRipStar	Sound Recorder	15.51	0.82	903	25	98.59%
Syntactic Tree Desig.	System Modeling	6.99	0.68	32	4	100%
JSymphonic	File Manager	34.28	0.70	85	11	92.70%
jMemorize	Education	38.96	1.35	83	14	98.96%
RepairsLab	System Support	35.06	0.70	393	26	96.42%
HoDoKu	Puzzle Game	138.00	4.85	863	40	91.80%
YaMeG	Converter	11.59	0.54	22	2	95.83%
Bitext2tmx	Converter	9.00	0.65	44	7	94.11%
SyncDocs	File Transfer	6.48	0.40	23	3	100%
JMJRT	Converter	8.67	0.83	67	7	97.29%
OpenRocket	Simulator	102.22	4.76	1497	64	93.97%
Screen Pluck	Screen Capture	4.77	0.31	10	1	100%
PasswordManager	Password Manager	8.80	0.32	44	8	82.26%
MyPasswords	Password Manager	15.66	0.02	119	15	88.80%
IPMonitor	IP Monitor	11.79	0.35	61	7	100%
Biogenesis	Simulator	9.58	0.31	154	8	100%
Hash Calculator	Calculator	6.80	0.77	66	4	98.57%
MidiQuickFix	Sound Player	10.57	0.31	105	12	94.87%
File Master	File Manager	5.11	0.49	22	3	100%
EarToner	Ear Training	7.25	0.69	11	2	100%
Mail Carbon	SMTP Proxy	25.73	0.56	24	3	85.18%
Simple Calculator	Calculator	5.79	0.31	18	1	100%
JTurboExplorer	Database	12.31	0.68	88	8	70.83%
JConvert	Converter	7.32	0.53	37	5	90.47%
Java LAN Messenger	Instant Messaging	6.49	0.63	56	4	25%
JScreenRecorder	Screen Capture	11.77	0.32	54	7	100%
BlueWriter	Word Processor	4.65	0.03	67	4	100%
ExtractData	Data Extractor	12.64	0.37	75	14	79.77%
OSwiss	Board Game	28.27	5.07	71	11	95.12%

5.1.1 Discussion

It is worth noting from Table 1 that the degree of connectivity, as stated in Definition 5.5, achieved by our GUI model builder is high in general, where in some cases it is 100%. Below we present a brief discussion towards what can be done to increase the degrees of connectivity of the several applications.

- Considering unconnected starting windows (splash windows), we can get the following increments: **Rachota** (94.76%), **StreamRipStart** (100%), **YaMeG** (100%), **MyPasswords** (100%), **MidiQuickFix** (97.43%), **Hash Calculator** (100%), and **JConvert** (100%);
- Discarding unconnected graphs related to testing (unused code), we can get: **JSymphonic** (98.93%), **PasswordManager** (100%), and **Mail Carbon** (92.59%);
- Discarding unconnected graphs that are not called at runtime, we can get: **Bitext2tmx** (100%), **OpenRocket** (100%), and **JTurboExplorer** (78%);
- Considering unconnected starting windows (splash windows) as well as discarding those not called at runtime, we can get: **RepairsLab** (100%) and **Java LAN Messenger** (100%).

5.1.2 Threats to Validity

We now discuss some identified factors that may affect the validity of our results. The first threat to validity is that our GUI model represents an approximation of the actual event-flow of the GUI application. This might cause the generation of some unreachable event sequences invalidating the test case. This is presented in Table 1 as the DC our tool can achieve by analyzing an application. From these data, we can see that the performance of our algorithm and patterns is considerably good enough.

The second threat to validity is the adaptation of our approach to other paradigms of building graphical user interfaces. For instance, the current Android trend. We have evaluated our approach under Java applications built upon the Swing toolkit. Assuming that the Soot framework can handle Android Application Package (APK) files and an APK file is implemented using Java, we think that our proposal is easily portable.

