Bridging the gap between GUI functional system testing and functional system logic testing

by

Abdulaziz Alkhalid

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Carleton University
Ottawa, Ontario

© 2018, Abdulaziz Alkhalid
Abstract

Graphical User Interface (GUI) accounts for half or more of the amount of source code in modern software. Software verification of GUI-based software has been discussed in literature. GUI testing and system testing are two tools used in software verification of GUI-based software. Static analysis is a tool for software verification that can study relationships between inputs and executed parts of software code. The thesis bridges the gap between system testing and GUI testing through static analysis. Specifically, it introduces the use of static analysis combined with system tests to verify the GUI-based Software Under Test (SUT). The static analysis will be applied on the source code of the GUI part of the SUT. System tests will be applied on the source code of the non-GUI part of the SUT.
Acknowledgements

I would like to thank my thesis advisor, Dr. Yvan Labiche.

I would also like to thank members of the examination committee (Dr. Tanja Vos, Dr. Guy-Vincent Jourdan, Dr. Douglas Howe and Dr. Dorina Petriu).

I thank people from the Department of Systems and Computer Engineering.

I thank my friends from Carleton University.

Many thanks to all people who assisted me in finishing this work.
Dedication

To my father, Muhammad Alkhalid.

To the memory of my mother, Halimeh Algori.
# Table of Contents

Chapter 1  Introduction  1

1.1 Preliminary ................................................................. 1
1.2 The scope of the thesis.......................................................... 2
1.3 The problem statement ......................................................... 3
1.4 Research questions and proposed solution ................................. 4
1.5 Contributions ..................................................................... 7
1.6 Organization .................................................................... 9

Chapter 2  GUI testing and system testing definitions and related work  10

2.1 Software testing definitions ..................................................... 11
  2.1.1 System testing................................................................. 13
  2.1.2 GUI testing .................................................................... 16
  2.1.3 A taxonomy of terms pertaining to testing of GUI-based software ...... 19
  2.1.4 Mapping existing work with our taxonomy ................................ 24
  2.1.5 Conclusion .................................................................... 27
2.2 Entity Control Boundary (ECB) design principle ............................ 29
2.3 Boundary changes the syntax but not semantics ............................. 30
2.4 Literature review .................................................................... 31
  2.4.1 Recent surveys on GUI functional system testing/GUI functional testing.. 32
  2.4.2 Test suite construction....................................................... 34
  2.4.3 GUITAR........................................................................ 37
  2.4.4 GUI functional system testing using GUITAR .......................... 39

Chapter 3  Background and related work on static verification  44
Appendix A  The complete list of books, both from the research laboratory and the Library
Appendix B  Steps of GUI testing (Demo of using GUITAR on our case study)
Appendix C  Measuring coverage of automated GUI-testing
Appendix D  Steps for conducting GUI-testing on Boolean logic software using Replayer v.1.1
Appendix E  Integrating GUITAR and COBERTURA with Ant for GUI-testing on Boolean logic software using GUITAR 1.2
Appendix F  Steps for measuring coverage when using Replayer v.1.1
Appendix G  Measuring coverage using GUITAR 1.2 and COBERTURA 2.1
Appendix H  Clicking the push button on the SUT in GUI testing
Appendix I  Automated GUI testing using GUITAR -lessons learnt
Appendix J  The use of Marathon
Appendix K  Functional test suite description
Appendix L  GUI testing experimental results
Appendix M  Using COBERTURA in the case study
List of Figures

Figure 1. Functional and non-functional system testing .................................................. 3

Figure 2. Verification of input-output relation ................................................................. 6

Figure 3. Relationship among different testing types ....................................................... 20

Figure 4. GUI functional system testing using GUITAR .................................................. 41

Figure 5. An example of software slicing [109] (Statements belonging to the same slice are in underlined font) ............................................................................................................... 47

Figure 6. Functions of abstract interpretation [118] ......................................................... 52

Figure 7. Code that swaps two integers and symbolic execution tree [15, 20-23] .......... 54

Figure 8. Relationship between GUI functional system testing and functional system logic testing in the context of the thesis ................................................................. 73

Figure 9. Main window of the Boolean logic software application ............................... 75

Figure 10. Truth table window of the Boolean logic software application ................. 75

Figure 11. DNF expression window of the Boolean logic software application .......... 76

Figure 12. Variable negation window of the Boolean logic software application ....... 77

Figure 13. Line coverage vs. Boolean expression in experiment A (for details see Appendix L) ......................................................................................................................... 97

Figure 14. Value of diff_coverage System testing (I) - GUI functional system testing (E) ............................................................................................................................ 108

Figure 15. GUI verification ............................................................................................... 115

Figure 16. The class diagram of the simple software ..................................................... 131

Figure 17. The main window of simple software after typing the input and clicking the copy button ........................................................................................................... 132
Figure 18. The input-output illustration diagram for the use case of entering input and clicking the copy button ................................................................. 133
Figure 19. Dummy case study .................................................................................. 137
Figure 20. Window of 1-1 relation with string input .................................................. 137
Figure 21. Window of 1-N relation ............................................................................ 139
Figure 22. Window of N-1 relation ............................................................................ 140
Figure 23. Passing input from GUI to logic ............................................................... 161
Figure 24. Visual representation of EFG for Boolean logic application using Gephi .... 198
Figure 25. The Arrangement of folders in framework that can be used for coverage measuring after GUI-testing ................................................................. 202
Figure 26. GUI of MyFrame program ....................................................................... 204
Figure 27. Truth table tab of Boolean logic application ............................................ 218
Figure 28. DNF expression tab of Boolean logic application .................................... 218
Figure 29. Variable negation tab of Boolean logic application .................................. 219
Figure 30. The new GUI of the SUT ......................................................................... 240
Figure 31. Manual test process (left) and capture/replay testing process (right) [215] .. 242
Figure 32. Marathon start window .......................................................................... 244
Figure 33. Project settings ....................................................................................... 245
Figure 34. Marathon control center ........................................................................ 245
Figure 35. Saved test case ...................................................................................... 247
Figure 36. Playing the script ................................................................................... 247
Figure 37. The generated report .............................................................................. 248
Figure 38. Our framework for GUI functional system testing by integrating GUITAR [8] and COBERTURA [187].
List of Tables

Table 1. Classification results ........................................................................................................ 25
Table 2. Justification of classification results ................................................................................ 26
Table 3. Classes and packages of the SUT ..................................................................................... 74
Table 4. Details of the experimental plan ....................................................................................... 84
Table 5. Line and Branch coverage for experiments F, G, H and I (class numbers are used later in the text to refer to specific classes instead of their names) ........................................ 94
Table 6. Total Line and Branch coverage for experiments A ......................................................... 98
Table 7. Total Line and Branch coverage for experiments B ......................................................... 98
Table 8. Total Line and Branch coverage for experiments C ......................................................... 99
Table 9. Experiment C, coverage of GUI functional system testing for variable negation functionality .............................................................................................................................. 100
Table 10. Total Line and Branch coverage for experiment D ....................................................... 101
Table 11. Detailed results for experiment D, coverage of GUI functional system testing for three operations experiments ................................................................. 101
Table 12. Line and Branch coverage for experiment E .................................................................. 103
Table 13. Line and Branch coverage for all the experiments ......................................................... 107
Table 14. Line and Branch coverage for Control classes .............................................................. 110
Table 15. The part of source code of GUI that changes the input value ......................................... 132
Table 16. The adding of verification variable in the source code .................................................. 135
Table 17. The adding of a precondition in the source code ............................................................ 135
Table 18. Part of the content of output file generated by ESC ...................................................... 135
Table 19. JML code to verify the 1-1 relation ................................................................................. 137
Table 20. JML code to verify the 1-N relation ............................................................................... 139
Table 21. JML code to verify the N-1 relation ................................................................. 140
Table 22. The adding of Julia watchpoint in the source code................................. 159
Table 23. The content of the file NumericalInvariant.pl generated by Julia .......... 160
Table 24. XML for the first level of GUI of the Boolean logic application ............. 194
Table 25. A representation of a widget (JPanel) of Boolean logic application ....... 195
Table 26. XML representation for EFG of the Boolean Logic application ......... 196
Table 27. Part of Event Flow Graph's XML file for Boolean logic application ....... 197
Table 28. Test case 5 generated by GUITAR for Boolean logic application ........... 198
Table 29. The state trace file content ........................................................................... 200
Table 30. Source code of simple GUI program .......................................................... 205
Table 31. Coverage report generated by COBERTURA ............................................. 206
Table 32. Detailed coverage report generated by COBERTURA ......................... 207
Table 33. Observations with a simple GUI app, test case length = 3, button does not get enabled till text is entered in text field ........................................................................ 214
Table 34. Observations with Boolean logic software, test case length = 3 .......... 215
Table 35. Description of source code of the case study ............................................. 217
Table 36. Coverage of all classes during ripping on the Boolean logic application .... 230
Table 37. Adding data input to test case ...................................................................... 234
Table 38. Description of functional system logic test suite ................................. 249
Table 39. Truth table coverage information ............................................................... 251
Table 40. The DNF component coverage information ............................................. 252
Table 41. The variable negation coverage information ............................................. 253
Table 42. The three operation coverage information .............................................. 254
Table 43. The expressions from Internet coverage information .......................... 255
Chapter 1   Introduction

Advances in technology used as platforms for Graphical User Interface (GUI) software lead to more complex, platform-independent GUI-based software. Current GUI software is capable of serving different types of users with different levels of abilities (e.g., ordinary user, user with disability, web user or mobile user). The rest of this chapter is organized as follows. Section 1.1 presents a preliminary. Section 1.2 describes the scope of the thesis. Section 1.3 presents our problem statement. Section 1.4 presents our research questions and the proposed solution. Section 1.5 presents the contributions. Section 1.6 describes the organization of the thesis.

1.1 Preliminary

These advances in technology produce challenges for software testers who are responsible for software verification of those GUI-based software. The advances in mobile phone technology motivate companies to develop new GUIs to present their software services to web users and mobile phone users. The new developed software needs to be tested, though it provides the same functionalities but for a different user. Consequently, software testers find themselves with several testing types to choose from and use, such as GUI testing and system testing. We argue that having clear definitions for these testing types, including GUI testing, would allow software practitioners to have consensus on stopping criteria for testing of a GUI-based software application, including the testing of the GUI and the testing of the business logic, and hence avoiding the effort of testing underlying functionality of the software twice: for instance, once when testing
the software with a desktop GUI and a second time when testing the software with web GUI implementations. Another source of duplication of effort could be that one tests the logic of software through the UI as well as directly by bypassing the UI.

1.2 The scope of the thesis

We focus on desktop software which is motivated by Forrester and Miller who, through an empirical study of the reliability of GUI software for Windows NT [1, 2], show that a large number of software failed to deal reasonably with random input, and hence recommend production of more robust code. Another reason for restricting the scope of our study to desktop software is the need for improved, more cost-effective verification technologies for GUI-based desktop software. In our experience, for a trivial GUI with one text field, two radio buttons and one push button (i.e., from GUI elements), using the GUI testing tool GUITAR [3] with its default configuration is not practical as it is too expensive.

Figure 1 shows the focus of this thesis. It illustrates the definitions among two main testing types and also shows the software is divided into its GUI layer and its application logic layer. Note that this is in line with prominent software design methods that promote a separation of concerns between GUI code and application logic code (e.g., Gomaa’s notion of Boundary, Control and Entity classes for object-oriented software [4]). The figure illustrates that system testing can focus on the functional aspects of the Software Under Test (SUT) (referred to as functional system testing in the figure) or the non-functional aspects of the SUT, sometimes referred to as the “alities” (referred to as non-functional system testing). Both kinds of system testing can trigger only the GUI (an
arrow stops at the GUI layer), the GUI and the underlying application logic layer (arrow to the GUI layer, going through the GUI as a dashed line and triggering the application logic layer), or only the application logic layer. The figure also shows that our scope (the non-greyed-out part), is limited to functional system testing and does not deal with the “alities” of the SUT. When functional system testing is applied through the GUI, we call it GUI system testing in order to distinguish it from functional system testing applied to the application logic directly.

**Figure 1. Functional and non-functional system testing**

1.3 The problem statement

System testing can be defined as a phase of testing conducted on the complete software to evaluate its compliance with its requirements, be they functional or non-functional [5]. However, there is a confusion about alternative definitions of GUI testing one can find in the literature. For example, Ammann and Offutt classified GUI testing into usability testing and functional testing, and further classified the latter into GUI system testing, regression testing, input validation testing and GUI testing [6]. They
argue that GUI system testing is system testing of the entire software through its GUI. They explain that GUI testing is verifying that the GUI works correctly without verifying the underlying application code. Memon et al. defined GUI testing as system testing for software that has a graphical user interface [7]. We observe that Memon’s notion of GUI testing encompasses both notions of GUI testing and GUI system testing of Ammann and Offutt.

There is no clear agreement in the research community about what GUI testing is, at least as inferred from definitions we find in published work.

We conclude that the engineer interested in testing a GUI-based software may found himself lost in the many alternative definitions of GUI testing that exist and their relation to the notion of system testing. For instance, using Memon’s definition of GUI testing, one can use a tool like GUI Testing frAmewoRk (GUITAR) [8] to trigger both the GUI and the underlying functionalities, whereas when using Ammann and Offutt’s definitions one can use JUnit to directly test the application code, bypassing the GUI, and verify the GUI separately. One risk of using incompatible definitions for GUI testing and system testing is to duplicate testing effort: one conducts system testing of the application logic by bypassing the GUI and conducts GUI testing of the software with GUITAR [3] thereby testing the application logic twice.

1.4 Research questions and proposed solution

The first research question of this thesis is then to analytically study how available definitions for system testing/GUI testing relate to each other. We ask: What are the available definitions for system testing/GUI testing and how do they relate to each
other? We complement this analytical study with an experiment on a second research question which is to study how GUI system testing of the software logic compare to functional system testing applied to the logic by bypassing the GUI (in terms of structural coverage). We ask: Is GUI system testing of the software logic equivalent to functional system testing applied to the logic by bypassing the GUI in terms of structural coverage? We notice analytically and experimentally that there is a difference between system and GUI testing. This leads us to the first contribution of this thesis: a taxonomy of terms that clarify different types of testing activities that can be conducted on GUI-based software. For instance, the taxonomy distinguishes between the testing of the UI and testing of the application logic.

We also notice experimentally that testing with a GUI testing tool like GUITAR, a prominent GUI testing tool, appears too expensive to be practical. Simultaneously, Adamoli et al. [9] conclude that capture and replay tools evaluated during the survey of 50 papers on automated GUI testing are unable to capture realistic interactions with real-world software. Given GUI testing and system testing are two different activities, given the cost of system testing through the UI, and given limitations of capture and replay tools, we ask whether it is possible to verify separately the UI and the application logic. Specifically, we ask a third research question: Is it possible to verify a GUI-based software using static analysis (no execution) of the GUI code combined with system tests executed on the application logic? We therefore devise a verification procedure that recognizes differences between testing of the UI and testing of the application logic, avoids the cost and limitation of capture and replay tools.
We argue this is feasible because GUI-based software relies typically on the Entity-Control-Boundary design principle that separates the UI concern from the application logic concern. With the Entity-Control-Boundary principle [10], an Entity class realizes the data structure, a Control class takes care of the logic of software and a Boundary class deals with the communication with an actor. According to design principles, Boundary classes are not supposed to change the semantics of the data they receive, perhaps only changing the types of those data before providing that data to a Control class. A Boundary class gets inputs and provides outputs, likely without much data computations (only change of types). We use the term input-output relation to refer to the relation between inputs provided to a Boundary class and arguments passed to a method defined in a Control class as illustrated in Figure 2. We formulate the following question: What technique can be used to verify the GUI part of the software, specifically verify such input/output relations.

**Figure 2. Verification of input-output relation**
The figure shows that static analysis will be applied to the GUI. In other words, the static analysis will be used to verify the aforementioned input-output relation, thereby confirming that the Boundary classes do not change the semantics of data they transmit from the user to Control classes. This leads us to the main thesis contribution of this manuscript. The manuscript shows how this thesis can be realized with existing verification technologies, though there are technical limitations. If static analysis confirms this, then we have an indication that the right data reach Control classes, i.e. the application logic, and if system tests directed to Control classes pass, then we can have some confidence the whole software works as intended.

1.5 Contributions

To summarize, the thesis contribution is to answer the following research questions:

Research Question 1. What are the available definitions for system testing/GUI testing and how do they relate to each other?

Research Question 2. Is GUI system testing of the software logic equivalent to functional system testing applied to the logic by bypassing the GUI in terms of structural coverage?

Research Question 3. Is it possible to verify a GUI-based software using static analysis of the GUI code combined with system tests on the application logic?

The result of each research question listed above is a contribution. The first contribution is to provide a taxonomy of definitions. In order to achieve this contribution, we surveyed the literature and found books on software testing. We looked for definitions
of GUI testing and system testing. We then presented the first level of our definitions. We refined those definitions to have a new second level of fine-grain definitions that we call a taxonomy. We evaluated the taxonomy by checking its applicability on published work. In other words, how a published work fits into one definition in the taxonomy. The second is to compare two testing types in terms of test coverage. To conduct such comparison, we experimentally compare GUI functional system testing using GUI testing tool called GUITAR with functional system logic testing using JUnit test cases. We use a case study software in the experiment. Result shows that functional system logic testing overcomes GUI functional system testing. The third is to present a method for the verification of GUI-based software through applying functional system testing on the software application logic code directly, i.e., bypassing the GUI, and using static analysis of the functions of the GUI. We think that this combination of two verification techniques is sufficient to verify the entire GUI-based software. This is the core contribution of the thesis. The reason is that with the way of verification that we present, there is no need for GUI functional system testing. GUI functional system testing is a very expensive testing activity in terms of time and effort as the manuscript and appendixes describe later. Instead of doing GUI functional system testing, we verify the GUI layer using Java Modeling Language (JML) preconditions and a static checker such as the Extended Static Checker (ESC), and we test the underlying functionally using functional system logic tests implemented using JUnit library.

A set of papers out of this thesis were published. Chapter 4 is published in one paper: Alkhalid, A. and Y. Labiche, How does GUI testing exercise application logic functionality?, in STA 2017 in conjunction with 2017 IEEE 41st Annual Computer
1.6 Organization

Chapter 2 surveys possible definitions of GUI testing and system testing and shows how they relate to one another. In doing so, we reveal a risk of redundancies in testing effort between GUI testing and system testing. The chapter also introduces definitions we will rely on in the remainder of the manuscript, present a taxonomy and classify published work based on this taxonomy.

Chapter 3 provides background on static verification.

Chapter 4 reports on an experiment whereby GUI system testing and functional system testing are compared in terms of structural coverage on the case study, which confirms the risks of redundant efforts when using overlapping definitions of GUI testing. We use GUITAR, a well-known GUI testing software, to conduct GUI testing.

Chapter 5 describes work we conduct as part of this thesis on the use of static analysis on the GUI code, combined with system tests on the software logic code, to verify a GUI-based software.

Chapter 6 concludes this manuscript.
Chapter 2  GUI testing and system testing definitions and related work

This chapter presents a set of definitions that we collected from literature. We analyze those definitions. We show the lack of clarity in some major definitions. For instance, we can find two authors each with his own definition of a single term. For example, does GUI system testing test the GUI only? or the GUI and only the underlying functionally that is accessed through the GUI? We show how two definitions may agree or disagree on testing a specific aspect of a GUI-based software.

We use our understanding of those definitions available in the literature to create our own definitions. These new definitions cover a wide range of testing types. By having them, we present a first level of definitions but with no clear separation among testing types. In other words, a different testing practice may go under the same type (e.g. functional system testing) which does not make them perfect ones. Based on the first level of our definitions, we define a taxonomy. We use the taxonomy to classify the published work. By doing such classification, we argue that our taxonomy is applicable.

In this chapter, we first discuss definitions of GUI testing and system testing in section 2.1. We discuss then the Entity-Control-Boundary design principle in section 2.2. We give more details about responsibilities of a Boundary class in section 2.3. We then discuss the related work on GUI testing in section 2.4. We first note that, to the best of our knowledge, we do not know of any attempt in the literature to compare system testing and GUI testing the way we do it. That is why we could not discuss or compare it to any
literature in this chapter. In this chapter, we present some definitions of system testing, GUI testing, and other testing activities. As discussed below, these definitions warrant the study of differences (if any) between system testing and GUI testing.

2.1 Software testing definitions

In this section, we investigate definitions of a couple of testing activities, specifically system testing and GUI testing. The intent of this section is not to report on a systematic mapping study on GUI testing definitions and other testing definitions. We simply report on representative definitions of main software testing terms to answer Research Question 1: What are the available definitions for system testing/GUI testing and how do they relate to each other?

As mentioned before (Chapter 1), we justify our motivation to answer this question that having a clear definitions for system/GUI testing types will allow software testers to have consensus on stopping criteria for testing of a GUI-based software application, including testing the business logic with/without testing of the GUI, and hence avoiding the redundant effort of testing underlying functionality of the software twice: for instance, once when testing the software with a mobile phone GUI and a second time when testing the software with desktop GUI implementations.

We used a systematic method, though not a systematic literature review or systematic mapping study, to identify relevant definitions. The method started by identifying books available in the Software Quality Engineering Laboratory (SQUALL) at Carleton University and Carleton University Library in the area of software engineering and software testing. In the case of library books, this meant using the
Library search engine to identify books using the following keywords: testing, software GUI testing, software verification, GUI testing. We then identified chapters from those books which discuss software testing, and in particular GUI testing, by browsing through the tables of contents and skimming through pages, looking for keywords like “GUI testing” or “system testing.” We identified a total of 52 textbooks. The complete list of books can be found in Appendix A. We believe that, for our search for definitions, consulting textbooks is an adequate procedure, rather than for instance searching in academic paper databases.

We nevertheless surveyed by searching online resources, i.e., Google Scholar, IEEE Xplore, Science Direct, ACM digital library, Engineering Village and Scopus, using the following search strings: Graphical User Interface Testing, GUI testing, GUI testing and system testing, definition of GUI testing, Oracle for GUI testing, GUI testing tools, automated GUI testing, survey of GUI testing, GUI testing taxonomy. This step was necessary to find recent surveys or taxonomies in the area of GUI testing. This allowed us to identify a recent (2013) systematic mapping study on GUI testing [7]. We used the Digital Bibliography and Library Project (DBLP) Computer Science Bibliography [11] to look for publications related to GUI testing when we needed to search for a specific author.

The rest of this section is structured as follows: Subsection 2.1.1 reports on definitions about system testing. Subsection 2.1.2 presents definitions of GUI testing. Subsection 2.1.3 presents our taxonomy and the definitions we will rely on in the reminder of this manuscript. Subsection 2.1.4 uses the taxonomy to map a representative
sample of existing research activities and tools on testing of GUI-based software. Subsection 2.1.5 concludes.

2.1.1 System testing

System testing is defined as a “testing phase conducted on the complete integrated system to evaluate the system compliance with its specified requirements on functional and non-functional aspects” [5]. This definition is in accordance with other authors’ definitions [5, 12-17], with the Institute of Electrical and Electronics Engineers (IEEE) definition of software system testing [18], as well as with the Software Engineering Body of Knowledge (SWEBOK) guide [19].

Ammann and Offutt define system testing as deriving tests from external descriptions of the software, including specifications, requirements and design [6]. Naik and Tripathy define functional system testing as deriving tests that verify the system as thoroughly as possible over the full range of requirements [20]. These requirements are usually specified in the requirements specification document, including requirements about the GUI, communication systems, logging and tracing, and security [20]. Lewis defines black-box functional testing as a way to test conditions on the basis of the program or system's functionality; consequently, a tester requires information about the input data and observed output, but does not know how the program or system works [21] [page 39].

System testing evaluates the functionality and performance of the whole application. Beside evaluating the functional requirements of the application, system testing consists of a variety of tests including the following [21] [page 233]: Performance
testing, which measures the system against predefined objectives by comparing the actual and required performance levels; Security testing, which evaluates the presence and appropriate functioning of the security of the application to ensure the integrity and confidentiality of the data; Stress testing, which investigates the behaviour of the system under conditions that overload its resources and the impact this has on the system processing time; Compatibility testing, which tests the compatibility of the application to interact with other applications or systems; Conversion testing, which investigates whether the software is robust to changes of data formats; Usability testing, which decides how well the user is able to use and understand the application; Documentation testing, which verifies that the user documentation is accurate and ensures that the manual procedures work correctly; Backup testing, which verifies the ability of the system to back up its data so as to be robust to software or hardware failure; Recovery testing, which verifies the system’s ability to recover from a software or hardware failure; and Installation testing, which verifies the ability to install the system successfully. This taxonomy of system test activities can be divided further [20, 21]. However, the possible extension of the taxonomy is out of the scope of this thesis.