5.2 Second Evaluation - Pruned GUI Model

In Table 2, we present experimental data about code coverage (related only to changed code regions) of exploratory testing sessions conducted in software from the literature (Rachota 2.4 [49]) and one from our industrial partner². It is important to highlight that as our approach is focused on Swing GUI code patterns and there are considerable differences between Swing and Android, we were only able to partially apply our proposed strategy in the industry partner. Rachota took exploratory sessions of 30 minutes and our industrial partner of 5 hours for both original and pruned GUI models.

In the first column (Original GUI), we have code coverage information from both applications when considering the original GUI model. That is, without trying to aid the exploratory tester about regions that are directly related to code changes. In this experiment, exploratory testers have only the usual information: manual inspection of both change request reports³ and release notes⁴ to see where they have to exercise the applications using their expertise. In the second column (Pruned GUI), coverage data is related to the exploratory testing using a simplified (pruned) GUI model that has the main characteristic of being directly related to code changed regions. In this situation, exploratory testers are allowed to use the same usual information as before as well as our pruned GUI model. As one can observe from these data, in both experiments we can find an increase in code coverage. In the Rachota experiment, which is a somewhat small application (with 63 classes and 105 changed methods), we almost doubled the code coverage data. For our industrial partner, which has an application with 162 classes and 531 changed methods, by using an adaptation of our solution, the increase in code coverage was not so impressive but it was enough to reveal 2 new bugs that were not found originally.

By comparing the GUI model before and after the pruning process, it is notable the reduction of the minimum amount of paths that needs to be exercised to cover the modified code regions, as depicted in Figures 16 and 17.

	Original GUI	Pruned GUI	Detected Bugs
Rachota	42.86%	71.43%	-
Industry	6.8%	9.75%	✖ ✖

Table 2 – Code coverage (related only to changed code) of exploratory testing

²We cannot reveal detailed information about this experiment due to Non-Disclosure Agreement (NDA) restrictions

³Bug tracker (<https://sourceforge.net/p/rachota/bugs/>) used to manage CRs related to Rachota

⁴Discussion list (<https://sourceforge.net/p/rachota/news/>) used to disclose the release notes of Rachota

5.2.1 Discussion

From Table 2, we can observe that the increase in code coverage was outstanding for the Rachota application but was very small for the industrial application. There are three main reasons these data appeared this way.

1. **Experience.** In the Rachota experiment, the same exploratory tester exercised the application before and after the pruning process. In the industrial experiment, due to difficulties in the real environment, we could only have a less experienced exploratory tester exercising the pruned GUI model;
2. **Amount of changes.** Rachota had a considerably small amount of code changes, only 105 methods. In the industrial experiment, which is 2.57 (162/63) bigger than Rachota in terms of number of classes, more than 531 methods were changed (> 5 times the number of modifications in Rachota) because industry takes more time to execute a new exploratory testing session and thus the number of changes increases considerably between testing sessions;
3. **Swing vs Android.** Our Soot patterns were created and implemented for Swing applications, to better compare with works in the literature. But our partner uses Android. Thus to perform the industrial experiment we had to manually prune the model based on changed code (new and modified methods). This took about 1 work week. Adapting our patterns to Android is one of our future work.

Although we can observe an increase in coverage in the experiment of our industrial partner, an important and worrying fact about these coverage data is that they are too low, even after the increase. We already showed these data to our industrial partner that is trying to adopt our proposed strategy as soon as possible as well as to decrease the time between exploratory testing periodic sessions, decreasing the amount of code changes to be tested as well. This shall increase code coverage and potentially detect more bugs. For

Figure 16 – Rachota GUI model after pruning

Figure 17 – Rachota GUI model before pruning

Figure 18 – Tooltip indicating how to execute event *e130* on Rachota

Rachota evaluation, Figure 16 show the resulting GUI model after applying the pruning process in the original GUI model, depicted in Figure 17.

5.2.2 Exemplifying the GUI model usage

During the second session of exploratory tests, the exploratory testers use the (pruned) GUI model as a complementary artifact to reach the changed code. We illustrate an example of the GUI model usage on Rachota application.