System testing is also said to exercise system-level behaviour, triggering behaviour from a system-level input, through the software, to a system-level output [17] [page. 191].

We conclude that in the case of GUI-based software, the GUI is one element of the "complete integrated system" with GUI-specific functional and non-functional requirements. We therefore conclude from previous definitions that system testing includes the verification testing of the GUI when there is one.
Last, acceptance testing and usability testing are two types of testing which the reader may think are related to the GUI. As such, we investigated these two types of testing, too. For instance, we are interested in acceptance testing since, being primarily conducted by the user/client of the software under test, this activity primarily involves the GUI in the case the SUT has a GUI. However, though their naming seems related to system testing, acceptance testing [5, 22] and usability testing [23] are different from system testing.

Acceptance testing [5, 22] is typically conducted by the customers or their representatives, who define a set of test cases that will be executed to qualify and accept the software product according to acceptance criteria [20]. The set of tests is usually a subset of the set of system tests [14, 21], including both functional and non-functional tests. As a consequence, since system testing, according to previous definitions, includes the verification of the GUI, acceptance testing does also involve some verification of the GUI [13].

In addition to its role in exposing faults to be removed to improve a product, system testing assesses whether software is ready to be released. Acceptance testing, meanwhile, evaluates the actual usefulness and usability rather than conformance to a requirements specification [22]. System and acceptance testing help make a decision whether or not to release the product.
2.1.2 GUI testing

While searching for definitions of GUI testing, the general observation we can make is that most of the authors do not provide a clear definition of GUI testing. This is not the case for other types of testing, even those that are close to the notion of GUI testing, such as system testing. Few published papers in the literature defined GUI testing as system testing.

GUI testing can be defined as system testing for software that has a GUI [3, 7, 24] (i.e., it is system testing of the entire software performed through its GUI). Tests are then sequences of events developed to exercise the GUI’s widgets (e.g., text fields, buttons and dropdown lists) [7, 24]. Similarly, Grilo et al. defined GUI testing as an activity for increasing confidence in the software application and its correctness by finding defects in the GUI itself or the whole software application [25].

Assuming the standard, IEEE definition of system testing we already discussed, we argue that tests of Memon et al. [3, 7, 24] do not address performance, usability, safety, installation, or other “alities” (section 2.1.1). Except perhaps for some robustness tests, which may incidentally be created by GUITAR, Memon’s notion of GUI testing is more about functional characteristics of the GUI-based software than its non-functional characteristics. This illustrates a major difference between system testing and Memon’s definition of GUI testing.
According to Ammann and Offutt, determining whether the GUI of a GUI-based software behaves as expected includes usability testing and functional testing [6]¹. The former refers to the assessment of how usable the interface is according to principles of user interface design. The latter refers to whether the user interface works as intended.

During usability testing, individuals, representing targeted groups of potential users of the software product, conduct the evaluation with respect to usability criteria [23].

Ammann and Offutt further classified functional testing in this context into four categories: GUI system testing, regression testing, input validation testing and GUI testing. GUI system testing refers to “the process of conducting system testing through the GUI.” Regression testing is about “testing of GUI after changes are made” [6]. We note the authors do not specify whether these changes are made to the user interface only, the logic of the software, or both of them. Input validation testing aims to verify whether the GUI “recognize[s] the user input and respond[s] correctly to invalid input” [6]. This is similar to robustness testing, which has been defined by the IEEE as a test to measure the degree to which a system or component can function correctly in the presence of invalid input”[18]. In this decomposition of functional testing in the context of a GUI-based software, GUI testing (the last of the four categories) is about assessing whether the GUI works – that is, whether the user interface (UI) controls work and allow the user of the UI to navigate between screens.

¹ Ammann and Offutt discuss that usability testing and functional testing are the two activities of GUI testing. They then split functional testing into four categories, including GUI testing, which results in a circular definition of the notion of GUI testing. We believe this circular definition was not intentional. To avoid this circular definition, we rather write that usability testing and functional testing are the two activities involved in determining whether the GUI of a GUI-based software behaves as expected.
We first notice that Ammann and Offutt’s definitions do not account for alternative non-functional requirements of the UI to usability and robustness (input validation), which also need to be verified. Also, well-known discussions about testability and sensitization [26, 27] tell us that exercising the SUT through its UI (GUI system testing) and verifying the UI itself (GUI testing) would not be sufficient to ensure the entire SUT behaves as expected. Some verification of the application logic itself would be necessary.

Contrasting Ammann and Offutt’s definition to Memon’s definition, we see that the notion of GUI testing by Memon is identical to the notion of GUI system testing by Ammann and Offutt, except with regards to non-functional requirements.

The top part of Figure 3 illustrates the main definitions we have encountered in our survey and that we just discussed (the “orange” arrows are discussed next).

In red the figure illustrates Ammann & Offutt's definitions. GUI testing is about the functional aspects of the GUI, focusing only on the UI layer, so the arrow goes to the functional part of the GUI and stops there. From their definitions, we do not have evidence that GUI testing also focuses on non-functional characteristics, especially since usability testing is a separate activity in their discussion. Usability testing is about an "ality" so the arrow goes to the "alities" part of the GUI and stops there. GUI system testing is system testing through the UI so arrows go to the UI (both functional and non-functional) and go through to the application logic. We already discussed that Memon’s definition of GUI testing is identical to Ammann & Offutt’s definition of GUI system testing, though only focusing on the functional characteristics (only the left arrow for GUI system testing in the figure).
2.1.3 A taxonomy of terms pertaining to testing of GUI-based software

In line with the majority of the literature on the topic, including the IEEE definition, we abide by the definition that states that system testing is about evaluating compliance of an entire software system with its specified functional and non-functional requirements. It follows that, although prominent definitions of system testing [5, 12, 13, 21] do not explicitly mention the GUI, in the case that the software system has a GUI, system testing encompasses the evaluation of the GUI against (GUI-specific) functional and non-functional requirements because system testing works on the entire product. This confirms that system testing includes GUI testing, which is very much like Ammann and Offutt’s definition. We, however, notice a difference with Ammann and Offutt’s definitions: Ammann and Offutt introduce the notion of GUI system testing, which is system testing through the UI, which is not identical to system testing; there might be system levels characteristics to check that would need to be checked by bypassing the UI. Those characteristics could be related to the functions of the logic such as correctness. For example, we may have a software function that calculates the value of a Boolean expression entered by the user. Testing that such a function returns the right value of a possible Boolean expression can be done by bypassing the GUI. It can be done through the GUI as well. Another example is a function which calculates the salary of an employee by accessing the database and evaluates the employee activities during the last month of her service. Usually, testers would prefer testing such function by bypassing the GUI because they want to isolate the error that could be generated in the logic of calculating the salary from other technical errors that can be generated by the GUI. Moreover, system testing may include testing characteristics related to the alities of the
logic such as performance. If a tester wants to evaluate the performance by testing throughput, response time or delay, this can be done by bypassing the GUI because the effect of the GUI on such characteristics is negligible. In other words, the main factor which contributes to throughput is the application logic calculations or database, not the GUI. Bypassing the UI for such verification will facilitate automation.

**Figure 3. Relationship among different testing types**

Figure 3 illustrates the main definitions we have encountered in our survey and that we have discussed in previous sections. It shows, as we did in the Introduction, that a GUI-based SUT can be decomposed into its UI and its application logic layers and their respective functional and non-functional (“alities”) characteristics (left- and right-hand sides of the layers, respectively). Arrows point to the layer that is directly exercised by tests and show whether tests go through the UI layer (dashed lines) or not. When an arrow points to the UI layer and does not continue through the UI layer to the application
logic layer, this means the UI is verified in isolation of the underlying application logic layer.

The figure illustrates the general definition of system testing we abide to (orange arrows): directly exercising the UI or the application logic layers (direct, plain arrows), possibly exercising the latter through the former (dashed arrows). Those tests focus on either functional or non-functional characteristics, which we refer to as functional system testing and non-functional system testing, respectively.

In red, the figure illustrates Ammann and Offutt's definitions. GUI testing is about the functional aspects of the GUI, focusing only on the UI layer, so the arrow goes to the functional part of the GUI and stops there. From their definitions, we do not have evidence that GUI testing also focuses on non-functional characteristics, especially since usability testing is a separate activity. Usability testing is about an "ality," so the arrow goes to the "alities" part of the GUI and stops there. GUI system testing is system testing through the UI, so arrows go to the UI (both functional and non-functional) and go through to the application logic. We already discussed that Memon’s definition of GUI testing is identical to Ammann and Offutt’s definition of GUI system testing, though it only focuses on the functional characteristics (only the left arrow for GUI system testing in the figure). As for input validation, it should be done on the GUI to evaluate how much the GUI tolerates invalid inputs. It has nothing to do with the underlying functionality, unless input validation takes place there instead of in the GUI.

In light of previous discussions, we introduce a new terminology to refer to different aspects of verification testing of GUI-based software.
We define *Functional System Testing* as checking conformance of the entire GUI-based software against its functional requirements, either by directly interacting with the application logic (arrow 3 in Figure 3), by isolating and focusing only on the UI (arrow 1), by focusing on the UI in combination with the application logic (arrows 1 and 2), or a combination of those.

We also define *Non-Functional System Testing* as checking conformance of the entire GUI-based software against its non-functional requirements, either by directly interacting with the application logic (arrow 6), by isolating and focusing only on the UI (arrow 4), by focusing on the UI in combination with the application logic (arrows 4 and 5), or a combination of those.

GUI verification can be either functional or non-functional, so we use the term *GUI functional system testing* for arrows 1 plus 2 and *GUI non-functional system testing* for arrows 4 plus 5. We also refer to system testing of the application logic code as *functional system logic testing* (arrow 3) and *non-functional system logic testing* (arrow 6).

We also call *GUI functional testing* the testing of the functional aspects of the UI that does not require the application logic (arrow 1 only), and *GUI non-functional testing* the testing of the non-functional aspects of the UI that does not require the application logic (arrow 4 only). These are specific, focused versions of the notions of functional system testing and non-functional system testing discussed earlier. Since we focus on functional aspects in this paper, non-functional system testing and GUI non-functional system testing are out of our scope. However, our taxonomy for testing types consist of: GUI functional system testing, GUI functional testing, functional system logic testing,
GUI non-functional system testing, GUI non-functional testing, and non-functional system logic testing.

One general issue with software testing is how to provide the right values to the software. Software controllability describes how easy it is to provide a program with the needed inputs, in terms of values, operations, and behaviours [6]. For example, it is easy to control a piece of software for which all inputs are values entered from a keyboard [6, 28, 29]. On the other hand, when the software gets its input values from sensors, it is difficult to control. Typically, a tester has less control with component/system testing than with unit testing. Controllability can also mean the ease to reach some predefined level of coverage, i.e., to exercise specific behaviour or pieces of code: it is more difficult to reach coverage of units with system testing than with unit testing. In general with a higher level of testing (e.g., system testing) it is harder to trigger specific elements of the code/functionality provided by lower levels of the code than with a lower level of testing (e.g., unit testing). When doing integration testing, it is harder to trigger specific statements of the code than with testing those units of the code directly. Similarly, when doing GUI functional system testing (arrows 1 plus 2 of Figure 3), it is harder to trigger code elements or behaviour of the application logic than when doing functional system logic testing while bypassing the UI (arrow 3 in Figure 3), and even more so than when doing unit testing. This is another reason that helps justify the distinctions we make between the different testing activities mentioned earlier and illustrated in Figure 3.

This figure also allows us to scope our research: we focus, as discussed in the Introduction, on conformance to functional requirements (arrows 1, 2 and 3). Moreover, our thesis is to look for technologies to conduct functional system testing on the UI in
isolation (arrow 1), or rather, as we will discuss later functional verification (rather than testing). This kind of verification will be combined with functional system testing of the application logic (arrow 3). The GUI verification and functional system logic testing (1+3) will be combined and conducted without conducting functional system testing of the UI and application logic through the UI (arrows 1 plus 2).

We illustrate these terms by characterizing a sample of existing works. For example, a work by Zebao et al. [30] about regression testing can be considered as GUI functional system testing; a work by Do et al. [31] to improve the JUnit framework can be considered as functional system logic testing; and a work by Jin et al. [32] about the use of JFCUnit testing can be considered as GUI functional testing. These three examples show different ways to test the functionality of a GUI-based software, which is the focus of this thesis.

2.1.4 Mapping existing work with our taxonomy

In this section, we use the new terms we introduced to map existing research. To do that, we look for primary studies in literature. We found a recent (published in 2013) systematic mapping study of GUI testing techniques [7]. As an initial study we sampled the list of references Banerjee et al. classified and selected the 29 journal papers they identified. Two of them, references [33] and [34] are surveys and cannot be mapped with our taxonomy, resulting in 27 studies to map. We selected journal papers since they admittedly represent the most developed research activities in a field. To classify a paper, we follow a set of steps:

1. We study the testing technique presented in the paper;
2. We identify the testing objectives of that technique and classify it as one or more of the testing types we introduced earlier (arrows 1, 2, 3, 4, 5, 6 in Figure 3);

3. We analyze, based on our own judgment, whether the technique potentially (though this is not the primary purpose) covers other elements of our taxonomy.

Table 1 shows the results of our classification; a short justification can be found in Table 2 (except for survey papers).

TABLE 1. CLASSIFICATION RESULTS

<table>
<thead>
<tr>
<th>Ref</th>
<th>Primary</th>
<th>Secondary</th>
<th>Ref</th>
<th>Primary</th>
<th>Secondary</th>
</tr>
</thead>
<tbody>
<tr>
<td>[35]</td>
<td>1+2</td>
<td>1+2</td>
<td>[36]</td>
<td>1+2</td>
<td>1+2</td>
</tr>
<tr>
<td>[37]</td>
<td>1+2, 3</td>
<td>1+2, 3</td>
<td>[38]</td>
<td>1+2</td>
<td>1+2, 4+5</td>
</tr>
<tr>
<td>[39]</td>
<td>1+2</td>
<td>1+2</td>
<td>[40]</td>
<td>1+2</td>
<td>1+2</td>
</tr>
<tr>
<td>[41]</td>
<td>1+2</td>
<td>1+2</td>
<td>[42]</td>
<td>1+2</td>
<td>1+2</td>
</tr>
<tr>
<td>[43]</td>
<td>1</td>
<td>1</td>
<td>[44]</td>
<td>1+2</td>
<td>1+2</td>
</tr>
<tr>
<td>[45]</td>
<td>1+2</td>
<td>1+2</td>
<td>[46]</td>
<td>1+2</td>
<td>1+2</td>
</tr>
<tr>
<td>[47]</td>
<td>1+2</td>
<td>1+2</td>
<td>[48]</td>
<td>1+2</td>
<td>1+2</td>
</tr>
<tr>
<td>[49]</td>
<td>1+2</td>
<td>1+2, 4+5</td>
<td>[50]</td>
<td>1+2</td>
<td>1+2</td>
</tr>
<tr>
<td>[33]</td>
<td>NA</td>
<td>NA</td>
<td>[34]</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>[51]</td>
<td>1+2</td>
<td>1+2, 4+5</td>
<td>[52]</td>
<td>1+2</td>
<td>1+2</td>
</tr>
<tr>
<td>[53]</td>
<td>1+2</td>
<td>1+2</td>
<td>[54]</td>
<td>1+2</td>
<td>1+2</td>
</tr>
<tr>
<td>[55]</td>
<td>1+2</td>
<td>1+2</td>
<td>[56]</td>
<td>1+2</td>
<td>1+2</td>
</tr>
<tr>
<td>[57]</td>
<td>1+2</td>
<td>1+2</td>
<td>[58]</td>
<td>1+2</td>
<td>1+2</td>
</tr>
<tr>
<td>[59]</td>
<td>1+2</td>
<td>1+2</td>
<td>[60]</td>
<td>1+2</td>
<td>1+2</td>
</tr>
<tr>
<td>[61]</td>
<td>1+2</td>
<td>1+2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The technique works based on a specification of the software functionality in terms of finite state machines in which states do input validation, a secondary objective is GUI non-functional system testing.

An approach combines event-level dynamic profiling for the whole software to collect user profiles information and GUI reverse engineering to extract the GUI widget information to generate/replay test cases. This is GUI functional system testing.

The authors use common testing tools such as SpecExplorer and NModel to do event-based testing of the whole software. This is GUI functional system testing.

The paper combines event-based model with capture/replay technique to test the software. It is GUI functional system testing.

An approach presents event-based model for testing. The model is used with GUITAR. This is GUI functional system testing.

A work that specifically focuses on the oracle when exercising both the UI and logic during tests. Hence, this is GUI functional system testing.

A special release from GUITAR for regression testing. This is GUI functional system testing. Since GUITAR incidentally does input validation, a secondary objective is GUI non-functional system testing.

The approach presents event-based test suite reduction technique based on the call-stack coverage criterion for the whole SUT. This is GUI functional system testing.

The authors use a finite state machine to represent legal state transitions in a GUI and underlying functionality. This is GUI functional system testing.

Testing of GUI code changes when the user makes an event on the GUI, later, it replays the test cases. This is GUI functional system testing.

This is an article paper in which the authors present a tool for testing the GUI interface and underlying functionality. This is GUI functional system testing.

For each referenced journal paper (1st/4th column), the table indicates the primary purpose of the work in terms of arrows in Figure 3 (2nd/5th columns) and potential

### Table 2. Justification of classification results

<table>
<thead>
<tr>
<th>Ref.</th>
<th>Justification</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>The approach watches GUI movements (e.g., mouse/button click) during executions to build a model that is then used for testing. This is GUI functional system testing.</td>
</tr>
<tr>
<td>37</td>
<td>The approach builds a finite state machine for the GUI layer and another one for the application logic layer, which are then used to create tests. This is GUI functional system testing and functional system logic testing.</td>
</tr>
<tr>
<td>39</td>
<td>The approach uses information from use cases to build a finite state machine and other reverse-engineered information to build control flow graphs and generate test cases. This is GUI functional system testing.</td>
</tr>
<tr>
<td>41</td>
<td>The approach builds a test model that represents all code without distinguishing between GUI code and non-GUI code. This is GUI functional system testing.</td>
</tr>
<tr>
<td>43</td>
<td>A work that specifically focuses on the oracle when exercising both the UI and logic during tests. Hence, this is GUI functional system testing.</td>
</tr>
<tr>
<td>45</td>
<td>A model is presented for event-based testing similarly to GUITAR, which does GUI functional system testing. This is GUI functional system testing.</td>
</tr>
<tr>
<td>47</td>
<td>A work that focuses on the oracle when exercising both the UI and logic during tests. This is GUI functional system testing</td>
</tr>
<tr>
<td>49</td>
<td>A special release from GUITAR for regression testing. This is GUI functional system testing. Since GUITAR incidentally does input validation, a secondary objective is GUI non-functional system testing.</td>
</tr>
<tr>
<td>51</td>
<td>A special release from GUITAR for regression testing. This GUI functional system testing and GUI non-functional system testing</td>
</tr>
<tr>
<td>53</td>
<td>The authors present an event-based test suite reduction technique based on the call-stack coverage criterion for the whole SUT. This is GUI functional system testing.</td>
</tr>
<tr>
<td>55</td>
<td>The authors use a finite state machine to represent legal state transitions in a GUI and underlying functionality. This is GUI functional system testing.</td>
</tr>
<tr>
<td>57</td>
<td>The authors present techniques to build an oracle for event-based testing after replaying a single event or after replaying an entire test case. This is GUI functional system testing.</td>
</tr>
<tr>
<td>59</td>
<td>The author combines event-level dynamic profiling for the whole software to collect user profiles information and GUI reverse engineering to extract the GUI widget information to generate/replay test cases. This is GUI functional system testing.</td>
</tr>
<tr>
<td>61</td>
<td>The authors use common testing tools such as SpecExplorer and NModel to do event-based testing of the whole software. This is GUI functional system testing.</td>
</tr>
<tr>
<td>36</td>
<td>The paper presents a performance evaluation of a capture/replay technique for GUI and underlying functionality. This is GUI functional system testing.</td>
</tr>
<tr>
<td>38</td>
<td>An event-based testing work using GUITAR. This is GUI functional system testing.</td>
</tr>
<tr>
<td>40</td>
<td>The paper presents coverage criteria for event-based testing. They evaluate their efficiency and effectiveness. This is GUI functional system testing.</td>
</tr>
<tr>
<td>42</td>
<td>The authors do random testing (by user clicks) and functional testing (black box testing for function points) of GUI-based SUT. Both are GUI functional system testing.</td>
</tr>
<tr>
<td>44</td>
<td>A technology similar to GUITAR combined with techniques from artificial intelligence. This is only GUI functional system testing.</td>
</tr>
<tr>
<td>46</td>
<td>The authors analyze execution paths of the software using log files and generate a model to assist in writing test cases to be used in testing of GUI-based software. This is GUI functional system testing.</td>
</tr>
<tr>
<td>48</td>
<td>The paper combines event-based model with capture/replay technique to test the software. It is GUI functional system testing.</td>
</tr>
<tr>
<td>50</td>
<td>An approach presents event-based model for testing. The model is used with GUITAR. This is GUI functional system testing.</td>
</tr>
<tr>
<td>52</td>
<td>The technique works based on a specification of the software functionality in terms of finite state machines in which states are identified based on the GUI events. This is GUI functional system testing.</td>
</tr>
<tr>
<td>54</td>
<td>Memon presents a regression testing approach. This contribution came before the release of GUITAR. It is GUI functional system testing.</td>
</tr>
<tr>
<td>56</td>
<td>Testing of GUI-based software using video recording. Ultimately, it is a capture/replay mechanism because it tracks the code changes when the user makes an event on the GUI, later, it replays the test cases. This is GUI functional system testing.</td>
</tr>
<tr>
<td>58</td>
<td>A work on an event-based testing of GUI-based application as a whole. This is GUI functional system testing.</td>
</tr>
<tr>
<td>60</td>
<td>This is an article paper in which the authors present a tool for testing the GUI interface and underlying functionality. This is GUI functional system testing.</td>
</tr>
</tbody>
</table>
purposes (3rd/6th columns). We do not use the terminology in the table for space reasons. Results show that the vast majority of works (25, 93%) do GUI functional system testing (arrows 1+2). One study does GUI functional system testing and functional system logic testing (arrows 1+2, and 3), and one study does GUI functional testing (arrow 1).

We did not find any work that specifically focuses on non-functional aspects (Primary objective). Only Memon and colleagues, with GUITAR, incidentally achieve more than their Primary objective, which is GUI functional system testing [38, 49, 51]. This is due to the fact that their tool, GUITAR, can be used to provide erroneous inputs to the GUI under test: there is some GUI non-functional system testing.

2.1.5 Conclusion

Recognizing there exist ambiguities around definitions of GUI testing, we present a taxonomy of terms that distinguishes testing of a GUI-based software along two dimensions: whether functional or non-functional aspects are specifically targeted; whether tests exercise the UI only, the UI and the application logic together, or only the application logic.

We evaluate a selected number of most developed related works and tools against this taxonomy and report that the vast majority of works look alike in light of the taxonomy: they conduct what we coined GUI functional system testing, which is system testing through the UI of functional aspects of the entire application, that is functional aspects of the UI as well as functional aspects of the application logic.

We first note that our definitions help distinguish functional aspects of the UI from functional aspects of the application logic. It appears from our mapping that existing
works and tools primarily focus on functional aspects of the application logic, through the UI, and not necessarily on functional aspects of the UI.

We also note that controllability issues are not discussed in these works and tools. Specifically, controllability issues may arise and prevent achieving all objectives in terms of functional testing of the application logic code through the UI, which calls for additional system level testing of the application logic which we coined functional system logic testing.

Our results also show that very few works and tools consider non-functional aspects of a GUI-based software, i.e., both non-functional characteristics of the UI as well as non-functional characteristics of the application logic. Very few of them distinguish the (testing) verification of the UI from the (testing) verification of the application logic, despite the fact that, according to our discussion and according to standard software design principles, the two might be different. For instance, it is conceivable to observe a GUI-based software that passes verification conditions (functional and non-functional) established for the application logic but fails to pass verification conditions (functional or non-functional) established for the UI.

Acknowledging our taxonomy may be debated, we believe it is nevertheless a good starting point to continue the discussion as to what we should call “GUI testing”. We argue the taxonomy is useful to have an overview of the field and pave the way to future work: e.g., do GUI functional system testing (arrows 1+2) and functional system logic testing (arrow 3) exercise functionalities differently? How is this related to controllability, if ever? What GUI functional testing, the testing of the functional aspects
of the UI without the application logic (arrow 1) look like? What about specifically focusing on non-functional aspects?

2.2 Entity Control Boundary (ECB) design principle

The overall architecture of the Software Under Test (SUT) we use later follows the Entity Control Boundary (ECB) design principle which divides classes over three main categories [10, 62-64]: Entity classes represent the information the software needs to manipulate and determine the state of the software (i.e., the temporary and permanent information); Control classes realize the use cases, implement the logic of the software, and determine how the state of the software changes (i.e., when and how the state changes); Boundary classes realize the interactions between the software and the actors (e.g., human, hardware, other software), transmit requests and data without changing the semantics of the information (Section 2.3), though they possibly change the type of the information (e.g., from an \texttt{int} variable to an \texttt{Integer} object) and determine how the software is presented to the outside world [65].

One benefit of the ECB principle is to concentrate any update to the interface in Boundary classes, update to the complex functionality in Control classes and updates in the information in Entity classes [65]. Complex functionality is a functionality that requires calls to a several classes. Hence, it is usually implemented in the Control classes. The separation of concerns between the semantics of the software functional core and the user interface provided to the user enhances portability, reusability and customization of multiple interfaces [65]. An Entity class is used to store persistent information in data structures and hence isolate changes to the information they represent [65]. Modelling
software using ECB facilitates separation of concerns, understanding, maintenance and evolution, among other things. For instance, Boundary classes are more likely to change when porting software to service a different category of users (e.g., a person with a disability). We found it easy to change the Boundary classes when updating the GUI to be able to use GUITAR (Chapter 4).

2.3 Boundary changes the syntax but not semantics

Syntax is a set of rules that describe whether the program is correct while semantics is about the meaning of the program. We use the words semantics and syntax in this thesis. However, we assume slightly different meanings than their original interpretations. In our context, we use the term "a change in semantics" to refer to a change of the value of a variable. We use the term "a change in syntax" to refer to a change in the formatting of the value of a variable. A change of formatting does not mean a change of value. If we change the format of a number equal to 2 to be 2.0, we do not change its value in our context. A change of value in our context is a change of 2 into 2.01 or into 3.

Bruegge and Dutoit describe that a Boundary class should collect the information from the actor and convert it into a form that can be dealt with in the Control and Entity classes (page 182) [10]. We conclude from this statement that a Boundary class is responsible for changing the syntax of information to be understandable by the Control. We understand the word “a form” to mean that no change to the semantics should be done by a Boundary class. This means that the value should not be changed by a GUI class.
External systems such as communication interfaces, sensors, actuators, printer interfaces and Application Programming Interfaces (APIs) are represented by Boundary classes [65]. A Boundary class is about the interactions between the software and its external environment. It receives/sends information from/to that external environment. A Boundary class focuses on the requirements at the system boundaries. This means that a Boundary class implements the ways through which the software can interact with the actors because those ways are described at the boundaries of the software [65].

Bruegge and Dutoit describe a Boundary object as a high-level representation of the user interface when the actor is human [10]. In other words, a Boundary class does not represent the visual properties of the GUI. For example, a menu item or scroll bar is not a Boundary class.

The ECB design principle also applies to the other direction which is from the Control to the Boundary. The Control passes the output data to GUI, the GUI is supposed to show that data to the user as an output of the software. The GUI can change the syntax of this output data but it should not change the semantics.

2.4 Literature review

This section is a continuation of our literature review that we reported in the beginning of Chapter 2. However, it is not the last section in our literature review. We continue to report published work in Chapter 3. In this section, we focus on testing of GUI-based software rather than on functional system logic testing. We used a systematic method to identify relevant papers. Our search targeted the area of GUI functional system testing. We started by using Google scholar to identify founding papers. This meant using
the Google scholar to search for papers using the following keywords: GUI testing, GUI functional system testing, automated GUI testing, and “automated” and “GUI” and “testing”. We also searched the computer science archive at Cornell University Library (https://arxiv.org/).

We then identified survey papers from those founding papers by browsing through the list of papers obtained by Google scholar. We used Google scholar to move back and forth to check which papers cite a specific paper and which ones are referenced by that paper. We ignored papers that are not dedicated to desktop applications such as web and mobile applications. Then we selected test case generation papers and summarized the contribution of those papers. This enabled us to find GUITAR, a highly cited GUI testing tool. We investigated the way that GUITAR works through studying most of the papers related to GUITAR. The rest of this section is organized as follows. We discuss recent related surveys (section 2.4.1) and test suite generation (section 2.4.2). Then, we discuss a set of GUI functional system testing and GUI functional testing tools (section 2.4.3). Later, we explain how GUITAR tests software (section 2.4.4).

2.4.1 Recent surveys on GUI functional system testing/GUI functional testing

Adamoli et al. [9] surveyed 50 papers related to automated GUI functional system testing/GUI functional testing during work on GUI performance testing. The authors evaluated capture and replay tools in terms of their ability to record and replay, and found that such tools are unable to capture realistic interactions with real-world software through an interactive session with the user. They studied tools by comparing the
performance of automatically replayed software sessions with software sessions driven by real users.

Garousi-Yusifoglu et al. refer to all activities that should be conducted during the entire lifecycle of code testing as Software Test-Code Engineering (STCE) [66]. They surveyed the literature about those STCEs. They reported the existence of numerous publications in this field linked to the increase in its popularity since 2007, which led them to conclude that there is a need to start conducting comprehensive secondary studies. Their systematic review investigated the studies in the area of STCE, published between 1999 and 2012 to unveil the state of the art in STCE. Through a well-defined article selection process, 60 studies from 72 studies originally retrieved by their initial search were included in their study. They found studies for identifying test smells and proposed guidelines for test code refactoring aiming at removing test smells in the context of GUI test scripts. They observed that the detection of test code smells is an active area of research. The authors referenced a study by Memon [67] about an approach for comparing the quality of automated test oracles for GUI-based software applications [67]. They also found a study for the maintenance of GUI test code written using the IBM Rational Functional Tester tool. Later Garousi et al. compared several GUI testing tools in testing of industrial case study [68]. Recently, Garousi and Yildirim utilized the observations made by Garousi et al. (in their systematic reviews and surveys) to develop an automated GUI functional testing approach [69]. The approach simply depends on systematically created test scripts. It chooses the best practices from the state-of-the-art to fit with the SUT. The authors also incorporate their experience in testing industrial software.
2.4.2 Test suite construction

Reis and Mota extract information from requests of change raised by users of the software for fixing the bugs that were found in that software [70]. Authors parse the change reports (CRs) to find which GUI widgets trigger the functionality that contain the bug. This would help them in GUI functional system testing. Kurabayashi et al. present a method based on static and dynamic analysis to generate test cases for GUI testing [71]. The authors generate test scripts from new and old software and then check whether some functions have been changed. They obtain diagrams that describe the transition of the screen in addition to their test cases as an output of their approach. Memon’s Ph.D. thesis [72] presents a framework for GUI functional system testing [3], a reverse engineering tool that generates, runs, and assesses GUI tests [1]. Descriptions of the main components of that framework with further optimizations and improvements of the process may be found for reverse engineering [73], coverage analysis of test cases [53], test oracle generation [67], and regression testing [74, 75]. Memon et al. survey and exclude work on testing non-functional aspects of GUI, keeping only work on testing functional aspects. This may be an indirect indication that Memon indeed considers both kinds of requirements during GUI testing, i.e., they follow the standard definition of system testing, including both kinds of requirements.

Another approach for GUI functional system testing is to represent the behavior of the GUI as a state model, possibly with a technology which shows a significant reduction in the number of required tests to avoid the state explosion problem [76], and to generate tests using state-based testing strategies [77]. Model-based testing (MBT)
approaches for GUI functional system testing present a test case selection process which constructs test cases based on the model [78, 79].

Other approaches exist for capturing manually entered sequences in a “capture/replay” fashion, whereby test case selection involves entering input sequences of events manually. Approaches based on computer vision exist for the development of test scripts using visual language [80].

We discuss several pieces of work to show possible alternatives to our use of GUITAR to generate our test cases. Several tools exist for generating test cases automatically through automated test planning [44] using artificial intelligence (AI) planning to generate test cases based on the state of a GUI before and after executing a user-defined operation [3]. Nguyen et al. presented an approach [81] that enables testers to convert business logic test cases into presentation logic test cases because model-based GUI functional system testing approaches do not model and test presentation logic and business logic separately. This can be done by a technique to separate business logic from the GUI behavior model. This is especially useful for GUI variants (only one application with multiple GUI variants, which can be either a Windows, Java, or web-based front-end). To achieve their aims in generating test cases for the GUI and logic, user introduces a mapping language. The language is able to map a sequence of user actions with a sequence of expected outputs to those actions. Authors finally build a model and use a capture/reply tool to generate test cases. Another approach [82] identifies patterns in GUls and generates generic testing solutions based on the patterns.

Beside Memon's work, reverse engineering approaches [78, 83] include Silva and colleagues’ approach [78], which automatically reverse engineers a behavioural model of
the GUI from the source code of Java Swing-based GUI application, and Pavia and colleagues’ approach [84], which reverse engineers a GUI into a specification model that can be used [85] to generate test cases [3]. Amalfitano et al. presented an approach [83] that is similar to model-based testing using a reverse engineering technique to automatically construct the GUI model. The approach [83] automatically generates test cases from the state machine whose results can be automatically checked against pre-defined constraints for mobile applications. Tools such as Crawljax [79] and Revangie [86] employ similar techniques for web applications. Other approaches [87] enable testers to evolve test scripts by detecting differences between the original and modified versions of a GUI and providing warning if a script needs correction.

Vos et al. presented the Testar tool for automation of GUI functional system testing [88]. Testar scans the SUT to generate a tree of widgets that represent the software. Then, Testar generates test cases from those widgets. In addition to GUI functional system testing, Testar tests how well the software copes with erroneous input and errors during execution. After generating a tree of widgets, Testar uses reinforcement learning to find which sequences of actions lead the software to crash, for example, due to a high memory consumption or a long execution time [89]. In addition to Testar, Elyasov and Vos extend a tool called Log-based pattern interface (LOPI) to present a new type of testing oracle. The oracle uses the information in log files that software generate to obtain data which help make a decision whether two sequences of events are equivalent [90]. Esparcia-Alcázar integrated genetic algorithms to improve the events selection process and traversal procedure used in Testar [91]. There are other GUI testing tools in the literature like AUGUSTO [92]. The authors indicate that common
functionalities are independent from the type of software application. The common functionalities include authentication operations, CRUD (Create, Read, Update, Delete) operations, and "search and booking" operations. In other words, these functionalities are widely presented in software applications and their behaviour remains always the same despite minor differences. AUGUSTO is a test case generation tool that utilizes a built-in knowledge of the semantics attached with common functionalities. Such information is repeated as patterns in CRUD operations. This information is used by AUGUSTO to automatically generate effective test cases with automated functional oracles. The authors report that AUGUSTO can detect faults that cannot be revealed with state-of-the-art techniques [92]. Ramler et al. presented an experimental study on GUI functional system testing of large industrial software [93]. They used an adapter to help automated test generation tools to achieve better test coverage. The adapter simply changes method calls into events. Then, a sequence of events will be selected. This sequence of events is considered to be a test case [93].

2.4.3 GUITAR

In order to do GUI functional system testing, we selected GUITAR [8], a tool that was used in the industry to for instance test VMware (as the download page mentioned). This is encouraging for us as other testing tools based on pattern recognition techniques suffer problems with industrial software such as long test script execution time, high cost of maintenance and analysis of false positives [94]. GUITAR is a well-known automated GUI functional system testing tool that provides a higher level of automation than simpler capture/replay tools. In capture/replay the values of input, window coordinates, etc., are hard-coded in test script. This makes editing these scripts a challenging task.
since these values may change as the SUT evolves, leading the test script to fail during test (re) execution [95]. Test scripts therefore need to be updated manually, reducing the level of automation one can obtain [96, 97]. We utilize automated testing in this thesis to avoid problems attached with capture/replay.

Marathon [98] would be another solution than GUITAR, though it provides fewer functionalities. For example, Marathon does not generate a GUI test suite automatically and the tester needs to capture/replay test cases one by one (See Appendix J. The use of Marathon).

To provide appropriate context for our discussion of GUITAR, we now consider how GUITAR, from an automation engineering perspective, compares to existing alternatives. GUITAR proved to be a superior alternative to other tools in a comparative study [3] that included a comparison between: (1) GUITAR [8], a research tool that handles automated GUI functional system testing from ripping the SUT to replaying test cases; (2) NModel [99], a model-based testing framework for C# programs; (3) Quick Test Pro [100], a multi-platform tool for test automation; and (4) Selenium, typically used to create browser-based regression tests [101]. We also observed—by experimentation or reading papers—that it is superior to (5) Marathon [98], a capture/reply tool that uses Python; (6) Sikuli [80], a reverse engineering tool that uses image recognition techniques; (7) TESTAR [102] a GUI functional system testing tool uses computationally-expensive machine learning techniques (e.g. reinforcement learning) in finding efficient test cases and (8) JAutomate [103], a tool for GUI functional system testing based on image recognition.
GUITAR generates a set of XML files when testing a GUI-based software. Based on our experience it is possible to use Gephi [104] in order to visualize the graph used to extract test cases. This is an advantage of using GUITAR, as visualization allows manual verification of models. Consequently, we found GUITAR superior to many other tools and decided to use it.

We use GUITAR for several reasons. First it is a free research tool. It is free to download and use for academic purposes. We got support from a student and a professor who worked in the development of GUITAR. The number of downloads of GUITAR is more than 12 thousand downloads. The research group that developed GUITAR worked on the tool for the last eighteen years. Professor Atif Memon is the head of the research group. He has more than 11 thousand citations, and the majority of his research is in the area of GUI testing. Lelli et al. presented an empirical assessment of GUITAR against defects related to incorrect scaling and converting of data [105]. They found that GUITAR is able to find bugs defined in their fault model with some limitations on that type of defects.

2.4.4 GUI functional system testing using GUITAR

The data flow diagram of Figure 4 illustrates how GUITAR should be used by showing the different components of GUITAR (Ripper, Convertor, Generator and Replayer [3]) in four vertical processes starting with Ripping and ending with Replaying.

The Ripper is capable of automatically extracting GUI structural information from a GUI SUT through interacting with GUI widgets. The default GUI events such as left-click on clickable widgets (e.g., push button, radio button, check box) and type-in-text on
editable widgets (e.g., text field, text area) are recognized and captured. In addition, the Ripper uses an algorithm that needs specific guidance to discover relationships among GUI events and widgets (e.g., which widget causes which event). This information is encoded in the tool’s adapters. Ripper includes custom adapters for ripping containers and then moves to contained widgets. We are using GUITAR’s default adapters for Java widgets since our case study is written in Java.
The Converter (i.e., graph convertor) converts the GUI Tree generated by the Ripper into an Event Flow Graph (EFG) as an XML file. An EFG is a directed graph representing the event interactions on a GUI. Each node in an EFG represents a GUI event (e.g., click-on-ComputeTruthTable) [3]. An edge from node v to node w represents
a “follows” relationship between v and w, indicating that event w can be performed immediately after event v. An EFG is analogous to a control-flow graph [3].

The Generator generates test cases. The resulting test suite contains one test case for each event sequence of length L in the input EFG. L = 1 results in a suite that covers every event in the EFG and L = 2 results in a suite that covers every pair of consecutive events occurring in the EFG, and so on for longer lengths. These test objectives are those typically obtained by the so-called n-switch test criterion for graphs [6, 26, 106]. If needed—that is, if the event sequence of length L does not start from the first node in the EFG (i.e., initial state)—a prefix is applied to the sequence of events to make it accessible from the application’s initial state and hence the actual length of test cases may exceed the value of L [3]. A search algorithm can be used to identify the prefix in the GUI tree and EFG generated by GUITAR. Depending on the search algorithm, the prefix is not necessarily the shortest path from the initial state to the beginning of the event sequence of length L. In addition to the prefix, GUITAR may add connecting events inside the sequence of events between test cases. Connecting event (sequence) c2 may be added to test cases e1 and e2, given a prefix c1 to e1, to create a longer test case c1, e1, c2, e2 [107]. The user of GUITAR does not have control over whether GUITAR does this or not. This insertion makes e1 accessible from the initial state of the GUI and also makes e2 accessible from e1. Here, the added event sequence c1 is called a prefix event. GUITAR's algorithm generates the necessary prefix on-the-fly during test case execution [108].

The Replayer executes test cases on a GUI-based SUT. In fact, the Replayer executes a test case in isolation, i.e. one by one, each time starting from the initial state of
the SUT, rather than in the exploratory fashion of the Ripper. The ripped tree shows all paths that are executed during ripping. These are reachable states. When generating the EFG some abstraction and inference must take place. As a result, traversing the EFG will result in the tree paths. Moreover, the traversing will result in new paths that have not been explicitly explored/discovered during ripping. These new paths include state sequences that may not be feasible. Thus, a state observed during replaying may be unique, having never occurred during the ripping process. This GUI state can be analyzed to determine if a test case passed. The analysis may include comparing the state with the state stored from a previous execution—for example, during regression testing [3].

GUITAR supports test oracles capable of verifying any detectable aspect of the GUI state both during and after test execution [3]. Monkey is a tool for testing Android application GUIs [3]. GUITAR’s approach to testing oracles is similar to the Monkey's approach, in that failures of a certain class can be detected by a class-specific Test Monitor [3], but in our own experience there was no need for specific Test Monitors as the information in log files was enough to specify the failure type (e.g., crashes, permissions errors, timeouts). GUITAR’s support for customization throughout the model-based testing process also allows the framework to support development of tools with capabilities such as GUITAR’s combination of model-based, multi-platform support. Customization throughout the testing process makes GUITAR an innovative tool in comparison to current approaches in the industry [3].

One disadvantage of the use of GUITAR evident in this comparison is a lack of tool-based support for any manual (scripted or captured) test development. In other words, test cases are not editable by a tester simply because there is no test script.
Chapter 3  Background and related work on static verification

In this chapter we discuss the background on potentials techniques for static analysis of software. We limit our study of static analysis to those we believe are applicable to our domain and can provide a solution to our problem. The background familiarizes the reader with the static analysis techniques that we use in Chapter 5. It also clarifies the reason of our use of specific types of static analysis technique in Chapter 5.

We used a systematic method but not a systematic literature review to identify relevant papers. The method started by identifying related papers using Google scholar in the area of program analysis and software verification. This meant using the Google scholar search engine to identify papers using the following keywords: program analysis, software verification, software verification without testing, input-output relation in software, verification of input-output in software, GUI verification, and GUI output-input verification.

We then identified papers from those general topics which discuss software verification without testing by browsing through the list of papers obtained by Google scholar and skimming through pages trying to understand how that could be done. By using Google scholar to move back and forth to check which papers cite a specific paper and which ones are referenced by that paper, we realized that we can categorize those papers into dynamic or static analysis. Then, we picked up some keywords that we found: slicing, symbolic execution, abstract interpretation and Java Modeling Language JML
verification. A major part of the papers we found belong to either one or the other of the categories. However, we found other techniques that do not belong to any of those categories.

We found further tools/techniques by contacting participants in the 27th International Symposium on Software Reliability Engineering, Ottawa, Canada, ISSRE 2016 and discussing with them our research problem. Then, we surveyed by searching online resources, i.e., Google Scholar, IEEE Xplore and Scopus, using the aforementioned categories. This step was necessary to find recent papers in the area of program analysis especially for GUI.

We used the Digital Bibliography and Library Project (DBLP) Computer Science Bibliography [11] to look for publications related to GUI verification when we needed to search for a work of a specific author. In parallel, we used Google scholar to obtain the most cited papers in the area. For example, we used DBLP to find all papers of author "Fausto Spoto" because we found that he had contribution in abstract interpretation. As for using Google scholar to find important work, the reason for this use is to avoid missing important, most cited works in the field. For example, we would investigate more time in understanding recent work which has 300+ citations than a work published before 2000 with less than 15 citations.

Recall that we define the input-output relationship of GUI as a relationship between the input variables of the GUI and the argument variables of methods in the GUI to the Control. These output variables work as arguments to functions in the logic of software (Chapter 1). We focus on some static analysis techniques as we found them to be the most promising techniques to study the input-output relationship. These are slicing
(section 3.1), abstract interpretation (section 3.2), symbolic execution (section 3.3), and the Extended Static Checker (section 3.4). We present selected related work from literature about their use in the area of GUI software verification and (section 3.5). Finally, section 3.6 presents the conclusion.

### 3.1 Static slicing

Slicing is a technique for extracting part of a software by tracing the program’s control and data flow related to some data. Slicing is typically used to handle software complexity by focusing on specific parts of the code [109]. Hence, slicing filters out a subset of the software that preserves some relevant behaviour of the original software [109]. The kept parts of the code that can influence the value of a variable at a given location in the source code. Slicing can be used in debugging, software integration and testing. Static software slicing happens at compile-time [109], whereas dynamic slicing takes place at run-time.

Assume that we use slicing to locate a bug that is responsible for erroneous output in the first output statement of the piece of code below (Figure 5). A slice with respect to variable $a$ being used in the one but the last output statement is shown in an underlined typeface. The slice contains all the statements that affect the value of $a$ at the location of interest; all statements involving variable $b$ are omitted. We have a slice that is related to the variable $a$. Hence, all lines that affect the value of $a$ are included in this slice.
3.2 Abstract interpretation

Abstract interpretation depends on using mathematical approximation concepts [110]. It can be defined as follows: “a unified lattice model for static analysis of programs by construction or approximation of fixpoints” [111]. It can also be defined as follows: “A program denotes computations in some universe of objects. Abstract interpretation of programs consists in using that denotation to describe computations in another universe of abstract objects, so that the results of abstract execution give some information on the actual computations” [111, 112].

Abstract interpretation was developed in the seventies and has since been used in many aspects of computer science such as static analysis and verification and Boolean Satisfiability Theory (SAT) or Satisfiability Modulo Theories (SMT) solvers [113]. For software S, it is impossible to write a software system S that computes all possible executions based on all possible inputs and to represent them mathematically [110]. Hence, this is an undecidable problem. Abstract interpretation can be used in the verification of the undecidable properties of the software. This is done by the use of the

```java
public static void main(String[] args) {
    int a = 3;
    int b = 4;
    if (a==3){
        a = a + 3;
        b = b + 7;
    }
    System.out.println("a = " + a);
    System.out.println("b = " + b);
}
```
mathematical approximation of the data structures and the possible ranges of values [113]. Hence, abstract interpretation is used in the formal description and verification of undecidable properties of software [113]. It is a technique for automatic static analysis that consists in replacing a specific part of code by a less detailed abstraction in order to calculate some properties of the program [114]. This technique enables detecting runtime errors such as division by 0 and overflow. It can also detect shared variables and dead code [114].

The central concept is the abstraction of properties of the system, starting from the strongest ones, as specified by the system semantics (called: collecting semantics). Abstract interpretation is a kind of static program analysis in which reachable states are abstracted by local numerical intervals which are ranges of possible values for the states [113].

3.2.1 A brief introduction of abstract interpretation [115]

The concrete semantics of software is the set of all its possible executions. A semantics includes the input, program state and output variables. It changes by taking new values as a function of time while running the program (i.e., the values change during the execution of the program). A software trajectory is a vector of values, each of which describes the status of the software. Since the status of the software changes over time during execution, the values of the vector change over time as well.

An invariant is a property which holds for all trajectories of the software, i.e., all software execution paths [110]. Abstract interpretation analyzes software trajectories [115] and identifies safe zones and forbidden zones. When a trajectory does not lead to an
error it is called a safe zone. A forbidden zone is a part of a trajectory that may lead to an error [115]. An example of an error that leads to a forbidden zone is an overflow in a condition of a while loop. Abstract interpretation builds global variables for the state of the software in those trajectories and then partitions those trajectories into stages that represent similar behaviours of the trajectories [115].