We restrict the scope of the example to some events to simplify our explanation. To facilitate understanding, we zoom in the GUI model to focus on event *e130*, that is one of the targets of our analysis. By hovering this event it is displayed a tooltip indicating how to execute this path like depicted in Figure 18.

This hint informs that in the window of type `MainWindow`, after clicking on a widget of type `JMenuItem`, it shall display a window of type `AboutDialog`. Thus, exploratory testers may associate this information with their previous knowledge about the application to reach the changed code. After following the hint on Rachota application we have the

Figure 19 – Executing event *e130* on Rachota

Figure 20 – About screen on Rachota

screen like displayed in Figure 19. By clicking on *About* menu item, it opens the About dialog (see Figure 20).

In the About dialog, by following the hint displayed over e_{242} (see Figure 21), it informs that after clicking on a JLabel (blue square on Figure 20), the About dialog is opened, that is, the focus keeps on the current displayed About dialog. One can think that it is an error on the GUI model the edge of event e_{242} to return to the About dialog, since clicking on the link opens a web page pointing to an *http://rachota.sourceforge.net* but this assumption it not true. As the model encompasses the scope of the Rachota application, as well as the About dialog does not dispose after clicking on the link, we may affirm that event e_{242} is correctly modeled.

Figure 21 – Tooltip indicating how to execute event e_{242} on Rachota

5.2.3 Threats to Validity

The first threat is the dependency between the increase in coverage and testers experiences. For instance, one can say that the impressive increase in the Rachota experiment is because the tester was the same person (before and after) and thus this can influence in the analysis because an experience is gained in the first exploratory session. But we think this is not directly the case because in the industrial experiment the tester using the pruned GUI model was less experienced than in the first experiment and even in this situation we can observe an increase in code coverage of the changed regions. We intend to perform a Latin square controlled experiment to more precisely confirm the potential gain of our proposal.

The second threat is the number of experiments we performed. We know that just two experiments are considerably little but (i) we tried to compare literature and industry and we could have only a single experiment from our industrial partner (such experiments are difficult to be obtained because they can interfere in the daily schedule of the company), and (ii) there is a logical reasoning behind these experiments that are independent of quantity. By focusing on a smaller region, it is logical that the same time used on an exploratory testing session tend to cover more code.

6 CONCLUSION

This chapter presents the conclusions about the present work, describing the main contributions, as well as discusses related works and the perspectives of future.

In this research, we propose a simple GUI model formalization and evaluate this model by applying it on more than 30 applications found in public repositories. Although this evaluation is a starting point for further analysis, the results demonstrate the applicability and potential contribution of proposed the GUI model. By following our proposed formalization about how to calculate the DC in a GUI model our GUI model builder can achieve (see Table 1) a high graph-connectivity degree for each application where some of them were full connected (100%). This can be a proof of concept that with a few more patterns we can obtain a generic GUI model builder based on static analysis.

The creation of the GUI model is performed by applying a proposed algorithm based on Swing Source Code Patterns described in terms of the Jimple intermediate Soot representation language. This allows us to extend our work easily as well as explore other frameworks by adjusting the patterns. In principle, the algorithm fits on other GUI frameworks without changes.

We focus on Swing applications just to better compare with other works found in the literature but our proposal can be used on other GUI frameworks as well, like Android; indeed one of our future work.

As another contribution, we propose a pruning strategy applied on created GUI model. The pruning algorithm is fed by changed code regions determined by examining two consecutive application versions (the current and the last tested one). Our goal was to provide a focused GUI model to exploratory testers that try to use the most recent modified regions to increase their chance to find bugs.

In Section 5.2, we illustrated our pruning strategy proposal using two representative experiments: one from the literature (the Rachota application) and another from our industrial partner (we cannot be more specific due to NDA restrictions). In both experiments, we observed an increase in code coverage related to the modified regions, where such an increase was more profound in the literature experiment because the amount of modified code was smaller. Our industrial partner, due to real environment difficulties related to people allocation, performs exploratory testing sessions with a huge amount of changed code in our opinion. We already pointed out our concerns (from the coverage data) to our partner and process adjustments are currently taking place. Although our industrial experiment was not so impressive in terms of code coverage, we found 2 bugs as a side-effect of this small increase in code coverage. And this for the industry is spectacular.