An invariant is a condition that holds for all trajectories [115]. There is a specific way to calculate the abstract invariant by a static analyzer. The text of the program is converted into an internal form that can be used by the analysis. That internal form is called an abstract syntax tree. Then, software instructions are interpreted by no longer considering the concrete value of variables, but by considering the abstract values [114]. While symbolic execution (see section 3.3) creates a symbolic value for each variable in the software and analyzes possible conditions on that value, abstract interpretation focuses only on collecting semantics. The collecting semantics, which are important properties of the software, are used to build a model. The model represents software transitions from one state to another one. The model is a less precise representation than individual trajectories of how software in the real-world transforms from one state to another.

In the absence of loops, this propagation of abstract values is used to calculate the abstract state in an iteration after each instruction until reaching the end of the software execution [114]. In the case of a loop, abstract interpretation uses the idea of widening. Widening means watching two consecutive iterations and trying to focus on stable conditions among those two iterations. For the specific case of a loop, an initialization condition states that the invariant should hold before starting the loop. The iteration
condition states that if the loop invariant holds after any number of iterations and one more iteration of the loop is performed, then the invariant must remain true after this iteration. This leads to a so-called fixpoint [115]. It is the abstract invariant corresponding to this fixpoint that is propagated in the rest of the program [114].

3.2.2 Details of abstract interpretation

Abstract interpretation depends on building a graph. It starts by analyzing the Java bytecode to produce a representative graph for that bytecode [116]. A node in the graph represents a block of the code. An edge represents the control flow, including flow due to exceptions or recursion [116]. Abstract information spots relations among different parts of the code by building a model as a graph [116]. After building a graph that represents code information, an algorithm for finding strongly connected components can be used to determine the fixpoint [116]. Hence, a set of nodes can be replaced by one super-node, which is the strongly connected component. In this way, we have a new graph in which nodes are strongly connected components. The edges among the strongly connected components represent higher levels of abstraction than the original graph.

Spoto uses the concept of watchpoint semantics to name the collecting semantics term presented by Cousot [117]. He uses a set of labels to classify semantics so that, finally, he gets a hierarchy of semantics. This hierarchy is a classification of semantics into abstract, static, watchpoints and trace semantics [117]. Abstract semantics is a superset of the concrete program semantics [110]. An abstract value represents a superset of values that can be taken on by a variable of the software as shown in Figure 6. Thus, it is not the real state of the software that the analyzer is changing but its abstract state.
Abstract interpretation calculates an abstract semantics of the software by so-called abstract invariants.

Figure 6 illustrates the abstraction mechanism. It simply shows two domains in oval shapes, one of them containing concrete values (1, -1, etc.), and the other abstract values (positive, negative, etc.). Every element in the concrete domain can be represented by an element in the abstract domain. There will be a loss of precision, but within limits that guarantee the representation is acceptable. In order to do that, the abstract interpretation uses two functions: One function moves from the concrete semantics to the abstract one; the other one moves from the abstract semantics to the concrete one. The abstract semantics should cover all possible cases in the concrete domain. In order to achieve that, the abstract semantics must be selected as a superset of the concrete semantics \[111\]. The abstract interpretation analysis begins with the abstract values of the input variables of the program and propagates them instruction-by-instruction until it finishes. The abstract semantic is said to be safe when it does not intersect the forbidden zone \[110\]. If the abstract semantic is safe then so is the concrete semantic. If the abstraction is safe, then each abstract invariant represents a superset of the concrete states after each instruction. In the presence of a loop, the repeated interpretation of loop instructions leads to the abstract invariant getting stabilized, a fixpoint having been reached \[114\].
3.3 Symbolic execution

Symbolic execution is a technique for systematic exploration of software behaviours using symbolic variables that was first introduced by King [119] and used by Clarke [120]. The execution of a software needs an actual input data. Symbolic execution provides a way for representing software execution on that input data. This is achieved by using a symbolic value of that actual data and symbolic expressions to express the software variables [121].

By using symbolic values instead of actual data for input values, symbolic execution represents the values of software variables as symbolic expressions. Hence, the output values computed by a software are expressed as a function of the input symbolic
values [122]. Static symbolic execution does not need to execute the code, while dynamic symbolic execution needs to execute the code. Static symbolic execution is less accurate than dynamic symbolic execution [123]. On the other hand, dynamic symbolic execution is often time consuming. Static symbolic execution has been used for many software engineering activities, including privacy preserving error reporting [124], handling security issues [125], and load testing [126].

The state of a symbolically executed program includes [122]: (1) The symbolic values of software variables; (2) A path condition that is a Boolean formula over a set of symbolic inputs that accumulates constraints which the inputs must satisfy in order for an execution to follow a particular path attached with that execution; and (3) A program counter that is a control pointer that defines the next statement to be executed. A symbolic execution tree characterizes the execution paths during the symbolic execution of software. The nodes represent program states and the edges represent transitions between states [122].

3.3.1 A running illustrative example

Figure 7 shows a code snippet that swaps (exchanges) the values of integer variables x and y, when x is greater than y. It is an example commonly used to illustrate the use of symbolic execution for software testing [15, 20-23]. We show next the symbolic tree structure of this code by describing steps to build that tree. The symbolic tree will have a set of levels. For each of those levels there will be at least one node that represents operations for a line of code. The operations are represented symbolically. For example, each variable name is assigned a symbolic value instead of a real (in this case
integer) value. At the leaf nodes, we will have a set of conditions that represent an execution path in the software regardless of whether it is satisfiable or not (there are no values for the input variable which may lead to this execution path). Figure 7 shows the matching symbolic execution tree as well.

```
1 int x, y;
2 if (x > y) {
3   x = x + y;
4   y = x - y;
5   x = x - y;
6   if (x - y > 0)
7     assert(false);
8 } 
9 }
```

**Figure 7. Code that swaps two integers and symbolic execution tree [15, 20-23]**

In the symbolic tree, the path condition is assigned the value true at the beginning of the symbolic execution. This is due to the fact that the first instruction in the software will be executed for any value of the input. The two variables x and y will be assigned symbolic values X and Y, respectively. At each conditional software statement (if, while, etc.), the path condition will be updated with new conditions which represent the assumptions about the inputs. In other words, the path condition contains the condition of the conditional statement in order to choose between alternative paths. In this specific example, the first conditional statement is the if-statement at line 2. At line 2, after the
execution of this statement, both the then and else alternatives are possible. Hence, the path conditions are updated accordingly: (a) if the input value X is less than Y, then the control moves to the right branch of the symbolic tree. For this branch, the value of the path condition will be: X is less than or equal to Y; (b) if X is greater than Y, then the control moves to the left branch of the tree in which the value of the path condition is: X>Y. Thus, an input of software which causes it to take that path is obtained by solving the condition in the path condition.

Then, the regular execution of the statement at line 3 in the code includes adding the value of variable y to the value of variable x and assigning the output to variable x. Hence, in the matching part of the symbolic tree, the symbolic value Y will be added to the symbolic value X and the output will be assigned to variable x. However, the symbolic value of variable y stays the same.

In the next step, a simulation for the execution of the statement at line 4 is done in which a new value will be assigned to variable y, which is the difference between the last value of x and the value of y. In the symbolic tree, since the last value of variable x is the sum of two symbolic values (X and Y), the output is: X + Y - Y = Y. Consequently, variable x is assigned the symbolic variable Y.

In the next step, the statement at line 5 in the piece of code is simulated symbolically. In this statement, the value of variable y is subtracted from the value of variable x and the output is assigned to the variable x. In the symbolic tree, since the symbolic value of variable y is the symbolic value X, the symbolic value of variable x is the result of this subtraction: X - (the symbolic value of variable y) = X - (X + Y) = Y. Hence, the symbolic value Y is assigned to the symbolic variable x.
Then, the program counter moves to the statement at line 6, which is again a conditional statement. If the output of subtraction of y from x is positive, the statement inside the conditional block is executed. Consequently, in the matching symbolic tree, the path condition is whether the symbolic value of variable x is greater than the symbolic value of variable y. Hence there are two options: (a) the right branch of the tree in which the symbolic value of variable y is the symbolic value X and the symbolic value of variable x is the symbolic value Y. At this stage we would have reached the objective of the software, which is swapping the two numbers. In this case the path condition contains the two conditions X > Y and Y - X >= 0; (b) If the program counter moves to the left branch of the tree, then there are two conditions in the path condition: the first is X > Y and the second is Y - X > 0. Since it is impossible to find two numerical values which satisfy these two conditions at the same time, then this path is unsatisfiable (false !).

3.3.2 Remarks

Since symbolic execution produces Boolean constraints on symbolic variables, symbolic execution requires the use of constraint solvers [127-129]. Symbolic execution is therefore computationally more expensive than other types of software analysis (such as control flow and data flow analysis) because it makes sure that all constraints are satisfiable at control statements [130]. The limited computational capability of older-generation computers made it impossible to symbolically execute large software. However, with advances in computer machines, the barrier to applying symbolic execution to a large, real-world software is significantly lower than it was a decade ago [131]. However, one of the limitations of symbolic execution is that symbolically executing all feasible program paths does not scale to large software, which leads to path
explosion. The number of feasible paths in a program grows exponentially with an increase in software size [132].

Based on our reading of materials [133, 134] related to symbolic execution we found the following limitation: static symbolic execution is less computationally expensive than dynamic symbolic execution, although the latter is more accurate. In fact, this accuracy comes with the necessity to execute the code.

In addition, symbolic execution using Java Path Finder (JPF) may not work well with all packages of Java as it was not designed to work with all packages. For example, it has problems with GUI packages which rely extensively on complex and inherently concurrent framework libraries such as AWT/Swing [135]. Hence, a special version of JPF called JPF-AWT has been designed to handle GUI code [135]. However, the verification of GUI-based software using model checking (check whether a model meets a given specification) still requires a user model. The user model is one that provides a way to understand how the user of the software interacts with the GUI. The actions of the user when interacting with the GUI could be: clicking a button, editing a text field, selecting an item from the dropdown list, etc. In other words, this model describes the activity related to the user rather than the software (i.e. it represents the user actions on the GUI when interacting with that GUI). Consequently, the user behaviour is critical in the verification process [135]. We did not notice special implementation of Julia for GUI-based application. We do not have any reason to consider that abstract interpretation will not work well with GUI code. Moreover, we were able to apply abstract interpretation in our case study by providing only the software as an input.
3.4 Extended Static Checker (ESC)

JML is a language to describe the behaviour of a software module written in Java [136]. ESC is a framework that contains methods for using JML constraints to check the correctness of a software statically [137].

ESC checks errors in software behaviour due to discrepancy among data types for software constants, methods and variables. For example, storing int in a double variable. Errors come when the software try to do operations on values which are not of suitable data types [137].

Furthermore, ESC extends the previous type checking capability to report errors based on user defined annotations in the form of pre and post-conditions and invariants. Hence, ESC extends type checking [137]. Type checking is part of the multi-phases process that ESC use to verify JML conditions

The ESC frontend type-checks the software source code and annotations to produce a structure of data types. The structure describes software statements (while, return, assign, compare), operators (> , >=, -), and variables regardless of the syntax of Java. This structure is an abstract syntax one because it not specific to syntax of Java and it does not have details about the source code like parentheses [137]. A translator receives this structure and produces a set of (assert expr) commands where expr is Boolean expression. A verification conditions generator receives the commands and studies both the specification (the JML statements of software) and implementation (the source code of software)[137]. Hence, the generator produces functions that map the pre-condition to a post-condition for each software routine (software function or method). The function is
the weakest pre-condition on the initial state attached to a function. That state is transformed to a final state after executing the function. The final state should satisfy the post-condition. This type of analysis is done by the generator to produce a set of statements that may be evaluated to true or false depending on values of its variables (Boolean functions). A theorem prover takes these Boolean functions as an input, other Boolean functions related to the type from the frontend (about the class in which the routine is defined), and general information about Java (e.g., any type in Java comes from the type Java.lang.Object as an input) [137]. The prover tries to prove the Boolean functions. However, the prover (Simplify) may not be able to prove all Boolean functions. Hence, results are passed to post-processor which gives warning to user [137].

3.5 Related work

In this section, we present examples from literature where technologies introduced earlier have been used, focusing on their use for GUI software. Section 3.5.1 presents related work on slicing. Section 3.5.2 justifies why some examples from literature on dynamic slicing will not work in our case study. Section 3.5.3 presents examples from literature about abstract interpretation. Section 3.5.4 and section 3.5.5 present related work on symbolic execution. Section 3.5.6 presents related work using other kinds of static analysis. Section 3.5.7 presents a background about Julia. Finally, section 3.5.8 presents other tools that we investigate.

3.5.1 Slicing-related work

Slicing has been used to extract a model containing all possible GUI behaviours [138] in order to achieve a better understanding of the GUI code and, hence, test it. The
GUI code is difficult to understand as part of it could be generated by tools. The understanding of the code would assist in evaluating the software usability and quality, for example, by separating GUI code from none-GUI code. It could therefore be used in maintaining the code by guaranteeing that maintenance did not generate unintended errors in the new code [138]. Authors use a reverse engineering approach for GUI layer extraction and GUI behavioural abstraction. Then, authors combine model-based GUI functional system testing with their reverse engineering approach. The purpose of GUI behavioural abstraction is to understand the code related to the user interface [138]. GUI functional system testing is done by construction and utilization of a GUI state machine for the purpose of testing. Authors build an abstract syntax tree by parsing the code [138]. They specify which parts of that tree are related to the GUI layer. Slicing would be applied to the abstract syntax tree to extract a UI abstract syntax tree related to the GUI code for the whole software code. The authors use a traversal algorithm that scans the tree based on the dependency graph and detects all GUI nodes.

A Finite State Machine (FSM) is built based on the extracted code. Then, the code will be tested using that FSM [138]. The whole approach is for GUI reasoning and testing.

As for slicing tools, we can use Indus [139] or Wala [140] static analyzers. Other tools like JSlice [141] are not interesting to us because they do dynamic slicing.

3.5.2 Dynamic slicing-related work

Dynamic slicing is not better than static slicing (the focus of this thesis) in terms of performance. For example, a combination of static and dynamic slicing is used to
determine sources of possible values for variables [142]. This option of finding sources of values enables a user or a tester to have a better understanding of the code. Such an option can be considered as a reasoning tool. For example, using this reasoning tool that authors provided, developers can right click on one line of code and ask the tool why a variable x took a value of 5. The tool would present explanations for why a specific line of code (highlighted by the developer) was reached or not. Hence, the tool provides a new way for software debugging. This is an indication that each slicing type has advantages and disadvantages, otherwise researchers would always use dynamic slicing, which is more accurate than static slicing.

Arlt, Podelski and Wehrle used static slicing rather than dynamic slicing to reduce a GUI test suite. They justified their choice by the efficiency and precision trade-off between static and dynamic slicing. In other words, while static slicing does not need to execute the software, it loses precision when compared to dynamic slicing [143].

3.5.3 Abstract interpretation-related work

We did not notice the use of abstract interpretation for the verification of the GUI of software in the literature. However, we found a framework called Bandera, which enables the automatic extraction of finite-state models from the source code of software [144]. Bandera uses abstract interpretation in order to extract the finite-state model.

In addition, we noticed the use of abstract interpretation by Airbus France in hard real-time avionics software, such as flight control software which is always expected to react in time [145].
3.5.4 Review of published work on the use of symbolic execution for GUI verification

Zhang, Chen, and Wang used symbolic execution for generating test cases and their inputs [146] by finding feasible paths and then generating test cases. The test case tests both the GUI and underlying functionality. Another tool is presented by Cheng et al. for concolic execution using a cloud service [147]. Hentschel et al. presented a tool for Debugging [148]. Their tool uses symbolic execution to collect information about the code. Specifically, it collects information about the control flow in the code. The tool is used as an Eclipse plug-in to generate a tree describing the flow of control in Java blocks such as Java try-catch, loops and exceptional path. The plug-in visualizes this information to make it easy for the software tester to verify the code, whether it is a GUI code or underlying functionally code. Ganov et al. [1, 122, 149, 150] present the use of symbolic execution for GUI functional system testing. However, their approach is similar to the approach that GUI functional system testing tools like GUITAR use, but with a different implementation. They use symbolic execution to build a graph that is similar to the event flow graph constructed by GUITAR. Any sequence of events in that graph is a test case. The only advantage over GUITAR is that since they use symbolic execution, they know what the satisfiable sequences (i.e., test cases) are. On the contrary, GUITAR does not know whether its test inputs actually trigger the expected control flow paths. Moreover, in our thesis, we need to verify the UI and not test it. We do not need a technique that performs the same tasks as GUITAR because we are not interested in executing sequences of events. GUITAR cannot always generate valid user input data in GUI functional system testing. However, Ganov's approach to finding feasible paths and then
generating test cases does not solve the problem of input data either. In other words, Ganov's approach reduces computation by focusing on successful execution paths but does not generate the input itself.

Symbolic execution has been used to generate test inputs to improve code coverage and expose software bugs [151-153] as well as privacy preserving error reporting [124], fault localization [154], regression testing [155], robustness testing [156], and inputs anonymization for testing of database applications [157]. Symbolic execution is also used for program verification of GUI software [1, 149] by generating test input and test scripts to execute the GUI code (white box testing). The combination of symbolic execution and safety property assessment can be used for verification of safety critical applications [158]. Khurshid, Pasareanu and Visser claim that their testing method is an efficient way to verify software with complex data structures [159]. They solve the problem of state space explosion due to the many inputs provided to the software. They used symbolic execution to avoid the increase in the number of states attached to software with complex data structures. Castro et al. [124] use symbolic execution to generate test inputs that can reproduce software failure.

Santelices et al. [155] generate test inputs for regression testing. Majumdar and Saha [156] use symbolic execution to generate test inputs whose slight changes cause a significant difference in a program's output.

We tried with symbolic execution in order to find a mathematical formula as a function which should represent the output in terms of input. However, we found that this technique does not find a function. Instead, it can formulate only a set of conditions on the input.
3.5.5 Dynamic symbolic execution

Dynamic symbolic execution [131] is a more popular version of symbolic execution. It includes running software while collecting path constraint on inputs from predicates encountered in branch instructions [133]. It also includes deriving new inputs from a path constraint by a satisfiability solver in order to steer next executions toward new program paths [133].

Dynamic Symbolic Execution has been used in concolic testing, which is a technique that combines symbolic execution along with concrete testing [160, 161]. There are tools to implement a (dynamic) symbolic-execution system [78, 162-165].

Qi et al. [166] used dynamic symbolic execution to generate test inputs by comparing execution traces of the failing input and new generated inputs to provide clues to the root-cause of the failure. These test inputs are similar to a given program input that causes the software to fail. However, the inputs do not cause failure.

Khurshid et al. present an approach to generate a sequence of method call [159]. This sequence of calls is used to generate test cases. Khurshid et al. use dynamic symbolic execution and model checking testing. The heart of their framework is an algorithm for symbolically executing a method that takes complex data structures as inputs. The approach is a generalization of traditional symbolic execution. In addition, Khurshid et al. define new instrumentation methods for software and test the input generation mechanism. Then, they enable symbolic execution of the software to be performed using a standard model checker.
One way to do dynamic symbolic execution is to instrument a software application and execute the instrumented code such that the execution of the instrumented software essentially performs symbolic execution of the original program [130]. In this technique, the declared type of each program variable is replaced with another type in the instrumented program so that variables in the instrumented software can store symbolic values. This technique [153] is used in existing symbolic execution systems such as Java Path Finder [167] and its adaptation Symbolic Pathfinder [168]; Juzi [169], the system used in context-sensitive analysis in which execution context is analyzed to check memory allocation and reduce computation [170]; and JFuzz [162].

Another procedure is to maintain an explicit map from variables to symbolic values during execution. Different tools implementing this procedure are available: JCUTE [171], Klee [151] and Pex [172].

Some types of testing utilize both concrete execution and symbolic execution of code simultaneously [173]. One of these examples is concolic testing. LIME Concolic Tester (LCT) is a concolic testing tool [173]. The LCT approach includes instrumenting the bytecode of the SUT to enable symbolic execution. At the same time, it collects conditions on values of input to be used in guiding the testing to execution paths that had previously been missed [173].

3.5.6 Related work on other static analysis techniques

Separation Logic is a static analysis technique that studies semantics of software fragments. Tonder and Le Goues used Separation Logic to identify patches of code and then verify their correctness to detect defects such as memory leaks and null references
The application of static analysis is not only the use of sophisticated tools such as JPF to analyse the code. Other types of code analysis, even without the use of sophisticated tools, can be considered to be static analysis. Arlt and colleagues [175] applied static analysis of events relationships by checking the bytecode of a GUI application and its dependent libraries for GUI functional system testing. This allowed them to infer dependency relations between events. The relation is used to build a graph, which they call an Event Dependency Graph (EDG), to select relevant event sequences among the event sequences generated from a black-box model.

Arlt et al. [175] incorporate static analysis through the process of GUI system testing using black-box test cases. Zhang et al. [176] analyze the dependent libraries of a GUI application for test case generation. Yuan and Memon [177] obtain GUI run-time feedback from the execution of a “seed test suite” and then use static analysis (with a data-flow static analyzer) to analyze this seed test suite and iteratively generate new test cases. The authors utilize the run-time state to explore a larger input space and improve fault-detection effectiveness. They automated this feedback-based technique into an end-to-end GUI functional system testing process on fielded applications.

Techniques which are similar to static analysis have been used [178] to discover the most relevant functionalities of a piece of software and to generate test cases that thoroughly sample these functionalities. An example of those techniques is reinforcement learning for testing algorithms implemented in AUTOBLACKTEST (ABT) [178]. ABT is a verification tool that uses algorithms from the area of learning field.

Search-based techniques learn by themselves how to interact with the software and simulate its functionalities. Other approaches use search-based techniques to execute
actions and observe states of a certain behaviour in the source code [165] to generate test cases at the GUI level.

3.5.7 Julia

We provide here high-level description on a tool called Julia why we leave details on how to use for Chapter 5. We investigated an abstract interpretation tool called Julia to be used in our experiments [117]. In this section, we give a background on how Julia works. Julia observes conditional statements in the code, collects information about methods, and determines when and how they are being called [116]. The abstract interpretation using Julia is performed through a fixpoint calculation, focusing on program points called watchpoints [116]. This is done through a software component that receives software byte code and dumps results into a report file that describes the output [116]. Spoto (author of Julia) and Jensen observed that the information provided by many static analyses is significant or useful only at a limited set of software points, which they call the watchpoints [117]. For example, information about a variable which can contain zero at run-time is useful only before a division [116]. Class analysis includes identifying which call leads to a specific target method [116]. Hence a watchpoint must be put before the call of the software [116]. Julia depends on this idea to focus on specific software points.

Julia uses a hierarchy of semantics, which is a concept used to classify semantics into two types: Trace and watchpoint [117]. Trace is a sequence of states for a piece of code, while watchpoint is a program point that plays an essential role in program behaviour. A piece of code can be represented by a set of traces to create a sequence of
states for that code. There is a trace for every possible input state for the code. Abstract interpretation tries and tests several approximation mechanisms [111]. However, in terms of practice, Julia allows the user to add watch point at any part of the code she/he wants.

As for our scope, we use the Julia static analyzer as follows. Julia provides a set of features that the user can use to analyze the software statically. We are interested in a feature that enables us to find numerical invariants. This feature has been implemented in Julia version 2.3.4. In this deployment of Julia, there is a new option named termination checker which is available in the Julia wizard. This option is usually used in operations related to termination of Java programs. This has been an issue for some researchers as some Java programs do not terminate completely until shutting down the Java virtual machine. There is a branch of research on this topic that investigates how to make sure that a program is terminated [179]. However, in our scope of verification of software, we are interested in an option called dumpNumericalAssertions. By turning it on through clicking the check box in the Julia wizard, Julia generates an output text file with numerical invariants at user-defined program points. We discuss the details of usage in section 5.9.

Julia observes conditional statements in the code. It also collects information about the software methods such as when and how they are being called. Julia also tries to find an activation frame, which is a description on when and how a software method is called. Then, Julia uses a lookup procedure to find the target software method (software function) of the call. Then, Julia creates an activation frame for the called method [116]. The activation frame for a method is a setting that simulates the circumstances when a
call happens. Finally, Julia moves the output of the called method into the stack of the caller.

The abstract interpretation using Julia is performed through software components that receive software bytecode and dump results into a report file that describes the output. One component is a code preprocessor. A fixpoint engine is another component that uses an external module to abstract bytecode. It also has its own fixpoint strategies. A third component is a library that works as a low-level interface to .class files [116].