6.1 Related Work

Regarding the GUI model generation from static analysis, the solutions that come closest to our work are described in [53, 59]. In [59] the author uses another static approach for GUI analysis in order to help users in the program understanding by showing the GUI's structure and showing the flow of control caused by GUI events. That approach was validated in applications written in C or C++ which use some GUI library (for instance, GIMP Toolkit (GTK+) [25] or Qt [48]).

In the work reported in [53], the authors proposed the GUISurfer, a tool based on a language-independent approach to reverse engineering GUI code. It also uses a static approach, in this case, based on program transformation and program slicing, where they are used to extract the AST to build the GUI state machine. This approach requires that the code is generated by the NetBeans IDE and only works with “ActionListener” methods.

The approach described in [7] combines both black-box and white-box techniques to identify relevant test sequences. It uses a black-box approach, based on the work reported in [38], to build an event-flow graph. From this graph, it derives the executable test sequences, and then, via a static analysis approach based on program slicing, it eliminates redundant test sequences. Our work differs from this one because we use static analysis for all purposes. We choose this because of the current trend and that black-box GUI model building is limited by the state-space explosion problem.

The work reported in [41] presents TrimDroid, a framework for Android-based GUI testing. It uses static analysis to limit the number of widgets that should be taken into account, and consequently reducing the space of test sequences. Our work only compares to this one in the sense of using static analysis as well as Android is one of our future work.

The work [31] records app usages that yield execution (event) traces, mine those traces and generate execution scenarios using statistical language modeling, static and dynamic analyses. Finally, it validates the resulting scenarios using an interactive execution of the app on a real device. Our scenarios are independent of users initially because we focus on change requests. We also avoid validation of our GUI model because we rely on the experience of our exploratory testers when using our pruned GUI model.

In [23], it is proposed a way of improving the models used in MBT approaches by incorporating collected information during the exploratory testing activities. They introduce an approach and a toolset (called ARME) for automatically refining system models. This approach was validated in the context of three industrial case studies. In our approach, we perform the opposite process. We use a refined GUI model in order to assist

tester in the exploratory testing sessions. Our solution has more chances to cover code regions recently modified because the model is created based on both added and changed code. Our model acts as an aid in the discovery of application areas that are more likely to reveal bugs.

6.2 Future Work

To improve this work even more, these are some of the next steps that will be done in the near future:

- As the first future work, we want to integrate our two phases (building GUI model and pruning) in a single step.
- Another important future contribution is to improve the way of representing the hints displayed by the GUI model. We intend to represent a hint in a more user-friendly and non-ambiguous way.
- We intend to integrate our GUI model tool with model-based GUI test case generation tools created in our research group [8, 15, 45] to obtain systematic testing as well beyond exploratory testing.
- Another perspective is to adapt the proposal here to Android applications to be able to apply such a technology on Motorola Mobility testing, as consequently, to expand our experiments to large-used applications.

REFERENCES

- [1] Systems and software engineering – vocabulary. *ISO/IEC/IEEE 24765:2010(E)* (Dec 2010), 1–418.
- [2] ABEBE, S. L., ALI, N., AND HASSAN, A. E. An empirical study of software release notes. *Empirical Software Engineering* 21, 3 (2016), 1107–1142.
- [3] ANDROID. Android. <https://www.android.com/>. (accessed Mar 07, 2017).
- [4] ANDROID. Android open source project - issue tracker. <https://code.google.com/p/android/issues/entry>. (accessed Dec 07, 2016).
- [5] ARISS, O. E., XU, D., DANDEY, S., VENDER, B., MCCLEAN, P., AND SLATOR, B. A systematic capture and replay strategy for testing complex gui based java applications. In *7th ITNG* (Apr 2010), pp. 1038–1043.
- [6] ARLT, S., PODELSKI, A., BERTOLINI, C., SCHAF, M., BANERJEE, I., AND MEMON, A. Lightweight static analysis for gui testing. In *Proceedings of the 23rd IEEE International Symposium on Software Reliability Engineering* (Washington, DC, USA, 2012), ISSRE 2012, IEEE Computer Society.
- [7] ARLT, S., PODELSKI, A., AND WEHRLE, M. Reducing gui test suites via program slicing. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2014), ISSTA 2014, ACM, pp. 270–281.
- [8] ARRUDA, F., SAMPAIO, A., AND BARROS, F. Capture & replay with text-based reuse and framework agnosticism. In *SEKE* (2016), pp. 1–6.
- [9] ASM. A java bytecode manipulation and analysis framework. <http://asm.ow2.org>. (accessed May 26, 2016).
- [10] BACH, J. Exploratory testing explained. <http://www.satisfice.com/articles/et-article.pdf>. (accessed Nov 25, 2016).
- [11] BAE, G., ROTHERMEL, G., AND BAE, D.-H. Comparing model-based and dynamic event-extraction based gui testing techniques: An empirical study. *Journal of Systems and Software* 97 (2014), 15 – 46.
- [12] BELLI, F. Finite state testing and analysis of graphical user interfaces. In *Proceedings of the 12th International Symposium on Software Reliability Engineering* (Nov 2001), pp. 34–43.

- [13] BIRKELAND, J. O. *From a Timebox Tangle to a More Flexible Flow*. Springer Berlin Heidelberg, 2010, pp. 325–334.
- [14] BUGZILLA. <http://www.bugzilla.org/>. (accessed Dec 01, 2016).
- [15] CARVALHO, G., BARROS, F., CARVALHO, A., CAVALCANTI, A., MOTA, A., AND SAMPAIO, A. *NAT2TEST Tool: From Natural Language Requirements to Test Cases Based on CSP*. Springer International Publishing, 2015, pp. 283–290.
- [16] CAVALCANTI, Y. C., DO CARMO MACHADO, I., DA MOTA S. NETO, P. A., AND DE ALMEIDA, E. S. Towards semi-automated assignment of software change requests. *Journal of Systems and Software* 115 (2016), 82 – 101.
- [17] CHECKSTYLE. A development tool to help programmers write java code that adheres to a coding standard. <http://checkstyle.sourceforge.net/>. (accessed Jan 22, 2017).
- [18] COMET. Community event-based testing. <http://comet.unl.edu>. (accessed May 31, 2016).
- [19] DARAB, M. A. D., AND CHANG, C. K. Black-box test data generation for gui testing. In *2014 14th International Conference on Quality Software* (Oct 2014), pp. 133–38.
- [20] EINARSSON, A., AND NIELSEN, J. D. A survivor’s guide to java program analysis with soot. Tech. rep., 2008.
- [21] FINDBUGS. Find bugs in java programs. <http://findbugs.sourceforge.net/>. (accessed Jan 22, 2017).
- [22] FINSTERWALDER, M. Automating acceptance tests for gui applications in an extreme programming environment. In *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering* (2001), pp. 20–23.
- [23] GEBIZLI, C. Ş., AND SÖZER, H. Automated refinement of models for model-based testing using exploratory testing. *Software Quality Journal* (2016), 1–27.
- [24] GIMBLETT, A., AND THIMBLEBY, H. User interface model discovery: Towards a generic approach. In *Proceedings of the 2nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems* (New York, NY, USA, 2010), EICS ’10, ACM, pp. 145–154.
- [25] GTK+. The gtk+ project. <http://www.gtk.org/>. (accessed Dec 07, 2016).