3.5.8 Other studied tools

When looking for a tool to be used for static analysis, we found the Low Level Virtual Machine (LLVM) [180] as potential candidate. It is a compiler framework designed to support software analysis. LLVM can work as a static analyzer to optimize the code. It does not help to study the input-output relation of the GUI.

Hence, we did not move further to investigate the abilities of this tool in terms of abstract interpretation.

LLVM was initially designed to support research and development on compilers. It can therefore be used for the development of new compilers, but at the same time it can be used in software quality assurance through the software development life cycle. LLVM extracts information at compile-time, link-time, run-time, and in idle time between runs [181].

Soot is a static analyzer for Java code [182]. However, again, we did not find any evidence that Soot can give us information about input-output relation. Soot processing of exception-handling code is less accurate than when handling other code (non-
exception handling) [116]. We studied other techniques such as taint checking [183] but it did not provide the functionality that we want.

Polyspace [114] is another example that we considered. However, our case study is written in Java and we therefore need a static analyzer for Java. Unfortunately, Polyspace Code Prover was designed to prove the absence of run-time errors in C and C++ source code [184].

Atlas is another static analysis tool that we investigated [185]. We used Atlas on a dummy case study. The data flow diagram can be given to a software tester to assist him in the verification of the code. The graph of the Data Flow Diagram generated by Atlas is easy to inspect by a software tester. It also provides a comprehensive idea of all the possible calls to this argument in the software. However, Atlas does not generate any mathematical function that represents a software method. Moreover, the graph generated by Atlas (even in the case of a small dummy case study) is hard to navigate on a computer screen.

3.6 Conclusion

We investigated a set of tools such as symbolic execution because it works on symbolic values of the input of software functions. Specifically, the path condition which is used in symbolic execution accumulates some conditions on input variables. Hence, we investigated if there is a possibility that symbolic execution will help us in studying the input/output relationship. However, as discussed later, symbolic execution does not give us the exact function that we need to describe the relation between the input and output.
Abstract interpretation has a higher potential to be used as a static analysis technique, in our case to formulate the input-output relation. If we need to reduce the computations for abstract interpretation, we can use slicing as a pre-processing step to be followed by abstract interpretation. Though abstract interpretation was successful in the case of simple functions, it was not in the case of more complicated ones as discussed in Chapter 5. Then, we moved to the investigation of static checking using Extended Static Checker (ESC). ESC proved to be efficient to detect the violation of conditions due to faults as we show experimentally in Chapter 5. However, ESC has some limitations with strings processing.
Chapter 4  GUI functional system testing versus functional system logic testing on the application logic
– An experiment

In this chapter, we discuss an experiment we conducted whereby we compared GUI system functional testing and functional system logic testing as defined in section 2.1.3. This is an attempt to answer Research Question 2: Is GUI system testing of the software logic equivalent to functional system testing applied to the logic by bypassing the GUI in terms of structural coverage?

We measure the coverage of the logic of the software when doing two types of testing. We consider that the testing objective of the two types of testing is to test the functionality of the logic. GUITAR has been designed to test the functionality of the whole software. We use it for the same testing objective. However, we measure the coverage of underlying functionality. The testing objective of the functional system logic testing is testing the underlying functionality. Hence, we argue that our comparison is a valid one.

We first introduce the case study software we used in the experiment (section 4.1). Next (section 4.2) we discuss the measure we used in the experiment to answer the research question. Sections 4.3 and 4.4 discuss executing the functional system logic tests directly on the application logic code and GUI functional system tests created and executed through the GUI by GUITAR, respectively.
Figure 8 shows the relationship between GUI functional system testing and functional system logic testing in the context of our next case study: GUI system tests are executed on the GUI directly. The functional system logic tests can be executed directly on the application logic code. In the case of the ECB design principle, this means that tests are executed on Control classes.

Section 4.5 discusses results. We analyze results and costs in section 4.6. Threats to validity are discussed in section 4.7. We conclude the chapter in section 4.8.

4.1 Software Under Test

Our software under test performs a number of computations on Boolean expressions provided by the user through a GUI. The SUT provides three main functionalities: computing and displaying the truth table of a Boolean expression, computing and displaying the Disjunctive Normal Form (DNF) of a Boolean expression, and deriving tests according to the variable negation strategy for a Boolean expression [186]. Table 3 shows the class content of each package of the SUT.
### Table 3. Classes and Packages of the SUT

<table>
<thead>
<tr>
<th>Package</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>BooleanVariables</td>
<td>BooleanVariable, SetOfBooleanVariables</td>
</tr>
<tr>
<td>Controls</td>
<td>CheckTestCase, Cube, DNFControl, TruthTableControl, VariableNegationControl</td>
</tr>
<tr>
<td>Gui</td>
<td>DNFComponent, MainExeFrame, TruthTable_TableModel, TruthTableComponent, VariableNegation_TableModel, VariableNegationComponent, VariableNegationResultsTable</td>
</tr>
<tr>
<td>Parser1</td>
<td>LogicalExpressionParser, LogicalExpressionParserConstants, LogicalExpressionParserTokenManager, ParseException, SimpleCharStream, Token, TokenMgrError</td>
</tr>
<tr>
<td>Parser2</td>
<td>BinaryExpressionSolver, BinaryExpressionSolverConstants, BinaryExpressionSolverException, BinaryExpressionSolverTokenManager</td>
</tr>
<tr>
<td>SystemTest</td>
<td>AllTest, DNFControlTest, TruthTableTest, VariableNegationControlTest</td>
</tr>
<tr>
<td>Default</td>
<td>MainExeGUI</td>
</tr>
</tbody>
</table>

The `BooleanVariables` package contains Entity classes for the evaluation of a Boolean expression, the `Controls` package contains all the Control classes, the `Gui` package contains all the Boundary (GUI) classes, `Parser1` contains classes to parse a string according to a Boolean expression grammar and creates the syntax tree, and the classes in package `Parser2` use this parse tree to evaluate the expression for specific Boolean variable values. Parsers were created with JavaCC. `SystemTest` contains JUnit system test cases that interact directly with the application logic classes, specifically classes in the `Controls` package. The `Default` package contains the main class. The author of the SUT and the functional system logic tests is not the author of this thesis.

Later, the GUI of the SUT was updated to fit with the GUI functional system testing tool but no change was made to the remaining packages, including the test suite (Appendix I). For instance, the original UI was made of tabbed panes, with one pane for each of the three functionalities. Panes are not correctly handled by GUITAR and we changed that to new windows that are created by clicking on buttons. These changes and other lessons we learnt when using GUITAR are detailed in Appendix I. Thanks to the
use of the Entity-Control-Boundary (ECB) principle, those changes had no impact on the Control and Entity classes and therefore had no impact on the system-level JUnit tests.

Our case study has one main window with three buttons (Figure 9). Clicking on any of the buttons generates a child window to handle the corresponding functionality.

**FIGURE 9. MAIN WINDOW OF THE BOOLEAN LOGIC SOFTWARE APPLICATION**

The truth table window (Figure 10) accepts a string representing a Boolean expression (text field) and generates its truth table upon request (Compute Truth Table button) according to different user-selected formats (with strings false and true, or characters F and T, or characters 0 and 1). The truth table of the provided Boolean expression is shown in a text area at the bottom of the window. If the user enters a string that is not recognized as a Boolean expression by the parser (according to the grammar it uses), a parsing error message appears in a different pop-up window.

**FIGURE 10. TRUTH TABLE WINDOW OF THE BOOLEAN LOGIC SOFTWARE APPLICATION**
The DNF Expression window (Figure 11) computes (Compute DNF button) the disjunctive normal form of the Boolean expression provided in the top text field and displays the result in the bottom text area. If there is a problem in the format of the input (e.g., not a recognizable Boolean expression), the SUT’s parser triggers a window that contains the parsing error message.

**Figure 11. DNF Expression Window of the Boolean Logic Software Application**

The third window (Figure 12) is an interface for an implementation of the variable negation testing technique [186]. The input must be provided as a series of terms of the DNF of a Boolean expression. The user must enter those terms in the input text field(s), one term per text field. If the three default text fields are not enough (i.e., the DNF has more than three terms), the user clicks the AddProductTerm button to add a new text field. The user can then click the Compute Var. Neg. button and the result appears in the output text area. The user can press on the Show Test Cases button and the program shows a table with a set of test inputs satisfying the test objectives of the variable negation testing technique.
We report that Table 3 shows only the original structure of the SUT as it is updated. Since the GUI functional system testing tool (GUITAR) was deficient in testing the whole SUT, we made a set of steps to modify the GUI. These steps enabled us to reach our goal in measuring coverage. The modification of the GUI takes into account that our objective is not to contribute to GUITAR. We selected this solution from a set of solutions that we investigated. Some of those solutions try to make changes to GUITAR itself instead of the SUT (See Appendix I). Some other solutions are related to changing the default configuration settings of GUITAR (e.g., changing the length of test cases, the number of test cases, or the test cases generation algorithm). A second solution is about adding extra code to the SUT that will lead GUITAR to perform better (e.g., hard-coding the input inside the source code of SUT). A third solution is about changing the default text used as input in GUITAR. Each of those solutions has its own drawbacks. For example, a test case with a length of five is not better than a test case with a length equal
to three in terms of structural coverage of the code. We proved experimentally that increasing the length of test cases does not increase the coverage of the code (See Appendix I).

In our solution, we changed the GUI of the SUT. Since the SUT is equipped with an ECB design principle, we had no difficulty in changing the GUI. In other words, it was possible to change the GUI without changing the business logic. On the first try, we changed the GUI by putting all components in one window; however, GUITAR did not work well with this GUI (same issues as those faced before). On the second try, we used menus in the GUI, but GUITAR did not click the push buttons. The implementation of a new GUI was done by adding three classes to the GUI package: TruthTableExeFrame, DNFExeFrame, and VariableNegationExeFrame. Then, the class MainExeFrame was modified to make calls to those GUI classes instead of making direct calls to existing classes: TruthTableComponent, DNFComponent, and VariableNegationComponent. Thus, the three newly added classes extended the class JFrame. The three new classes also receive calls from DNFExeFrame and pass them to the three Component classes. In other words, they work as a middle layer inside the GUI tier. Finally, the modification of the class MainExeFrame was minor as well. Three internal classes are added to this class to handle the calls from this class to the three newly added classes. The addition of these new internal classes minimized modifications to the existing structure of the class MainExeFrame.
4.2 Measurement

Our comparative study is between GUI system testing and functional system logic testing. We will use two measures during this comparison: cost (section 4.2.1) and structural coverage (section 4.2.2).

4.2.1 Measuring cost

We are interested in test cost. We use the number of test cases as a surrogate measure of cost. Ideally a cost measure should account for elements such as the effort to set up tests, write test scripts, define oracles, and execution time. Since each system test case executing directly on the application logic is made of a single call to a method of a Control class realizing one functionality, we measure test suite cost as the number of test cases in a test suite. Each GUITAR test is made of a series of widget triggers and we measure the cost of a GUITAR test suite as the number of its test cases.

As discussed later in the chapter, an automatically generated GUITAR test does not necessarily click on a Compute button and therefore not every GUITAR test triggers a functionality of the SUT. We also count the number of GUITAR tests that contain a click on a Compute button as a measure of a GUITAR test suite. We believe this is a cost measure of GUITAR test suites that is comparable to cost measure of GUI functional system test suites.
4.2.2 Measuring structural coverage

We measure structural coverage of each testing campaign, specifically line coverage (i.e., the percentage of lines executed by test runs) and branch coverage (i.e., the percentage of branches executed by test runs) using COBERTURA [187].

In order to have a fair comparison between GUI system tests and functional logic system tests, since system logic tests interact directly with Control classes thereby bypassing the GUI, we do not measure coverage of the GUI classes (in package Gui, Table 3). For similar reasons we do not measure coverage of the system test classes (in package SystemTest as well as class CheckTestCase) since these are not exercised by the GUI tests, nor the main class (MainExeGUI). In other words, we only measure coverage of individual Control and Entity classes, which code should be exercised by both GUI functional system tests and functional system logic tests, and total coverage for those classes. We therefore measure coverage of 19 of the classes listed in Table 3.

When reporting on structural coverage later in this document, we only mention coverage of the 12 classes (Table 3) that are actually exercised: BinaryExpressionSolver, BinaryExpressionSolverTokenManager, BooleanVariable, Cube, DNFControl, LogicalExpressionParser, LogicalExpressionParserTokenManager, SetOfBooleanVariables, SimpleCharStream, Token, TruthTableControl, and VariableNegationControl.

Only those classes are instrumented for structural coverage measurement during all our experiments. Hence, we ensure a fair comparison of coverage between the same
set of instrumented classes: once in a GUI functional system testing experiment and another time with functional system logic testing.

The list of classes in packages which are excluded from coverage are: In package Controls, class CheckTestCase, is excluded because it is part of the system logic testing infrastructure; package Gui is completely excluded because it contains all the GUI classes. In the package Parser1, class LogicalExpressionParserConstants is excluded because it only defines constants; ParseException is excluded because it defines parsing exception handling; TokenMgrError is excluded because it is only related to error handling; in package Parser 2, class BinaryExpressionSolverConstants is excluded because it only defines constants and BinaryExpressionSolverException is excluded because it defines parsing exception handling; the package SystemTest is completely excluded because it only defines tests; finally, in the default package, class MainExeGUI is excluded because it is just the main class that launches the application.

The changes to the GUI do not affect structural measures because, thanks to the ECB design principle, changes only took place in the Gui package, in which structural coverage is not measured.

4.2.3 Assumption

For our experiment on measuring structural coverage, we assume that if the structural coverage obtained by each of those two types of testing (GUI functional system testing and functional system logic testing) are equal, then those two types of testing are
doing the same thing. Hence it is enough to have one of them for verifying the
application logic.

4.3 Experiments—Functional system logic testing

The functional system logic test suite exercising the application logic code
consists of 11 JUnit black-box tests (Appendix K). Eight of those tests exercise the truth
table functionality and hence they are applied on class TruthTableControl; three of
those tests exercise the DNF generation functionality and therefore exercise the
DNFControl class; and three tests exercise the variable negation functionality and
therefore exercise the VariableNegationControl class.

Functional system logic testing was performed by the original author of the
software prior to our experiments and the test cases were not subsequently changed.
Specifically, we did not try to improve the test suite, for instance to improve structural
coverage, since we are using it as a basis of comparison with GUI tests. System tests are
JUnit tests for each of the three main functionalities: three different classes in package
SystemTest.

We have access to one functional system logic test suite for each functionality,
which we used in three different experiments (experiments F, G, and H below), and made
the union of all those test cases to produce a larger, more comprehensive test suite, which
we use in experiment I (using class AllTest in package SystemTest).

Experiment F only uses the truth table functional system logic test suite, which
has six test cases. Each test case uses one Boolean expression as an input. These Boolean
expressions\textsuperscript{2} are: “\(x \text{ AND } (y \text{ OR } z)\)”, “\(x \text{ xor } b\)”, “\(a \text{ or } (\text{not } b)\)”, “\(a \text{ XNOR } (b \text{ nand } R)\)”, “\(a \text{ NOR } b\)”, “\((A \text{ AND } d) \text{ XOR } (\text{not } (C \text{ XNOR } (\text{NOT } b)))\)”. These tests trigger only the truth table functionality.

Experiment G only uses the DNF functional system logic test suite, which has two test cases, with Boolean input variables “\((a \text{ or } b) \text{ and } c\)” and “\((a \text{ AND } b) \text{ XOR } c\)”, respectively.

Experiment H only uses the functional system logic test suite exercising the variable negation functionality, which has three test cases. The first test case has two input Boolean expressions: Term 1 is “\(a \text{ and } b \text{ and } \text{not } c\)” and term 2 is “\(a \text{ and } d\)”. The second test case has four input Boolean expressions: Term 1 is “\(x \text{ and } \text{not } y \text{ and } z\)”, term 2 is “\(y \text{ and } w\)”, term 3 is “\(\text{not } x \text{ and } w\)”, and term 4 is “\(y \text{ and } \text{not } z\)”. The third test case has two input Boolean expressions: Term 1 is “\(x \text{ and } \text{not } y \text{ and } z\)” and term 2 is “\(y \text{ and } w\)”. Experiment I is the union of the three previous test suites and is therefore made of 11 test cases.

4.4 Experiments—GUI functional system testing

We first introduce GUITAR’s use with our case study (section 4.4.1) and then discuss experiments we conducted and their design (sections 4.4.2 to 4.4.7). Table 4 shows the details of the experimental plan.

\textsuperscript{2} Notice that Boolean variables and Boolean operators sometimes appear in caps and sometimes they do not. This was set up by the author of the test cases to exercise different aspects of the parser.
<table>
<thead>
<tr>
<th>Main ID</th>
<th>Main experiment description</th>
<th>Test input source</th>
<th>Sub-ID</th>
<th>Used Boolean expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Truth table functionality only (valid input). Invalid input for other functionalities</td>
<td>Functional system logic test</td>
<td>1</td>
<td>x AND (y OR Z)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>a xor b</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>a or (not b)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>a XNOR (b nand R)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td>a NOR b</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td>(A AND d) XOR (not(C XNOR (NOT b)))</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td>Randomization of all previous Boolean expressions [A.1-A.6] in such a way that each Boolean expression will be used at least once</td>
</tr>
<tr>
<td>B</td>
<td>DNF functionality only (valid input). Invalid input for other functionalities.</td>
<td>Functional system logic test</td>
<td>1</td>
<td>(a or b) and c</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>(a AND b) XOR c</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>Randomization of the previous two Boolean expressions [B.1,B.2] in such a way that each Boolean expression will be used at least once</td>
</tr>
<tr>
<td>C</td>
<td>Variable negation functionality only (valid input). Invalid input for other functionalities.</td>
<td>Functional system logic test</td>
<td>1</td>
<td>Term 1: x and not y and z</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Term 2: y and w</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Term 3: not x and w</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>Term 1: A and B and not C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Term 2: A and D</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Term 3: GUITAR's default input (No input by tester)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>Term 1: A and B and C and D</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Term 2: A and B and not E</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Term 3: GUITAR's default input</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>Randomization of the previous two Boolean expressions [C.1, C.2, C.3] in such a way that each Boolean expression will be used at least once and all terms will be filled (GUITAR's default text will not be used)</td>
</tr>
<tr>
<td>D</td>
<td>All three functionalities.</td>
<td>Functional system logic test</td>
<td>1</td>
<td>Truth table [A.1]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DNF component [B.1]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Variable negation [C.1]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>Truth table [A.7]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DNF component [B.3]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Variable negation [C.4] (It was [C.3] in the previous batch of experiments)</td>
</tr>
<tr>
<td>E</td>
<td>All three functionalities.</td>
<td>Internet</td>
<td>1</td>
<td>Truth table &gt; 17 Boolean expressions</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DNF component &gt; 11 Boolean expressions</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Variable negation &gt; 140 Boolean expressions</td>
</tr>
</tbody>
</table>
4.4.1 Using GUITAR on the case study

When using GUITAR for GUI functional system testing on our case study we generated a test suite using GUITAR’s default setting, as this proved to be effective in a number of experiments [3].

We obtained 200 test cases by using the default value L=3. Recall (section 2.4.4) that each of those tests is a traversal of the EFG representing the entire GUI of the SUT that the Ripper creates. So, in theory, each test case may trigger events for widgets related to none of the functionalities (e.g., a test containing only menu items), one of the functionalities, two of the functionalities or all three functionalities, though not necessarily triggering the application logic code that realizes those functionalities. For instance, we obtained a test case that provides a text input for the text field used in the truth table functionality and then a text input for the text field used in the DNF functionality, without any button click (i.e., no application logic code triggered).

Analyzing the 200 test cases, we found that 17 provide an input (through a text field) to the truth table functionality, 11 provide an input (through a text field) to the DNF functionality, and 129 provide an input (through several text fields—recall several are needed, Figure 12) to the variable negation functionality. The higher number of tests is due to the higher complexity of the variable negation GUI compared to the other two GUIs. For instance, the variable negation GUI has more widgets, which means there are more possibilities to involve them in tests.

We also note that GUITAR does not know when in a test case, i.e., in a sequence of events, to trigger a functionality, i.e., when to click on a “compute” button. As a
result, when a test case has events to fill a text field used in the truth table functionality, this event is not necessarily followed by a “click” on the Compute Truth Table button to trigger the functionality; the text case may very well continue with a different window, for a different functionality, without triggering the truth table functionality (code).

What we know for a fact, after having checked the 200 tests, is that in the 17 (resp., 11, 129) tests that provide a string to a text field used for the truth table (resp., DNF, variable negation) functionality, at least one test case has a text field followed by a click on a “compute” button in the same window, ensuring that the application logic code of each functionality is at least triggered once. A “compute” button triggering some functionality of the application logic was clicked eight times within the replay of 200 test cases. We also know that out of the 200 tests, 77 do not use any text field that is required by any of the three functionalities.

Last we note that during the replaying process, GUITAR uses its own set of strings for text fields. These are not Boolean expressions. Using the default strings provided by GUITAR would not trigger functionalities but would result in parsing errors: the application code starts to parse the input string and reports a parsing error (in an error window) if the string is not a Boolean expression that can be recognized by the parser.

Based on our experience (Appendix I), we agree on observations reported by Ganov [1] that GUITAR is not smart enough to generate inputs to be used in testing. We used GUITAR in a modified way such that test cases are updated with valid input. Since

---

3 Some of those tests provide a string to text fields belonging to more than one functionality window. A test may therefore contribute to more than one of these three sets with 17, 11, and 129 elements. Thus it would be wrong to sum up these numbers and compare the result to the total number of tests (200).
GUITAR is not able to generate valid input for all functionalities of software, some of them will be not be triggered in testing.

One can specify test inputs (i.e., Boolean expressions) for GUITAR tests in many different ways (Appendix I). In an attempt to be as systematic as possible and obtain coverage results that can be compared to those obtained by functional system logic tests, we devised a number of experiments (see the following five sections), each one having a specific test input selection procedure.

4.4.2 Experiment A for truth table

This experiment tests the truth table functionality only. Hence, a valid input (i.e., a Boolean expression) is used each time a text field for the truth table functionality appears in a test case, and we use GUITAR’s default strings, which are not Boolean expressions, for any other text field. The question is, however: Which Boolean expression should be used when a GUITAR test case needs one to exercise the truth table functionality? To have a fair comparison with system logic tests we used their input Boolean expressions and split experiment A into seven experiments. Since the system logic tests use six different Boolean expressions, we design six variations of experiment A where each one systematically uses one of those six Boolean expressions systematically when the text field for the truth table functionality needs an input. We design a seventh variation of experiment A where all those Boolean expressions are used randomly.

In experiment A.1, we use input “x AND (y OR z)” systematically each time we need such an input for a text field that is needed for the truth table. In experiment A.2,
we systematically use “a xor b”. In experiment A.3, we use “a or (not b)”. In experiment A.4, we use “a XNOR (b nand R)”. In experiment A.5, we use “a NOR b”. In experiment A.6, we use “(A AND d) XOR (not(C XNOR (NOT b)))”. In experiment A.7, we randomly select one of the above six inputs each time we need one, while ensuring that each input is used at least once.

4.4.3 Experiment B for Disjunctive Normal Form

This experiment tests the DNF functionality only. Hence, a valid input (i.e., a Boolean expression) is used each time a text field for the DNF functionality appears in a test case, and we use GUITAR’s default strings for any other text field. We proceed similarly to experiment A (for the truth table): we create several sub experiments; since the functional system logic test suite uses two inputs, we create two sub-experiments where each input is systematically used; and we create a third experiment where test inputs for DNF are randomly selected from these two functional system logic inputs, ensuring that each Boolean expression is used at least once.

4.4.4 Experiment C for variable negation

This experiment tests the variable negation functionality only. Hence, a valid input (i.e., a Boolean expression) is used each time a text field for the variable negation functionality appears in a test case, and we use GUITAR’s default strings for any other text field. Again, we split this experiment, re-using inputs we used for functional system logic testing.
In the first sub-experiment (C.1), each time the variable negation functionality needs an input (i.e., a series of Boolean expressions) we use the following: term 1 is “\(x \text{ and not } y \text{ and } z\)”, term 2 is “\(y \text{ and } w\)”, and term 3 is “\(\text{not } x \text{ and } w\)”. Notice that this is the second functional system logic test input for variable negation except that the last term has been omitted. The reason is that the GUI tests only use three inputs for variable negation: not more, not less. Recall (Figure 11) that the variable negation GUI provides three text fields to the user and a button to add more text fields if needed.