-
- [26] GUIDE, T. B. Life cycle of a bug. <http://www.bugzilla.org/docs/2.18/html/lifecycle.html>. (accessed Dec 01, 2016).
 - [27] JAVA FX. Javafx overview. <http://docs.oracle.com/javase/8/javafx/get-started-tutorial/jfx-overview.htm>. (accessed Mar 07, 2017).
 - [28] JSF. Javaserer faces technology. <http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html>. (accessed Mar 07, 2017).
 - [29] KHEDKER, U., SANYAL, A., AND SATHE, B. *Data Flow Analysis: Theory and Practice*, 1st ed. CRC Press, Inc., Boca Raton, FL, USA, 2009.
 - [30] LAM, P., BODDEN, E., LHOTÁK, O., AND HENDREN, L. The soot framework for java program analysis: a retrospective. CETUS 2011.
 - [31] LINARES-VÁSQUEZ, M., WHITE, M., BERNAL-CÁRDENAS, C., MORAN, K., AND POSHYVANYK, D. Mining android app usages for generating actionable gui-based execution scenarios. In *Proceedings of the 12th Working Conference on Mining Software Repositories* (2015), IEEE Press, pp. 111–122.
 - [32] LINDHOLM, T., YELLIN, F., BRACHA, G., AND BUCKLEY, A. *The Java Virtual Machine Specification, Java SE 8 Edition*, 1st ed. Addison-Wesley, Upper Saddle River, NJ, 2014.
 - [33] LOURIDAS, P. Static code analysis. *IEEE Softw.* 23, 4 (July 2006), 58–61.
 - [34] LOY, M., ECKSTEIN, R., WOOD, D., ELLIOTT, J., AND COLE, B. *Java Swing*, 2nd ed. O’Reilly Media, 2002.
 - [35] MANTIS. Mantis bug tracker. <http://www.mantisbt.org/>. (accessed Dec 01, 2016).
 - [36] MARIANI, L., PEZZÈ, M., RIGANELLI, O., AND SANTORO, M. Autoblacktest: Automatic black-box testing of interactive applications. In *Proceedings of the 5th International Conference on Software Testing* (April 2012), pp. 81–90.
 - [37] MEMON, A. An event-flow model of gui-based applications for testing: Research articles. *Software Testing Verification and Reliability* 17, 3 (Sep 2007), 137–157.
 - [38] MEMON, A. M., BANERJEE, I., AND NAGARAJAN, A. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *Proceedings of The 10th Working Conference on Reverse Engineering* (Washington, DC, USA, Nov 2003), WCRE ’03, IEEE Computer Society, pp. 260–269.
 - [39] MESZAROS, G. Agile regression testing using record & playback. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming*,

- Systems, Languages, and Applications* (New York, NY, USA, 2003), OOPSLA '03, ACM, pp. 353–360.
- [40] MIAO, Y., AND YANG, X. An fsm based gui test automation model. In *2010 11th International Conference on Control Automation Robotics Vision* (Dec 2010), pp. 120–126.
- [41] MIRZAEI, N., GARCIA, J., BAGHERI, H., SADEGHI, A., AND MALEK, S. Reducing combinatorics in gui testing of android applications. In *Proceedings of the 38th International Conference on Software Engineering* (2016), ICSE '16, ACM, pp. 559–570.
- [42] MORENO, L., BAVOTA, G., PENTA, M. D., OLIVETO, R., MARCUS, A., AND CANFORA, G. Automatic generation of release notes. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2014), FSE 2014, ACM, pp. 484–495.
- [43] MORGADO, I. C., PAIVA, A. C. R., AND FARIA, J. P. Reverse engineering of graphical user interfaces. In *Proceedings of 6th International Conference on Software Engineering Advances* (2011), ICSEA 2011, pp. 293–298.
- [44] NGUYEN, D. H., STROOPER, P., AND SÜSS, J. G. Automated functionality testing through guis. In *Proceedings of the Thirty-Third Australasian Conferenc on Computer Science - Volume 102* (Darlinghurst, Australia, Australia, 2010), ACSC '10, Australian Computer Society, Inc., pp. 153–162.
- [45] NOGUEIRA, S., SAMPAIO, A., AND MOTA, A. Test generation from state based use case models. *Formal Aspects of Computing* 26, 3 (2014), 441–490.
- [46] PAIVA, A. C. R., FARIA, J. C. P., AND MENDES, P. M. C. *Reverse Engineered Formal Models for GUI Testing*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 218–233.
- [47] PMD. A source code analyzer. <https://pmd.github.io/>. (accessed Jan 22, 2017).
- [48] QT. Cross-platform software development for embedded & desktop. <http://www.qt.io/>. (accessed Dec 07, 2016).
- [49] RACHOTA. A portable application for timetracking different projects. <http://rachota.sourceforge.net/>. (accessed May 26, 2016).
- [50] REDMINE. Flexible project management web application. <http://www.redmine.org/>. (accessed Dec 01, 2016).
- [51] SHARAN, K. *Beginning Java 8 APIs, Extensions and Libraries: Swing, JavaFX, JavaScript, JDBC and Network Programming APIs*. Apress.

- [52] SILVA, J. C., SARAIVA, J., AND CAMPOS, J. C. A generic library for gui reasoning and testing. In *Proceedings of the 2009 ACM Symposium on Applied Computing* (New York, NY, USA, 2009), SAC '09, ACM, pp. 121–128.
- [53] SILVA, J. C., SILVA, C., GONÇALO, R. D., SARAIVA, J., AND CAMPOS, J. C. The guisurfer tool: Towards a language independent approach to reverse engineering gui code. In *Proceedings of the 2Nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems* (New York, NY, USA, 2010), EICS '10, ACM, pp. 181–186.
- [54] SOOT. A framework for analyzing and transforming java and android applications. <https://sable.github.io/soot/>. (accessed Nov 22, 2016).
- [55] SOURCEFORGE. An open source community resource dedicated to helping open source projects. <http://sourceforge.net/>. (accessed May 26, 2016).
- [56] SPILLNER, A., LINZ, T., AND SCHAEFER, H. *Software Testing Foundations: A Study Guide for the Certified Tester Exam*, fourth edition ed. Rocky Nook Computing. Rocky Nook, 2014.
- [57] SPIVEY, J. M. *The Z Notation: A Reference Manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.
- [58] SPRING MVC. Web mvc framework. <https://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html>. (accessed Mar 07, 2017).
- [59] STAIGER, S. Reverse engineering of graphical user interfaces using static analyses. In *Proceedings of the 14th Working Conference on Reverse Engineering* (Oct 2007), WCRE 2007, pp. 189–198.
- [60] STEVEN, J., CHANDRA, P., FLECK, B., AND PODGURSKI, A. jrapture: A capture/replay tool for observation-based testing. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis* (New York, NY, USA, 2000), ISSTA '00, ACM, pp. 158–167.
- [61] SWING. Creating a gui with jfc/swing. <http://docs.oracle.com/javase/tutorial/uiswing/index.html>. (accessed Mar 07, 2017).
- [62] SWT. Swt: The standard widget toolkit. <https://www.eclipse.org/swt/>. (accessed Mar 07, 2017).
- [63] TESTLINK. Testlink open source test management. <http://testlink.org/>. (accessed Dec 05, 2016).
- [64] THE JAVA™ TUTORIALS. How to use root panes. <https://docs.oracle.com/javase/tutorial/uiswing/components/rootpane.html>. (accessed Mar 22, 2016).

-
- [65] UMANEE, N. Shimple: An investigation of static single assignment form. Master's thesis, McGill University, Feb 2006.
 - [66] UTTING, M., AND LEGEARD, B. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.
 - [67] VALLÉE-RAI, R., AND HENDREN, L. J. Jimple: Simplifying java bytecode for analyses and transformations. Tech. Rep. TR-1998-4, McGill University, 1998.
 - [68] WHITTAKER, J. *Exploratory Software Testing: Tips, Tricks, Tours, and Techniques to Guide Test Design*. Pearson Education, 2009.
 - [69] XIE, Q., AND MEMON, A. M. Model-based testing of community-driven open-source gui applications. In *2006 22nd IEEE International Conference on Software Maintenance* (Sept 2006), pp. 145–54.