The second experiment (C.2) uses the first functional system logic test input systematically each time the GUITAR test cases require an input for variable negation. The third experiment (C.3) uses the third functional system logic test input, with two terms, in a similar way to the second experiment above. The last experiment (C.4) is a random selection of the inputs used in the first three experiments, ensuring that each input is used at least once.

4.4.5 Remarks on experiment C

The Add product term button is sometimes clicked, thereby adding a fourth text field. When this is the case, the fourth text field is not used to provide a value. However, we do not update a text field that may be created at run time. This may affect the validity of our results. However, we consider such a threat to be negligible. This is due to the fact that there is a low probability of clicking the Compute button in this specific scenario. In other words, updating text fields with valid inputs will increase structural coverage of code only if there is a click on the Compute button. This way, the
text will be used as an input to function in the logic that conducts some Boolean calculation. Hence, coverage will be increased because more code is being used.

The length of each of our test cases is three. We discuss here the length of our test case and not the prefix. We do not discuss the prefix as GUITAR’s authors did not discuss the prefix when they did their experiments.

The size of our test suite is 200 test cases. Let us assume that one of those test cases has an event which clicks the Add product term button. If this event is the last event in the test case, then that would make no difference to coverage of code. This is because there is no click on the Compute button after it.

If it is the first/second event, then the second/third event should be clicking the Compute button. Only then will there be a difference in coverage due to triggering the functionality in the software logic. Hence, to make a difference in the coverage by triggering logic functionality, three conditions must be satisfied: The click on Add product term should be in a test case; it should be the first or second event in that test case; and it should be followed by clicking on the Compute button. Since the Compute button has been clicked only eight times in our test suite, we consider the possibility of having all these conditions satisfied to be low.

4.4.6 Experiment D for the three functionalities

This experiment works on all three functionalities: The truth table, variable negation and DNF together. We performed two experiments.

In D.1, each time an input was needed for the truth table we used that of experiment A.1, each time an input was needed for DNF we used the input of B.1, and
each time an input was needed for variable negation we used that of experiment C.1. We did not have any reason to believe that those first sub-experiments of experiments A, B and C would perform better or worse than the others, and we therefore selected the first sub-experiments.

In D.2, we used the randomized selection of the previous experiments. The occurrences of inputs for the truth table follow the same random selection procedure as experiment A.7. The occurrences of inputs for DNF follow the same random selection procedure as experiment B.3. The occurrences of inputs for variable negation follow the same random selection procedure as experiment C.4.

As mentioned before, we follow the same procedure of input selection used in the previous experiments A.7, B.3 and C.4. We do not use the same selected values which were selected in previous experiments. Rather, we repeat the procedure used in A.7, B.3 and C.4 instead of using the same values used in those experiments.

We want each input to be used at least once and all inputs should have an equal chance of being used an equal number of times. We keep a counter to avoid selecting the same input again unless all inputs in the same group have been selected. That is done by creating a counter for each input item and initializing it to zero. In other words, the same input is selected again only if all inputs in the same set of inputs have been selected once. For example, if input X is selected from the group of inputs used in A.7, we increase a counter attached to this input by one. We check the value of the counter before each selection. The next time we need an input from A7, input X will not be selected because the selection procedure will pick an item from the remaining inputs for which the counter value is equal to zero. Input X could be selected again only when the counter values of all
inputs are equal to 1. There is then a possibility that the input would be reselected. Consequently, our selection mechanism assures that no input will be selected twice while other inputs are not selected at all. The decision to use these combinations of experiments A, B and C in both sub-experiments was made prior to conducting any of the experiments we report on in this document, i.e., prior to obtaining the results of GUI functional system testing coverage.

4.4.7 Experiment E using expressions from the Internet

This experiment works on the three functionalities (truth table, variable negation and DNF) together. However, instead of using GUITAR’s default values or the values used during functional system logic testing, we used Boolean expressions we collected from the Internet [188-195] such that each time a Boolean expression is needed in GUITAR’s test cases we used a different one. The 200 GUITAR test cases require 213 Boolean variable test inputs, which is too much for us to report on in this chapter. The Boolean expressions were mostly available in websites related to academic mathematical topics and computer circuits design. In doing so, we attempted to simulate what a user may do with GUITAR test cases when testing our case study.

4.5 Experimental results

We first discuss the results of experiments F, G, H and I about functional system logic testing directly applied on application logic (i.e., Control) classes (section 4.5.1) and then the results of experiments A, B, C, D and E on GUI functional system testing (section 4.5.2).
4.5.1 Functional system logic testing results (F, G, H, I)

Table 5 shows coverage achieved by the test suites in experiments F, G, H and I. We notice that although the target of experiment F is the truth table functionality, which is directly supported by the TruthTableControl class, only 92% of the lines and branches of this class are covered. It appears from experiments G, H and I that TruthTableControl offers unique services to the other two Control classes (DNFControl and VariableNegationControl), specifically in terms of presenting truth table information in specific formats. We confirmed that without those specific services (i.e., methods in class TruthTableControl) that are only used by other Control classes, the truth table functional system logic test suite would achieve 100% line and branch coverage of class TruthTableControl. Because the test suite specifically targets the truth table functionality, the coverage of the two other Control classes, as well as accompanying classes (e.g., Cube), is zero (line and branch).

The test suite specifically targeting the DNF construction functionality (Experiment G) achieves 100% line and branch coverage of the DNFControl class, i.e., the Control class that implements the logic of the functionality being tested. We also notice that the test cases lead to some coverage of class TruthTableControl, though the functionalities of this class are not called directly by the tests. DNFControl relies on some services (delegation) of TruthTableControl to achieve its own functionality. Not surprisingly, the test suite does not trigger the third functionality: 0 line and branch coverage of VariableNegationControl.
Table 5. Line and Branch coverage for experiments F, G, H and I (class numbers are used later in the text to refer to specific classes instead of their names)

<table>
<thead>
<tr>
<th>Class #</th>
<th>Classes</th>
<th>Experiment F</th>
<th></th>
<th></th>
<th>Experiment G</th>
<th></th>
<th></th>
<th>Experiment H</th>
<th></th>
<th></th>
<th>Experiment I</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Line</td>
<td>Branch</td>
<td>Line</td>
<td>Branch</td>
<td>Line</td>
<td>Branch</td>
<td>Line</td>
<td>Branch</td>
<td>Line</td>
<td>Branch</td>
<td>Line</td>
<td>Branch</td>
</tr>
<tr>
<td>1</td>
<td>BinaryExpressionSolver</td>
<td>0.37</td>
<td>0.30</td>
<td>0.30</td>
<td>0.25</td>
<td>0.25</td>
<td>0.23</td>
<td>0.37</td>
<td>0.30</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>BinaryExpressionSolverTokenManager</td>
<td>0.59</td>
<td>0.49</td>
<td>0.43</td>
<td>0.26</td>
<td>0.39</td>
<td>0.22</td>
<td>0.59</td>
<td>0.49</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>BooleanVariable</td>
<td>0.80</td>
<td>N/A</td>
<td>0.80</td>
<td>N/A</td>
<td>0.80</td>
<td>N/A</td>
<td>0.80</td>
<td>N/A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Cube</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.95</td>
<td>0.94</td>
<td>0.95</td>
<td>0.94</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>DNFControl</td>
<td>0.00</td>
<td>0.00</td>
<td>1.00</td>
<td>1.00</td>
<td>0.00</td>
<td>0.00</td>
<td>1.00</td>
<td>1.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>LogicalExpressionParser</td>
<td>0.37</td>
<td>0.26</td>
<td>0.33</td>
<td>0.21</td>
<td>0.34</td>
<td>0.23</td>
<td>0.37</td>
<td>0.26</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>LogicalExpressionParserTokenManager</td>
<td>0.65</td>
<td>0.60</td>
<td>0.56</td>
<td>0.42</td>
<td>0.52</td>
<td>0.37</td>
<td>0.65</td>
<td>0.60</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>SetOfBooleanVariables</td>
<td>0.93</td>
<td>0.77</td>
<td>0.93</td>
<td>0.77</td>
<td>0.93</td>
<td>0.77</td>
<td>0.93</td>
<td>0.77</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>SimpleCharStream</td>
<td>0.37</td>
<td>0.35</td>
<td>0.30</td>
<td>0.25</td>
<td>0.30</td>
<td>0.25</td>
<td>0.37</td>
<td>0.35</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Token</td>
<td>0.75</td>
<td>1.00</td>
<td>0.75</td>
<td>1.00</td>
<td>0.75</td>
<td>1.00</td>
<td>0.75</td>
<td>1.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>TruthTableControl</td>
<td>0.92</td>
<td>0.92</td>
<td>0.60</td>
<td>0.40</td>
<td>0.70</td>
<td>0.44</td>
<td>1.00</td>
<td>1.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>VariableNegationControl</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.84</td>
<td>0.87</td>
<td>0.84</td>
<td>0.87</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Total coverage</td>
<td>0.42</td>
<td>0.39</td>
<td>0.36</td>
<td>0.27</td>
<td>0.49</td>
<td>0.40</td>
<td>0.60</td>
<td>0.57</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The test suite exercising the variable negation functionality (Experiment H) achieves 84% line and 87% branch coverage of class VariableNegationControl. The uncovered code was a single method that is responsible for printing results in a specific format for program debugging. Without this method, coverage (line and branch) would reach 100%. Similarly to the previous experiment, it is not surprising that the test suite does not cover the whole DNFControl class. We also observe that TruthTableControl is somewhat covered, suggesting VariableNegationControl delegates some work to TruthTableControl.

The union of the three previous test suites (experiment I) is made of 11 test cases. It achieves 100% line and branch coverage for the classes TruthTableControl and DNFControl, and only 84% line and 87% branch coverage of VariableNegationControl (Table 5).
4.5.2 GUI functional system testing results

Figure 13 shows details of line coverage values obtained for different classes covered in experiment A, i.e., for Boolean expressions of varying complexity (for instance, measured as the number of terms the equation contains). We have this figure only once for a single experiment (experiment A) to study the effect of increasing the number of terms and operators of the Boolean expression on the coverage of different classes. Results are similar for other experiments and we do not see a reason for repeating that for every experiment. The Control class related to the truth table functionality (TruthTableControl) and two Entity classes obtained the highest value of coverage. Figure 13 shows how the solver classes (BinaryExpressionSolver and BinaryExpressionSolverTokenManager) have been covered. These levels of coverage would not be possible when using GUITAR’s default test input values.

From this figure we can understand the relation between the complexity of Boolean expressions and coverage. The relation is that having more terms and logical operators in the Boolean expression increases the coverage of the classes that use that expression as an input.

The contribution of classes to the total coverage computed is not the same (i.e., different classes contribute differently to total coverage). For example, TruthTableControl got a higher coverage than the other Control classes. Hence, the contribution of TruthTableControl to the total coverage of the SUT is more than other Control classes. This is essential since the total calculated coverage by COBERTURA is not just the average coverage of all classes’ coverage. Instead, COBERTURA uses its own coverage measurements to calculate the value of total
coverage out of the coverages of individual classes. However, the more coverage each class achieves, the higher the total coverage of SUT will be. As such, the contribution of coverage to the total coverage of the SUT depends on the individual coverage obtained by that class. In addition, the increase in coverage of classes depends on the type of functionality triggered in the GUI. For example, assume that a user entered Boolean expression “(A AND d) XOR (not(C XNOR (NOT b)))” and pressed the Compute button of the truth table functionality. The input will pass to the Control class and will then pass to the solver classes. The obtained coverage will be as the figure shows:

- SetOfBooleanVariables: 0.93
- BooleanVariable: 0.8;
- TruthTableControl: 0.8;
- Token: 0.75;
- LogicalExpressionParserTokenManager: 0.6;
- LogicalExpressionParser: 0.54;
- BinaryExpressionSolverTokenManager: 0.49;
- BinaryExpressionSolver: 0.31;
- SimpleCharStream: 0.3;
- DNFCcontrol: 0.09;
- VariableNegationControl: 0.03;
- Cube: 0.

It is noticeable that Control classes related to the DNF and variable negation functionalities got the lowest coverage because they are not related to the truth table calculation functionality of Boolean expressions. It is noticeable as well that classes which are highly involved in the computation process got better coverage than other classes which either do not contribute to the calculation of functionality or are only slightly involved in the computation process.
Table 6 shows the coverage results for all the classes of the SUT for the variations of Experiment A.

The table shows how the coverage increases as the complexity of the test input Boolean expression increases. For example, in experiment A.6 line and branch coverage are maximal, whereas they are minimal in experiment A.2. In experiment A.7, the coverage values are at their minimum, which may look like a contradiction since we use all test inputs. However, remember that not every filled text field for the truth table is followed by a click on the Compute button event in test cases; in experiment A.7, it just...
happened that when a click occurs, the Boolean expression we selected at random for the text field was always the simplest one \((a \text{xor} b)\).

**Table 6. Total Line and Branch Coverage for Experiments A**

<table>
<thead>
<tr>
<th>Experiment Number</th>
<th>Boolean Expression</th>
<th>Line Coverage</th>
<th>Branch Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x AND (y OR Z)</td>
<td>0.35</td>
<td>0.25</td>
</tr>
<tr>
<td>2</td>
<td>a xor b</td>
<td>0.33</td>
<td>0.23</td>
</tr>
<tr>
<td>3</td>
<td>a or (not b)</td>
<td>0.36</td>
<td>0.26</td>
</tr>
<tr>
<td>4</td>
<td>a XNOR (b nand R)</td>
<td>0.37</td>
<td>0.28</td>
</tr>
<tr>
<td>5</td>
<td>a NOR b</td>
<td>0.34</td>
<td>0.24</td>
</tr>
<tr>
<td>6</td>
<td>(A AND d) XOR (not(C XNOR (NOT b)))</td>
<td>0.40</td>
<td>0.33</td>
</tr>
<tr>
<td>7</td>
<td>Random Boolean expressions</td>
<td>0.33</td>
<td>0.23</td>
</tr>
</tbody>
</table>

Table 7 shows the results for Experiment B. We can make observations similar to the ones for experiment A regarding the low coverage value obtained with the random selection of inputs. The test inputs used in the two test cases have a similar complexity, resulting in similar coverage levels.

**Table 7. Total Line and Branch Coverage for Experiments B**

<table>
<thead>
<tr>
<th>Experiment Number</th>
<th>Boolean Expression</th>
<th>Line Coverage</th>
<th>Branch Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(a or b) and c</td>
<td>0.36</td>
<td>0.25</td>
</tr>
<tr>
<td>2</td>
<td>(a AND b) XOR c</td>
<td>0.37</td>
<td>0.27</td>
</tr>
<tr>
<td>3</td>
<td>Random Boolean expressions</td>
<td>0.36</td>
<td>0.25</td>
</tr>
</tbody>
</table>

Table 8 shows the results of Experiment C. It is noticeable that in experiments C.2 and C.3 we obtained low coverage values in both line and branch coverage. Clearly, this is due to the lack of a third valid (Boolean) input in both experiments.
Table 8. Total Line and Branch Coverage for Experiments C

<table>
<thead>
<tr>
<th>Experiment Number</th>
<th>Line Coverage</th>
<th>Branch Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.52</td>
<td>0.43</td>
</tr>
<tr>
<td>2</td>
<td>0.16</td>
<td>0.10</td>
</tr>
<tr>
<td>3</td>
<td>0.16</td>
<td>0.10</td>
</tr>
<tr>
<td>4</td>
<td>0.52</td>
<td>0.43</td>
</tr>
</tbody>
</table>

Table 9 shows the detailed results from experiment C. One can notice that the values of line coverage for BinaryExpressionSolver are equal to zero in both experiments C.2 and C.3. The SUT parses all inputs, discovers that one is not a valid Boolean expression, brings this to the attention of the user and stops: the application logic does not execute. Thus, solver classes are not used. This observation shows how the input affects the coverage in our experiment and hence justifies our test selection procedure. Specifically, giving a variety of valid and invalid inputs to the SUT during the same Replay is good practice to simulate different scenarios that achieve different coverage values. In other words, it was necessary to see how the program would react by trying different settings of test input. One option was to measure the coverage of code when using two different types of inputs: valid and invalid input. For example, giving a valid input to the truth table functionality and invalid input for the other two functionalities (i.e., DNF and variable negation). We applied the same practice for DNF and variable negation functionalities. Finally, in our experiment D, which we explain next, all functionalities are provided with valid inputs by updating all the test cases with valid inputs.
Table 9. Experiment C, Coverage of GUI Functional System Testing for Variable Negation Functionality

<table>
<thead>
<tr>
<th>Number</th>
<th>Classes in this Package</th>
<th>Experiment</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Line Coverage</td>
<td>Branch Coverage</td>
<td>Line Coverage</td>
<td>Branch Coverage</td>
<td>Line Coverage</td>
</tr>
<tr>
<td>1</td>
<td>BinaryExpressionSolver</td>
<td>0.25</td>
<td>0.00</td>
<td>0.00</td>
<td>0.25</td>
<td>0.23</td>
<td>0.13</td>
</tr>
<tr>
<td>2</td>
<td>BinaryExpressionSolverTokenManager</td>
<td>0.39</td>
<td>0.00</td>
<td>0.00</td>
<td>0.39</td>
<td>0.22</td>
<td>0.20</td>
</tr>
<tr>
<td>3</td>
<td>BooleanVariable</td>
<td>0.80</td>
<td>N/A</td>
<td>0.20</td>
<td>0.80</td>
<td>N/A</td>
<td>0.50</td>
</tr>
<tr>
<td>4</td>
<td>Cube</td>
<td>0.95</td>
<td>0.09</td>
<td>0.09</td>
<td>0.95</td>
<td>0.94</td>
<td>0.48</td>
</tr>
<tr>
<td>5</td>
<td>DNFControl</td>
<td>0.09</td>
<td>0.00</td>
<td>0.00</td>
<td>0.09</td>
<td>0.00</td>
<td>0.09</td>
</tr>
<tr>
<td>6</td>
<td>LogicalExpressionParser</td>
<td>0.52</td>
<td>0.42</td>
<td>0.42</td>
<td>0.52</td>
<td>0.57</td>
<td>0.47</td>
</tr>
<tr>
<td>7</td>
<td>LogicalExpressionParserTokenManager</td>
<td>0.52</td>
<td>0.42</td>
<td>0.23</td>
<td>0.53</td>
<td>0.38</td>
<td>0.47</td>
</tr>
<tr>
<td>8</td>
<td>SetOfBooleanVariables</td>
<td>0.93</td>
<td>0.21</td>
<td>0.21</td>
<td>0.93</td>
<td>0.77</td>
<td>0.57</td>
</tr>
<tr>
<td>9</td>
<td>SimpleCharStream</td>
<td>0.30</td>
<td>0.26</td>
<td>0.26</td>
<td>0.30</td>
<td>0.25</td>
<td>0.28</td>
</tr>
<tr>
<td>10</td>
<td>Token</td>
<td>0.75</td>
<td>1.00</td>
<td>1.00</td>
<td>0.75</td>
<td>1.00</td>
<td>0.75</td>
</tr>
<tr>
<td>11</td>
<td>TruthTableControl</td>
<td>0.70</td>
<td>0.08</td>
<td>0.00</td>
<td>0.70</td>
<td>0.44</td>
<td>0.39</td>
</tr>
<tr>
<td>12</td>
<td>VariableNegationControl</td>
<td>0.94</td>
<td>0.03</td>
<td>0.00</td>
<td>0.94</td>
<td>0.96</td>
<td>0.49</td>
</tr>
</tbody>
</table>

Table 10 and Table 11 show the results of experiment D. As expected, the overall results are better than when testing each functionality separately, but coverage only reaches 58% (line) and 52% (branch) even though we used the same test inputs as with functional system logic testing. Again, this is due to the fact that not all text field input is followed by a button click in GUITAR tests. The random distribution of Boolean expressions in experiment D.2 proved to give the same line coverage as when we used one arbitrary Boolean expression, as in experiment D.1. However, the branch coverage is higher, which may indicate increased sensitivity of branch coverage to the test input.
In light of the results of GUI functional system testing coverage, another interesting combination would have been A.6, B.2 and C.1, as this maximizes coverage for each functionality separately.

**TABLE 10. TOTAL LINE AND BRANCH COVERAGE FOR EXPERIMENT D**

<table>
<thead>
<tr>
<th>Experiment Number</th>
<th>Line Coverage</th>
<th>Branch Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.58</td>
<td>0.50</td>
</tr>
<tr>
<td>2</td>
<td>0.58</td>
<td>0.52</td>
</tr>
</tbody>
</table>

**TABLE 11. DETAILED RESULTS FOR EXPERIMENT D, COVERAGE OF GUI FUNCTIONAL SYSTEM TESTING FOR THREE OPERATIONS EXPERIMENTS**

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Classes in this Package</th>
<th>1</th>
<th>2</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td></td>
<td>Line Coverage</td>
<td>Branch Coverage</td>
<td>Line Coverage</td>
</tr>
<tr>
<td>1</td>
<td>BinaryExpressionSolver</td>
<td>0.29</td>
<td>0.25</td>
<td>0.35</td>
</tr>
<tr>
<td>2</td>
<td>BinaryExpressionSolverTokenManager</td>
<td>0.46</td>
<td>0.29</td>
<td>0.51</td>
</tr>
<tr>
<td>3</td>
<td>BooleanVariable</td>
<td>0.80</td>
<td>N/A</td>
<td>0.80</td>
</tr>
<tr>
<td>4</td>
<td>Cube</td>
<td>0.95</td>
<td>0.94</td>
<td>0.95</td>
</tr>
<tr>
<td>5</td>
<td>DNFControl</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>6</td>
<td>LogicalExpressionParser</td>
<td>0.55</td>
<td>0.58</td>
<td>0.37</td>
</tr>
<tr>
<td>7</td>
<td>LogicalExpressionParserTokenManager</td>
<td>0.59</td>
<td>0.47</td>
<td>0.62</td>
</tr>
<tr>
<td>8</td>
<td>SetOfBooleanVariables</td>
<td>0.93</td>
<td>0.77</td>
<td>0.93</td>
</tr>
<tr>
<td>9</td>
<td>SimpleCharStream</td>
<td>0.30</td>
<td>0.25</td>
<td>0.30</td>
</tr>
<tr>
<td>10</td>
<td>Token</td>
<td>0.75</td>
<td>1.00</td>
<td>0.75</td>
</tr>
<tr>
<td>11</td>
<td>TruthTableControl</td>
<td>0.96</td>
<td>0.84</td>
<td>0.96</td>
</tr>
<tr>
<td>12</td>
<td>VariableNegationControl</td>
<td>0.94</td>
<td>0.95</td>
<td>0.94</td>
</tr>
</tbody>
</table>

Table 11 shows the detailed results of experiment D. All Control classes got coverage almost equal to one. This is due to the fact that all test cases were updated with valid input. Solver, parser and input handling classes got the lowest coverage. We do not think coverage of those solver classes can be increased by increasing the complexity of
Boolean expressions. In fact, in our whole experiment, regardless of the complexity of Boolean expression, the coverage of solver classes was between 25% and 35%. Hence, we do not think that choosing more complex input would increase the coverage of those classes. Other helping classes like SetOfBooleanVariables and Cube got high coverage, at almost 95%. The parser class LogicalExpressionParser got almost 50% coverage. To increase the coverage of this class, there might be a need to try different types of invalid input and not just a blank space or GUITAR’s default text.

Table 12 shows the values of line coverage and branch coverage in experiment E for all the classes, as well as total coverage. As the table shows, the total coverage was 0.57% for line coverage and 0.51% for branch coverage.
### Table 12. Line and Branch coverage for experiment E

<table>
<thead>
<tr>
<th>Class #</th>
<th>Classes Name</th>
<th>Line Coverage</th>
<th>Branch Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BinaryExpressionSolver</td>
<td>0.29</td>
<td>0.25</td>
</tr>
<tr>
<td>2</td>
<td>BinaryExpressionSolverTokenManager</td>
<td>0.50</td>
<td>0.34</td>
</tr>
<tr>
<td>3</td>
<td>BooleanVariable</td>
<td>0.80</td>
<td>N/A</td>
</tr>
<tr>
<td>4</td>
<td>Cube</td>
<td>0.95</td>
<td>0.94</td>
</tr>
<tr>
<td>5</td>
<td>DNFControl</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>6</td>
<td>LogicalExpressionParser</td>
<td>0.37</td>
<td>0.26</td>
</tr>
<tr>
<td>7</td>
<td>LogicalExpressionParserTokenManager</td>
<td>0.62</td>
<td>0.53</td>
</tr>
<tr>
<td>8</td>
<td>SetOfBooleanVariables</td>
<td>0.93</td>
<td>0.81</td>
</tr>
<tr>
<td>9</td>
<td>SimpleCharStream</td>
<td>0.30</td>
<td>0.25</td>
</tr>
<tr>
<td>10</td>
<td>Token</td>
<td>0.75</td>
<td>1.00</td>
</tr>
<tr>
<td>11</td>
<td>TruthTableControl</td>
<td>0.96</td>
<td>0.84</td>
</tr>
<tr>
<td>12</td>
<td>VariableNegationControl</td>
<td>0.94</td>
<td>0.95</td>
</tr>
<tr>
<td></td>
<td>Total coverage</td>
<td>0.57</td>
<td>0.51</td>
</tr>
</tbody>
</table>

The observations for experiment D hold for experiment E as well. This is specifically the case for the discussion about Table 11, which shows that the detailed results of experiment D apply for Table 12. We notice that the Control classes got the highest coverage value of all the classes in the SUT. This is due to the structure of the software: beside the Control classes, the rest of the software is basically a parser whose code was automatically generated by JavaCC. Moreover, the parser is only triggered through the Control classes, which causes problems related to the controllability of its code. Unit tests of the parser would help us increase this coverage.
4.6 Results analysis and cost of functional system logic testing vs. GUI functional system testing

In this section, we discuss the results presented earlier on functional system logic testing and GUI functional system testing. We start by discussing controllability (section 4.6.1). Section 4.6.2 describes the cost. Finally, we discuss structural coverage (section 4.6.3).

4.6.1 Controllability

In addition to comparing structural coverage between GUI functional system testing and system logic testing (discussed later), there is a need to justify how easy it was to obtain such coverage measurements. In other words, achieving such coverage is not done by using the same amount of effort or the same ease in each of functional system logic testing and GUI functional system testing. In fact, we compare automatically generated test cases with user-developed test cases. It is expected that user-developed test cases would be more efficient at covering the code than automatically generated test cases. However, there are redundancies among test practices [196] (i.e., testing the same code twice). We therefore discuss the concept of controllability to give another perspective of our obtained results. One common issue attached to software testing is how to provide the right values to the software. Software controllability describes how easy it is to provide a program with the needed inputs, in terms of values, operations and behaviours [6]. The providing of input is needed to reach an intent. Even if there is a way to provide an input, the difficulty is to do so in order to achieve the goal (intent). For example, it is easy to control a piece of software for which all inputs are
values entered from a keyboard [6, 28, 29]. On the other hand, when the software gets its input values from sensors, it is difficult to control.

Typically, a tester has less control with component/system testing than with unit testing. Controllability can also refer to the ease of reaching some predefined level of coverage, i.e., to exercise specific behaviour or pieces of code: it is more difficult to reach high coverage of units with system testing than with unit testing. In general, with a higher level of testing (e.g., system testing) it is harder to trigger specific elements of the code/functionality provided by lower levels of the code than with a lower level of testing (e.g., unit testing). When doing integration testing, it is harder to trigger specific statements of the code than when testing those units of the code directly. Similarly, when doing GUI functional system testing, it is harder to trigger elements of the code than when doing functional system logic testing, and even more so than when doing unit testing. Hence, it is safe to assume that GUI functional system testing would achieve lower coverage than unit testing, although there is no proof for that.

4.6.2 Cost

We discussed the set-up of GUITAR and the tailoring of the tests it generates to our SUT in section 4.4. The test input selection using GUITAR involved a lot of effort. This is seen, for example, when looking at the details of the EFG and the generated tests, and therefore time. It took significantly less effort and time for the original author of the code to generate system tests. Although we do not have precise measurements of these two-time efforts, we indicate the existence of a big difference. In terms of test suite execution, executing the entire functional system logic test suite is almost instantaneous
which is a single replay of the GUI test suite takes around 20 minutes on an ordinary computer with Intel(R) Core(TM) i7-2670QM CPU @ 2.2 GHz with 8 GB of RAM.

Yet another comparison one can make is about the number of tests: the functional system logic test suite has 11 tests and the GUI functional system test suite has 200 tests. All the tests for functional system logic testing call the functionalities while GUI system tests call functionalities only eight times (i.e., an input is provided to a text field and followed by a click event on the appropriate Compute button).

4.6.3 Structural coverage

The overall coverage results of our experiments are reported in Table 13 for our experiments that were split into sub-experiments: we show the average coverage values and standard deviations (in parenthesis).

Results show that the coverage of functional system logic testing is better than that of GUI functional system testing in all pairs of comparable experiments, i.e., for experiments targeting the same functionality: F and A, H and C, and I, D and E. The exception to this is experiment G and B, in which the coverage values are equal. Experiment C has the highest values of standard deviation. This is due to the lack of one input in two sub-experiments (only partial input was provided in the tests because tests do not have to fill all Boolean terms).
<table>
<thead>
<tr>
<th>ID</th>
<th>Main Experiment</th>
<th>Line Coverage</th>
<th>Branch Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Average Total (Standard Deviation)</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>Truth table</td>
<td>0.35 (0.03)</td>
<td>0.26 (0.04)</td>
</tr>
<tr>
<td>B</td>
<td>DNF</td>
<td>0.36 (0.01)</td>
<td>0.26 (0.01)</td>
</tr>
<tr>
<td>C</td>
<td>Variable negation</td>
<td>0.34 (0.21)</td>
<td>0.27 (0.19)</td>
</tr>
<tr>
<td>D</td>
<td>All three operations</td>
<td>0.58 (0.01)</td>
<td>0.51 (0.02)</td>
</tr>
<tr>
<td>E</td>
<td>Expressions from Internet</td>
<td>0.57</td>
<td>0.51</td>
</tr>
<tr>
<td>F</td>
<td>Truth table</td>
<td>0.42</td>
<td>0.39</td>
</tr>
<tr>
<td>G</td>
<td>DNF</td>
<td>0.36</td>
<td>0.27</td>
</tr>
<tr>
<td>H</td>
<td>Variable negation</td>
<td>0.49</td>
<td>0.40</td>
</tr>
<tr>
<td>I</td>
<td>All three operations</td>
<td>0.60</td>
<td>0.57</td>
</tr>
</tbody>
</table>

Remember that we excluded some classes from the instrumentation (section 4.2.2). In particular, we excluded classes that were not covered (0% line coverage) by functional system logic tests. These classes are handling parsing errors: e.g., ParseException and TokenMgrError. We omitted them because they are not part of the core functionalities that are tested by system tests (e.g., computing a truth table). We acknowledge the GUI tests do exercise these classes (coverage greater than 0). This is however only due to the fact that we rely on GUITAR’s default input values, which are not Boolean expressions. Should the functional system logic tests also focus on parsing non-Boolean expressions, these classes would also be covered.
Figure 14. Value of diff_coverage System testing (I) - GUI functional system testing (E)

Figure 14 shows the difference of coverage between functional system logic testing in experiment I and GUI functional system testing in experiment E as a function (x-axis) of the different classes (i.e., coverage of functional system logic – coverage of GUI functional system testing). As Figure 14 shows, the values of difference are greater than or equal to zero indicating that functional system logic testing does better than GUI functional system testing with the exception of two outlier classes: classes 8 and 12, where GUI functional system testing does better.

The first outlier is branch coverage for class SetOfBooleanVariables in the parser (class number 8). We inspected the source code and found that the only difference in coverage between GUI functional system testing and functional system logic testing is one branch, which is covered in GUI functional system testing but not in functional system logic testing. The branch is triggered when the Boolean expression test input uses several times the same Boolean variable. This never happens in the functional system
logic test inputs, but it does happen in Boolean expressions we collected from the Internet.

The second outlier is for class VariableNegationControl (class number 12). By inspecting the code we found a method in this class that is covered by GUI functional system tests but not by functional system logic tests. The method breaks the ECB principle as it provides GUI functionality but is placed in a Control class: it implements a service offered by the Control class to present data in a specific format. Hence, the functional system logic test does not trigger this method.

To summarize, the second outlier is due to code that is misplaced and should not be counted when measuring structural coverage of functional system logic tests. The first outlier, meanwhile, would not take place if a Boolean expression with twice the same Boolean variable were used as test input and is more about parsing the input than using the functionality.

We simulated, by considering the lines and branches these methods contribute, the coverage one would obtain if (1) the ECB principle were adequately followed, i.e., the code missed by original functional system logic tests were not in a Control class but more adequately placed in a GUI class, and (2) at least one Boolean expression with at least twice the same Boolean variable were used in functional system logic test inputs. Table 14 shows the values of line coverage and branch coverage for experiment E, experiment I and the simulated improvement of experiment I (Simulated-I).

In experiment Simulated-I, line and branch coverage would reach 94% and 98.9%, respectively. Based on the values of Table 14, we conclude that values for line coverage and branch coverage of Control classes for functional system logic testing are
better than those obtained with GUI functional system testing, even when accounting for the two outliers.

**Table 14. Line and Branch coverage for Control classes**

<table>
<thead>
<tr>
<th>Class</th>
<th>Experiment E</th>
<th>Experiment I</th>
<th>Simulated-I</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Line</td>
<td>Branch</td>
<td>Line</td>
</tr>
<tr>
<td>DNFControl</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>TruthTableControl</td>
<td>0.96</td>
<td>0.84</td>
<td>1</td>
</tr>
<tr>
<td>VariableNegationControl</td>
<td>0.94</td>
<td>0.95</td>
<td>0.84</td>
</tr>
</tbody>
</table>

### 4.7 Validity threats

As with any experiment, our work is subject to threats to validity [197].

Threats to construct validity relate to our choice of measurement as a way to compare functional system logic testing and GUI functional system testing. Although many different criteria can be considered when comparing two different testing techniques, structural coverage is a well-known measurement for such an objective, and statement and branch coverage are two standard criteria that are extensively used. We also measure cost. The cost of testing a system can depend on the time and resources required for executing the tests [26]. Further, these factors are typically directly proportional to the size of the test suite: the greater the number of tests the more resources will be utilized. We therefore measure, similarly to many others before us in the context of unit testing [198] or GUI functional system testing [199], the cost of a test suite as the number of test cases of that test suite and execution time. In our experiment, the number of times the Compute button was clicked using GUI functional system tests was eight times, while the number of times functionalities are triggered by functional
system logic tests was 11. This can be considered as a threat to validity because GUI functional system tests require fewer calls to the software functionality.

Conclusion validity is about the relation between what we manipulate and what we observe. Threats to conclusion validity are concerned with issues that affect the ability to draw the correct conclusion about relations between the manipulation and the observation of an experiment. We tried to set up our experiments as systematically as possible, accounting for possible points of variation (e.g., input selection for GUI tests), in a manner which we believe was as fair as possible. GUITAR exhibits a stochastic behaviour: several executions can result in different test suites. Although several executions of GUITAR would be needed to obtain a more thorough comparison, we only executed GUITAR once. We believe, however, that results of other executions would be similar, which we have observed, because test cases depend highly on the characteristics of the GUI, which did not change: e.g., regardless of the execution of GUITAR, there would be many more tests exercising the variable negation functionality because of the larger number of widgets. Randomly, GUITAR would generate a similar number of tests that do not click on a Compute button. We therefore consider our GUITAR test suite representative and the threat to conclusion very low.

Internal validity is about the set-up of our experiments. Threats to internal validity are influences that can affect the independent variable (structural and branch coverage, in our experiment) with respect to causality, without our knowledge. We started from a functional system logic test suite that we did not create and which achieves a very decent level of coverage; we used the default GUITAR settings that have been shown by others
to work well [3]; and we systematically designed our experiments, prior to conducting them, to ensure a fair comparison.

One threat to internal validity could be that we do not provide a valid input to text fields that could appear at run time. Also, it might be better to use the empty string instead of using GUITAR default text in some experiments, as the variable negation interface of the SUT accepts the empty string as an input in one text field if the other text fields have valid inputs.

Threats to external validity limit our ability to generalize results. We acknowledge we used only one case study, one functional system logic test suite and one GUITAR GUI functional system test suite, which hurts external validity. To give our results a better chance of being generalizable, we qualitatively explain results so that they become less dependent on the case study and test suites.

4.8 Conclusion

The chapter investigated relationships between GUI functional system testing and functional system logic testing experimentally. We conducted an experiment whereby we studied the differences in terms of structural coverage of the application logic code between functional system logic tests and GUI functional system tests [200].

The experiments used GUITAR to perform GUI functional system testing for a GUI-based Software Under Test (SUT). Results show that coverage achieved by functional system logic testing is better than, though close to, that of GUI functional system testing. Moreover, our experiments show that GUI testing “à la” GUITAR requires much more effort, time and computation cost than functional system logic
testing. Although replications of our experiments are necessary to understand precisely the phenomenon we have encountered, our results empirically prove the existence of duplicate effort when using GUI functional system testing and functional system logic testing simultaneously. Our investigation suggests that the use of functional system logic testing on the application logic code would be a less costly verification technique of the application logic of the SUT than by means of GUI functional system testing.

The validity threats were evaluated. This is important to do upfront to ensure that the threats are minimized [197]. It is close to impossible to avoid all threats [197]. But all threats in our experiment were identified and mitigated whenever possible. Based on our evaluation, we were ready to run the experiment and are confident it is possible to repeat the experiment. For example, many of our GUI functional system testing experiments were repeated many times using two versions of GUITAR and with two types of integration with COBERTURA, with ANT [201] and with shell scripts. We believe our results are to some extent valid and generalizable.

To summarize, this chapter provided empirical results to the software testing research community and testing practitioners about a comparison between GUI functional system testing and functional system logic testing. GUI functional system testing of GUI-based software (e.g., using GUITAR) is an expensive activity in terms of effort and computational cost. The use of GUITAR on a real-life software is almost impossible. For example, we had to identify the coordinates of text fields in order to update the test cases with valid inputs. This would be impossible for large software with alternative GUIs and many widgets in each single GUI.
Chapter 5 GUI verification

In this chapter, we present our work for verification of the GUI of a GUI-based software. Section 5.1 presents the introduction and motivation. Section 5.2 explains the proposed solution (thesis) and gives an idea about which faults our proposed solution can find. Section 5.3 describes several multiplicities of input-output relation that can be realized by Boundary classes. Section 5.4 presents our implementation of the solution. Section 5.5 presents advantages and disadvantages of the proposed solution. Section 5.6 presents an experiment towards the validation of the implementation of our proposed solution. Sections 5.7 and 5.8 explain scalability and extension possibilities. Section 5.9 presents suggestions to implement the solution using Julia. Section 5.10 presents the conclusion.

5.1 Introduction and motivation

Our objective is to verify the GUI without testing. Our focus is on the Boundary classes of the ECB design principle (sections 2.2 and 2.3). We remind the reader that Entity classes represent the information that the software manipulates, Control classes realize the use cases, and Boundary classes realize the interactions between the software and the actors.

Boundary classes ideally transmit requests and data without changing the semantics of that data. However, they possibly change the syntax or the type of the information (section 2.3).
GUI functional system testing of the GUI-based software (e.g., using GUITAR) is an expensive activity regarding effort and computational cost. The use of GUITAR on a real-life software is almost impossible (Appendix I).

This chapter answers Research Question 3: Is it possible to verify a GUI-based software using static analysis of the GUI code combined with system tests on the application logic?

5.2 The proposed solution (thesis)

Our solution is to verify the GUI using static analysis. The objective is that by adding the functional system logic tests, we can verify the whole software. As Figure 15 shows, our solution is to **statically verify the GUI without testing it**. We grayed out parts of the software which are out of scope in Figure 15.

Figure 15. GUI verification
We define the term “input variable” as any variable in the GUI that receives a value from the user. We define the “argument variable” as any variable in the header of a method in a Control class. We use the term input-output relation to refer to the relation between these two kinds of variables.

Our solution is to study the input-output relation. Our objective is to make sure that the GUI code receives the input provided by the user to the GUI classes and passes it to the Control classes without semantic change. We use a static analysis technique to achieve this objective.

In our solution, we describe an argument variable as a function of input variables. In the general case, we have the argument variable as a function of a number of input variables. The simplest case is that the function is an equality which means that the number of input variables is equal to one. In other words, we have an equality between the argument variable and input variable.

A fault model assists practitioners when doing test case generation, and data or control flow analysis [202]. It also allows one to qualitatively assess fault detection of a specific verification technique [203]. We explain some faults that are expected to be discovered by our approach. At the same time, we show that other faults cannot be detected using our approach. The first type of faults in our fault model is a change in a value of an input variable when the code should not change the value. In other words, the GUI changes the value entered by the user. In the normal execution without error, the GUI should pass the value as is. In this regard, we distinguish between several types of variables, assuming that the GUI should pass the input as is to Control:
A. A fault in a numerical value, which happens if there is a difference between the value entered by the user and the value passed to the Control if the input is numerical;

B. A fault in a Boolean value, if the value entered by the user has changed from true to false or vice versa;

C. A fault in a string, which could happen if the user entered a text and then some characters are added or removed from the text.

We have similar handling for the different variable types A, B and C. This handling is simply performing an equality check. The equality check is done between the entered value by the user and the passed value to the Control. We think that by using a static technique we can detect all types of errors as described later in this chapter. We show this in our next experiments.

The software developer may implement the GUI in such a way that it modifies the format of the input before passing it to the Control. Such changing of the syntax is allowed when using ECB. For example, the user may split the value of a String variable, then pass the two values to two different methods in the Control. We can handle this case similarly to the previous one discussed above by representing the code as a mathematical function. The expected mathematical function represents the supposed output of the GUI in term of input. We compare the mathematical function that represents the SUT code with the expected mathematical function. In this case, a change of the format of the input is not necessarily a fault. The fault is that the change of the format of the input is not done according to the expected mathematical function. This type of faults is that the GUI changes the format of an input variable incorrectly. In other words, the user enters a value for which the GUI should change the syntax, typically through applying a mathematical
function to it. The second type of faults is not a broader description of the first type because here we have a mathematical function that describes the output as a function of the input. The function in which the output is equal to the input is not a mathematical function in our case.

The third type of faults is that the values of two input variables get swapped with each other. We consider this kind of faults as a combination of two faults belong to the second type. Hence, we handle it as we handle the first case.

We think our solution can detect all previous three fault types. However, our solution is constrained by abilities of technology to implement the solution. For example, Extended Static Checker (ESC) has a limitation with strings. ESC can do an equality check on strings. Unfortunately, it cannot understand complicated operations on strings.

The change of semantics of the user input is a violation of ECB principle as software developer should not add a code to change the semantics in the GUI. However, there might be violations by practitioners. However, our solution can adapt even if the GUI changes the semantics of the input. We can check whether this change is done according to a mathematical function on the input.

We focus on the input-output relation to avoid the effort of doing an integration testing by the software tester. If we test the GUI alone, then the logic alone, we need an integration testing to verify the whole software. By working on the verification of the input-output relation, we think that we do not need such an integration testing. The reason is that since static analysis assures the correctness of values passed to the Control, then there is no other source for giving wrong value to the Control (beside the GUI). In other words, since we use a precondition before any method in the Control that receives an
input, we put a static variable to hold each value entered by the user, we study the input-input relation, then, there is no other source of values to be used in the Control which is not checked. We observe the source and the destination of the input data. Since we checked all input, we think that all inputs to the Control are correct (assuming preconditions passed). If functional system logic tests pass, then the whole software is verified. Hence, the assumption is that assuming the verification of the GUI passes and that function system logic level tests pass on the application logic, then the entire software is verified. This is a claim that will need to be further studied in future work.

We study the input-output relation in one direction which is from the GUI to the Control. The same study can be applied on the other direction of the relationship which is from Control to the GUI.

5.3 Multiplicities of input-output relation

We study multiplicities of the input-output relation. We believe that understanding of this relation enables us to implement our solution. Then, it will help us to investigate whether our solution fits with most kinds of input-output relation. In other words, the rationale for discovering such multiplicities is to use them in the validation of our solution after we implement it. In the next section, we try our solution on different types of multiplicities in hope of showing that our solution is not affected by the type of multiplicity. Hence, we try to generalize our results.

We remind that an input variable is any variable in the GUI that receives a value from the user. We remind that an argument variable is any variable in the header of a
method in a Control class. The input-output relation refers to the relation between these two types of variables.

We study the multiplicity of the relation between these two types of variables. We give examples from books about these multiplicities. These books do not discuss our fault model. They also do not discuss the ECB principle. However, they discuss GUIs which in case the software developer implements them; we think the GUIs will have a code with input-output relations that have such multiplicities. There are different multiplicities of this relation:

- Many to One (N-1): Several input variables to Boundary classes form one argument variable to a function in a Control class. One example to this relation is “The MathWorks” which is of the Control Point Selection Tool (cpselect), which is part of the Image Processing Toolbox 3.0 [204] (page 27). The software analyzes the image as a different layers. The software takes a set of measurements from each layer (the detailed view, the base image, the warp image, the overview), then passes them to the Control. The inputs are the variables related to different layers. The output is a variable contains measurement. The design of code assigns the measurement value to an argument of a method in the Control. This design suggests that the GUI does the measurements thereby breaking the ECB principle.

- One to One (1-1): One input variable to a Boundary class becomes an argument variable to a Control method. An example for this is iTune software which shows a set of audio tracks on the GUI [205] (page 139). The user selects one of the audio tracks. The selected item name is passed to the Control upon a click for playing the track. The software plays the selected sound item to the user.
• Many to Many (N-M): Many input variables to Boundary classes contribute to many argument variables. The multiplicity includes many arguments of one method or many arguments of several methods. An example of this multiplicity is the Record Macro window in MS Excel software [205] (page 159). This GUI (window) has several text fields to receive an input from the user. For example, the macro name, shortcut key, store macro in, and description. The code passes several inputs to several arguments in several methods in the Control. The Macro requires that several methods will be called in the Control concurrently. Not necessarily all of these methods are responsible for the Macro itself. For example, there will be one method to generate the Visual Basic code that is equivalent to the Macro. For each Macro, there is a Visual Basic code that does the same task of that Macro. We do not know this for a fact, but we think the Excel software uses a design principle as ECB or a similar one. This method of code generation is not related to the recording of the macro. The user can edit the code which is generated by that method. Hence, the macro recording is done by one method while generating the code is done by another one. However, both methods receive the same inputs. These inputs are related the user actions on the GUI.

• Many or One to zero (N..1-0): Many or one variables do not contribute to any argument variable to the Control class. An example of this relation is an option to change the color of a GUI. The software developer implements such a command through a couple of lines of code in the GUI code itself.

• One to Many (1-M): In this kind of relation, there is one input variable to the GUI that contributes to many arguments of methods in Control classes. An example of this
relation is an Excel sheet that works as an input for the GUI. Then, a data processing software passes the sheet to several sorting algorithms (increasing/decreasing order) in the Control. Each of those algorithms shows the data chart sorted differently [205] (page 168).

- Zero to one or Many (0-1..M): In this kind of relation, there is no input variable to the GUI but one or many arguments to methods. This multiplicity could happen when the input in one GUI depends on retrieving information related to a Control/Entity class that has its own GUI. For instance, showing the serial number of an employee in the GUI of a finance department requires calling functionality related to the human resources department. Usually, the software shows the serial number in disabled (non-editable) text field in a GUI related to the finance department. A user in the finance department does not enter the serial number. Later, the user can use the serial number as an input to some functionality related to finance even though it was not an input to the finance GUI. In other words, the software shows the serial number in the GUI of finance but does not allow the user to enter it there. We study the relation in one direction from the GUI to the Control. The study for the other direction from the Control to the GUI is not done now. We believe that the same study can be applied on the relation from the Control to the GUI.

5.4 The implementation of our proposed solution using Java Modeling Language

This section describes our implementation of our solution in Java. We depend on our definitions made in section 5.2 of input variable, argument variable and input-output
relationship. Subsection 5.4.1 explains the implementation steps that need to be followed by the software tester in order to be able to verify the GUI of the SUT. Subsection 5.4.2 describes one possible way of implementing the solution when the argument variable exists in the header of the constructor of a Control class. Subsection 5.4.3 describes another possible implementation in exceptional cases. Subsection 5.4.4 presents some of our observations when studying the input-output relation. Subsection 5.4.5 gives a simple illustrative example.

5.4.1 Implementation steps

The implementation of our solution depends on the design of the interface between the GUI and the Control. In other words, it depends on the way that the communication is done between the Control and the GUI. Specifically, it depends on how the data is passed between the GUI and the Control. However, the principle that we formalized about having an “input variable”, an “argument variable” and input-output relation between them (At the beginning of section 5.2), reduces the dependency of our solution on the data exchange mechanism between the GUI and Control classes. In fact, the tester can completely discard the way data is exchanged and focus on the three main terms mentioned above. The tester needs to follow four basic steps to implement the solution:

1. Find the input variables which are part of input-output relationship.

2. For each input variable, add a static verification variable and assign to it the value of the input variable.

3. Find the argument variables that are part of input-output relation.
4. For each argument variable, add a precondition to the method that has that argument variables in its own header. This should be done regardless of the type of the method (whether it is a constructor, a static method, etc).

The step # 4, which consists in writing preconditions, needs more effort than the other steps. The tester needs to understand the relation between the input variable(s) and the argument variable(s). The precondition may represent that one argument variable is a function of a single input variable. It may also represent that one argument variable is a function of several input variables. We may have several preconditions that represent several functions of the same input variable. Each one of those preconditions is dedicated to a different argument variable.

5.4.2 Argument variable exists in the header of the class constructor

We use our results of the discussion on multiplicities to implement a solution using Java Modeling Language (JML) and Extended Static Checker (ESC). We use the information of multiplicities to realize the core idea with ESC.

We present a way through which we can study the input-output relation by checking Java Modeling Language (JML) preconditions that the user adds. Hence, the user can verify the GUI by checking whether those preconditions passed.

Let us assume that the GUI is supposed to present a numerical input as is to a Control classes. Using a JML precondition, with a static checker, we can check if there is a difference in value between the input variable to the GUI and the argument variable passed to the Control class. This way, a static checker can be used for fault detection in the GUI. In case the static checker reports that there is a violation of the precondition due
to a difference between the input and the output of the GUI, then, there is a fault in the source code. If the static checker does not report any violation (difference), then there is no fault in the source code.

We add a static variable in the GUI class to hold the value entered by the user. We access the value of that static variable when writing our JML precondition. We want to access it later from inside the Control classes without making an instance of the GUI class but using the name of the GUI class. The Control class has no direct access to a variable defined inside the GUI class due to the encapsulation principle of Object Oriented programming. This principle applies regardless of the Java visibility of that variable (private, public, protected and default). One exception is a static variable which is not affected by this rule of encapsulation in Object Oriented though a static variable is affected by the modifier. We use this exception for verification purposes and hence we do not create the instance.

In the next step, we add a precondition. The location of the precondition depends on the design of the Control class. In other words, which methods from the Control class has an argument variable. We add a precondition to each method that has an argument variable. We do not discuss the special case of static methods in this section.

If the method in the Control class has an argument variable in its header, we add the precondition to that method of the Control class. The precondition may compare the value of the static variable added in the GUI with the value of the argument variable to the constructor. There may be more than one argument variable and, in this case, the tester should implement preconditions for all of them. The precondition may check that the argument variable is a function of one input variable.
In general, when the GUI calls the Control, it first creates an instance of a Control class. Hence, the GUI calls the constructor of the Control class first. This sequence of calls from the GUI to the Control is the reason that we add the precondition to the constructor. When we add the precondition to the constructor, we assure that the static checker checks the precondition before calling any code of the Control. In the execution path, this exactly happens at the intersection between the GUI and Control inside the GUI code. In other words, we want to check the precondition before any call to the Control code. Checking the precondition helps us to study the input-output relation. Later, we explain more about this in examples. The precondition applies a mathematical function on the value stored in the static variable. The precondition compares the output value of the mathematical function with the value of the argument of the constructor. The precondition passes if the two values are equal. Otherwise, the precondition fails.

The tester can use a simple search tool in any Java development environment to find the place where to define the precondition. For example, in Eclipse, the tester can open a GUI class in the editor. Then, the tester can use the search/replace option to search for the class name of the Control class. This will find the uses of the Control class inside that GUI class. Eclipse will highlight all lines of code which contain the name of that Control class. One of those highlighted lines can be a call to the constructor of the Control class. In complex scenarios, the software tester may use a tool like Atlas [185]. Atlas gives an option to right-click on any variable and then tracks calls to build a graph that shows how the variable is being used/called in the software.
5.4.3 Exceptional cases

We can add a precondition before a Control method that is different from the constructor as long as the method has an argument variable in its own header. There are some exceptions in which there is no call for the constructor such as static methods which can be called without creating instance of the Control class. Only few programmers use a static method because it is not recommended practice. It is actually breaks the concept of Object Oriented. The reason is that the code becomes as a product of structured programming without objects. Static modifier was designed to be used only for special and mandatory cases.

The constructor of a Control class may not receive any data from the GUI to the Control in some cases. In those cases, the data that should be forwarded from the GUI to the Control is passed through other methods in the Control class. In such cases, we add a precondition to those methods in the Control which receive the data. We do not need to add a precondition to the constructor in such case.

In some other cases, the constructor of a Control class receives data from a GUI class but the Control class does not handle that data inside the constructor body. The constructor instead passes the data to other method inside the same Control class. In this case, we add our precondition before that method. Another possibility is to create a fake method with a precondition and make a call to that method. The call to the fake method should be inside the method that receives the data from the constructor.

We may need verify the input-output relation at a specific point in the sequence of calls from the GUI to the Control. For example, we may need to check the value of the
static verification variable at a specific line of code inside a method in a Control class. In this case, we may use a fake method that does not have any functionality. We add a fake method to the Control class with a precondition, then add a call to that method before that specific line of code. For example, we add a call to a fake method in the middle of a function in a Control class to check a precondition at that line of code.

Some software developer may allow the underlying functionality to create instance of a GUI class. This is may be done especially in maintenance phase where developer don't understand code written by other developers. It may rarely happen by the original developer who wrote the code. However, the underlying functionally should not create an instance of a GUI class. Even if the underlying functionality needs more data from the user while doing some operation, there should be an event clicked on the current GUI by the user to create an instance of a new GUI which will receive that data. The new created GUI class either uses the already created instances of the underlying functionality or creates new instances (or both). In fact, if a business logic class contains a statement:

```java
import java.awt.*; or import java.swing.*; this is an indication of lack of high quality in the code. Even if an exception happens at level of underlying functionally and needs to create a GUI shows the exception message, it should throw the exception to the higher level which is the GUI level. Then, a new GUI with an error message is created. Anyway, while some practices may contradict software engineering principles about the multi-tier software application, we do not see any problem in applying our verification approach even with such hard to maintain code.
5.4.4 Technical observations on the implementation of the input-output relationship

Our use of preconditions with static checker verifies lines of code that implement the input-output relation. Those lines of code first usually receive the value entered by the user in widgets such as text fields. The code then may change the type of that value to other types. In fact, the code normally changes the syntax of that value but not its semantics. Then, the code passes that value to the Control class. All three types of code (the one which receives a value, the one which processes it and the one which passes it) is all together an implementation of an input-output relation. The developer of the GUI may change the semantics in violation of ECB. The change of semantics is a design error, but it does not affect our implementation of our proposed solution. If this is the case, we need to make sure that this change is done correctly as the developer wanted it to be. In this specific example, we want to make sure that the input received from the GUI is the same as the input provided to the Control class after applying a mathematical function on that input.

We do not study oracle here because oracle is a notion of testing while we are doing verification for the GUI. Instead of the oracle concept, we describe how to know whether there is a fault in the software. Simply we depend on Extended Static Checker (ESC) to verify the GUI and inspect whether there is any violation of the preconditions. If there is a violation, it means there is a fault in the code.
However, there are limitations of ESC as it cannot handle complicated operations on string variables. For example, ESC cannot evaluate operations such as to concatenate strings or sub-string a string.

5.4.5 A simple, illustrative example of a fault and the solution

This subsection presents a simple illustrative example. We focus on only one direction of the input-output relation which is the one from the GUI to the Control.

Figure 16 shows the class diagram for a simple software which we use to explain how we utilize Extended Static Checker (ESC). The software that we designed using the ECB design principle. The Entity class is MyTextEntity, which has a functionality for creating some text files on the hard disk and printing text on the standard output stream. TextControl is a Control class that receives inputs from the Boundary class and makes a call to the Entity class. RadioComponent, CopyTextComponent, and MainExeFrame are Boundary classes which receive the input from the user, process it and pass it to the Control class.
Figure 17 shows the main window of the software when the user enters an input, using the text field, and presses the copy button. The software takes the user input, adds the value 1 to it, and then shows it in the label under the radio buttons. The output is 2 in this case. In this specific example, the CopyTextComponent instance receives the input value "1" from the MainExeFrame instance. The class CopyTextComponent adds a value of "1" to it, and then it passes it to the Control class. This code intentionally introduces a fault. In the normal execution, this software should show the input on the output without any semantic change. The code which increases the input value breaks the ECB principle; the Control class should do this.
FIGURE 17. THE MAIN WINDOW OF SIMPLE SOFTWARE AFTER TYPING THE INPUT AND CLICKING THE COPY BUTTON

Table 15 shows a sample of the code from class CopyTextComponent. The CopyTextComponent instance obtains a value from the GUI (line 1) and stores it in the userInput variable. Then (line 2), it declares a variable called callArgument. The callArgument variable takes the value of userInput and adds the value 1 to it. Then it passes it to the Control class instance.

| 1 | int userInput=Integer.parseInt(inputExpression.getText()); // get the user input from the text field |
| 2 | int callArgument = userInput + 1; // process the entered value by simply adding 1 to it. |
| 3 | TextControl t = new TextControl(Integer.toString(callArgument)); // passing the value to a constructor of class TextControl in the Control package |
| 4 | t.printCopyMessage(evt.toString(),0); |

Table 15. The part of source code of GUI that changes the input value

Figure 18 shows a diagram to illustrate the input-output relation of the use case of clicking the copy button. The use case includes the scenario which starts by having the actor communicate to the software. Then the user enters a value in the text field and presses the copy button (Figure 17). The click by the user triggers the method actionPerformed, which creates an instance of the TextControl class. Then, the call proceeds to the method printCopyMessage(). The method printCopyMessage() triggers the log() method in the Entity class called MyTextEntity.
Based on Table 15 and Figure 18, we can notice that the value of the input variable provided to the GUI is passed to the constructor of the Control class when the actor calls the `actionPreformed` method. Hence, we need to make sure that the input given to the `TextControl` instance is the right input—in this case, the string value received from the text field. The code converts the string to an integer and increases the output by a value of one. The code converts the value (after the increment) to a string. After that, the code passes the string to the Control class instance.
One procedure would be to add a static variable for every single input provided by the user. Table 16 shows the modified `CopyTextComponent`. To identify this location in the code, user can search for the lines of code in the GUI class which make a call to the Control class. The GUI code which makes a call to the Control includes at least the line of code which uses the constructor of a Control class to create an instance. It may include any other lines which make a call to the Control. We define a variable called `verification_variable`. The line of definitions is not shown here because we define the verification variable as public static data-member. The variable is used to store the value entered by the user. We define it as a data-member because we want to access it from any point inside the class to assign a value to it. We define it as a static variable because we want to access it from any class inside the software. Static verification variable helps us to know the value of the variable at any time at the runtime of the software regardless of any instances in the memory. The same feasibility of static variable applies at any time in the static analysis of software by ESC.

We also add a local variable `output_variable` (line 5) to contain the data that the code passes it to the Control class. Hence, we avoid passing the data to the constructor directly. There is no local variable in the original code to contain the data which the GUI passes to the constructor of the Control. Adding this local variable helps ESC to analyze the code. The line is added in a specific part of code. It is the part of intersection of the GUI classes and Control classes.
The next step in our verification process is to add a precondition to the constructor of TextControl as in Table 17.

**Table 17. The adding of a precondition in the source code**

```java
/* process equality check on the entered value and value that reach the Control.
   @ requires s == CopyTextComponent.verificaton_variable */
4 TextControl(String s){

Then, ESC, when analyzing the modified code, generates an output file as in Table 18. There is a violation of the precondition as shown. Specifically, Table 18 shows a part of the output file that is generated by ESC. We obtain the output file in the case of using valid precondition (The GUI should pass the input to the Control as is). We conclude that we verified the GUI for this particular scenario. In other words, there is a fault for the variable callArgument. ESC indicates that a precondition is possibly not established. This means for us that there is a fault.

**Table 18. Part of the content of output file generated by ESC**

```
CopyTextComponent.java:109: Warning: Precondition possibly not established (Pre)
   TextControl t = new TextControl(s);
^
Associated declaration is "./CopyTextComponent.java", line 16, col 6:
   @ requires s == CopyTextComponent.verificaton_variable;
^
Execution trace information:
   Executed then branch in "CopyTextComponent.java", line 93, col 55.
```

5.5 Advantages and disadvantages of our solution

We clarify two main advantages for our solution: saving of testing time and effort.
As for the disadvantages, it may not be easy for a tester to formulate a precondition. The
user needs to understand the code. We are unsure that the GUI code of industrial software is error-prone to the kind of faults our solution can detect. In order to obtain an evidence of having the software error-prone to such faults, a comprehensive study on several industrial case studies need to be done. ESC has some issues with output streams such as System.out.println(). We commented on those lines when doing the experiments. ESC may not be able to understand all the code.

5.6 An experiment

We design an experiment through which we validate our implementation of our proposed solution. We follow two steps in our experiment. The first one is to validate the applicability of our proposed solution (section 5.6.1). The second one is to validate the solution in term of fault detection (section 5.6.2). Then, we discuss threats to validity (section 5.6.3).

5.6.1 The validation of applicability of the proposed solution

In this section, we validate the applicability of our solution on several types of multiplicity. It is hard to find one case study that has many types of multiplicities. At the same time we have to show that our solutions can work regardless of the type of multiplicity. To achieve our objective, we implement a synthetic example to be used as a case study. The GUI of this synthetic example simulates three types of multiplicities. We think that those multiplicities are the most common ones. If our solution works well on this synthetic example, we conclude that it will work on any software that has one or all of these three kinds of multiplicities. Hence, we argue that our approach will work on most of the possible scenarios. Consequently, we hope that such validation shows the
capability to generalize our results. Figure 19 shows the main GUI of a dummy case study we use to demonstrate how our verification technique is applicable on several multiplicities of input-output relation. We use faulty preconditions instead of generating faults in the software (i.e., simulation as generating fault cannot be done automatically in our case).

![Main Exe Frame](image)

**Figure 19. Dummy Case Study**

There are three buttons in this case study, and each of them shows one possible relation between the input to the GUI and the output to the logic. The first button simulates the first type of multiplicities which is 1-1. Clicking on that button shows another window. In the new window, the user can enter a text and click a button as shown in Figure 20.

![1 to 1 relation Exe Frame](image)

**Figure 20. Window of 1-1 relation with string input**

**Table 19. JML code to verify the 1-1 relation**

```java
13 //Constructor - Takes the logic expression as input
14 /*@ public normal_behavior
15 @ requires expr == OneToOneComponent.verification_user_input;
16 @*/
17 public OneToOneControl(String expr) throws ParseException{
```
When the user enters "String Input" and clicks the button, the GUI takes the input entered by the user in the text field and passes it to the Control. Figure 20 shows a label that has the text "String passed as is to control" at the bottom of the window. It means that the text entered to the GUI is passed to Control without change.

To verify the 1-1 relation, we add Java Modeling Language (JML) code at the Control class as shown in Table 19.

In the right part of the precondition, we have a variable \(\text{OneToOneComponent.verification_user_input;}\) which is the name of the GUI class followed by the name of static variable added for verification purposes to the GUI class.

For the previous precondition, the ESC does not complain about ESC. However, if we change the precondition to have the operator to be the inequality \(!=\) instead of the equality \(==\), then, the ESC warns about violation of precondition. We use this fault in the precondition to simulate a fault in the code. Hence, the ESC shows that it is capable of evaluating the code for this type of fault. In this scenario, we simulate the possibility of the code having a fault.

Figure 21 shows an example of 1-N relation.
Figure 21. Window of 1-N relation

The JML Code to verify this 1-N relation is shown in Table 20. The synthetic software simply receives an integer value as an input. Then, based on the value of that integer, the software creates an array of elements. The values of those elements and their indexes appear on the output. The GUI shows the output. However, this output is in fact the output of the GUI to the Control.

Table 20. JML code to verify the 1-N relation

```java
/**
 * public normal_behavior
 * @requires a + b + c == 3 + (3* OneToManyComponent.verification_user_input);
 */
public OneToManyControl (int a, int b, int c) throws ParseException{
...
```

Figure 22 shows an example to N-1 relation. In this synthetic example, the software receives several values as inputs. Then, it outputs the sum of those values. Hence, we have several inputs and one output. The software shows the sum as an output, but it is the output of the GUI to the Control.
Table 21 shows JML verification code for N-1 relation. The verification code shows that the output to the Control should be the sum of inputs to the GUI.

### Table 21. JML code to verify the N-1 relation

```java
/*@ public normal_behavior
@ requires a == ManyToOneComponent.verification_user_input[0] +
    ManyToOneComponent.verification_user_input[1] +
    ManyToOneComponent.verification_user_input[2];
@*/

public ManyToOneControl(int a) throws ParseException{

  // Validation of fault detection

  We use the same experimental unit that we used before in measuring structural coverage. The software performs some computations on Boolean expressions provided by the user through the GUI. The SUT provides a variety of functionalities discussed
before, i.e., computing a truth table, computing a DNF of a Boolean expression, and computing related to the variable negation strategy for a Boolean expression. Section 4.1 describes the experimental unit.

We focus on two packages as we want to verify the GUI: The package named gui contains all the Boundary (GUI) classes, and the package named control.

One possible way to study the efficiency of an approach in fault detection is to use mutants. In that way, mutants are generated using mutation tool. When we used MuJava mutation tool [206] to generate faults for truth table functionality, more than 90% of the faults were related to the non-functional aspect of widgets such as the size of text field. Finding such faults is out of the scope of this thesis because they do not belong to our fault model in which we focus on the functional aspect of the software.

We cannot use fault detection experiment when the GUI process the user input if the input is string. Many of the faults in the string processing cannot be detected by ESC. For example, if we have a text field tf1, one mutant generated by MuJava is to replace the `getText()` with the `getName()`. The issue is that ESC has limitations in evaluating operations of sting handling. Hence, it may be unable to discover potential faults in the string processing of input variables.

Instead of a faulty software, we use faulty preconditions. We formulate three preconditions for our case study. For each of these preconditions, we run ESC twice, one time using a valid precondition and another one using incorrectly modified (invalid) precondition. We do this to show that ESC can evaluate the code. Hence, we simulate that ESC report in case there are problems in the code such as existence of fault. We check the output of ESC; ESC should not complain about the valid precondition and
complain about incorrect precondition, and we consider this as an indication of the ability of ESC to detect faults. We argue that ESC will be successful in evaluating both valid and invalid preconditions. Hence, the answer for our research question in Chapter 1: Research Question 3. Is it possible to verify a GUI-based software using static analysis of the GUI code combined with system tests on the application logic? is yes. In other words, yes, static analysis can be used to verify the GUI through studying the input-output relation without testing it. Otherwise, the answer is no.

In our case study, there is one argument variable to the truth table Control class, one to the DNF Control class and three or more to the variable negation Control class.

In our proposed solution, we make sure that the GUI handles the data received from the user correctly. In our case study, correctly means that the output of the GUI to the Control is the same as the user input. This is due to the fact that our case study strictly follows the ECB design principle. In our case study the GUI does not change the semantics of the input. We have three GUIs that need to be studied: Truth table GUI, DNF component GUI and variable negation GUI.

There is no need to study the main window because it has three buttons only. The user input is mainly clicks, but no values to the Control. The main GUI is just an interface that has three buttons, and each of those buttons shows one of these three windows when the user clicks the button. We may need to check the code which the software triggers when the user clicks on the button. In other words, we may need to consider and check a click as an input from the user. However, we are more interested in input variables that receive values (such as text, number, or Boolean value) from the user. A click on a button without other input could lead to a call to the Control. Another
sequence of events for the software could be: The main window has three buttons when one clicks on the truth table button, then, there is a call to a Control class that creates the new window to get the data for the truth table functionality. Also, there are buttons to change the format of the input. We can consider this change of format as an input to the software. However, this input does not reach the Control. Hence, the code that implements this change has one input variable. The Control code does not have an argument variable to receive that input. If we formalize this input-output relation, the relation is of a N-0 type of multiplicity. In other words, the GUI does not have any output to Control.

For the truth table window, there is only one text input that is the Boolean expression. When the user clicks the compute button, the software takes the text from a text field and passes it to the Control.

We are also interested in the code that the software executes when the user interacts with the GUI. For example, the class JButton from the swing package has its listener class.

We are interested in verification of the listeners attached to the widgets but not exactly the whole listener code as part of it sometimes is generated by tools. The listeners work when the user triggers an event on the GUI. First, we study the code inside the actionPerformed method that is called by those listeners. The GUI code calls these listeners at the last step before calling the Control code. We next show the source code of this software method (actionPerformed) with a highlighted line of code. This line of code is at the intersection between Boundary and Control classes. In other words, the
intersection is the line of the code of the software in which the GUI classes and Control classes interact with each other.

This line is related to the input-output relation because it takes a value from the GUI and passes it to the Control. We assign values to the static variables before this line.

```java
public void actionPerformed(ActionEvent evt) {
    int i;
    if (evt.getActionCommand().equals(computeCmd)) {
        /*
         * The following two line of code were added in order to track execution
         * August -4- 2015
         */
        if (outputPanel.getComponentCount()!=0)
            //we remove the old truth table and create a new one
            outputPanel.removeAll();
        try {
            TruthTableControl t = new TruthTableControl(inputExpression.getText());
            JTable resultTable = new JTable(t.getFormattedTruthTable(currentOutputOption), t.columnHeaders());
            resultTable.setPreferredScrollableViewportSize(new Dimension(100,300));
            outputPanel.add(new JLabel("Truth table"));
            outputPanel.add(new JScrollPane(resultTable));
            parentFrame.pack();
            parentFrame.validate();
            computeButton.setEnabled(false); //disable the compute button
        } catch (Exception e) {
            computeButton.setEnabled(false);
            JOptionPane.showMessageDialog(parentFrame, e.getMessage(), "Parsing Error", JOptionPane.ERROR_MESSAGE);
        }
    } else { // one of the radio buttons
        for(i=0; i<outputOptions.length; i++)
            if (evt.getActionCommand().equals(outputOptions[i]))
                break;
        currentOutputOption = i;
        checkComputeButtonEnable();
    }
}
```

This kind of relation is a 1-1 relation. There is a need to save the output of the GUI before the highlighted line. The `inputExpression` is the text field which contains the text. The method `getText()` obtains the text from the input text field. In this specific example, one can notice that there is no a separate variable defined in the code to contain the text obtained from the GUI. Instead, the code passes that text directly to the Control class using the statement: `inputExpression.getText()`. ESC finds whether there is a difference between the value of the input variable and the value passed to the
Control. In this example, there is no intermediate variable between the GUI and Control. In other words, the value is taken from the GUI and passed directly to the Control. In this particular instance, we could argue that our solution is not needed. However, we mention that we do it anyway for illustration purposes.

To solve the problem of the lack of local variables in the GUI, we suggest the use of local variable defined inside the constructor of the TruthTableControl. ESC builds a syntax tree for the program. If we define local variable, we help ESC to understand the code better. We add the following verification code:

```java
String s = inputExpression.getText();
verification_user_input = s;
TruthTableControl t = new TruthTableControl(s);
```

Here is the code of the constructor of the Control class that receives input:

```java
//Constructor - Takes the logic expression as input
public TruthTableControl(String expr) throws ParseException, BinaryExpressionSolverException{
    logicalExpression = expr;
    parser = new LogicalExpressionParser(new StringReader(logicalExpression));
    parser.Logical_Expression();
    truthTable = new int[getVariantsNumber()][1+getVariableNumber()];
    variable_Names = new String[1+getVariableNumber()];
    buildTruthTable();
}
```

This example seems a simple one because in the original code, there is no way the string that software gets from the text field can be different from that passed as an argument. Since our approach is to study input-output relations, we need to ensure that the value returned by that expression is the same value passed to the Control class. We can notice that the received string has been assigned to the variable logicalExpression. We add the following JML code:

```java
//Constructor - Takes the logic expression as input
/*@ public normal_behavior
@ requires ((expr == TruthTableComponent.verification_user_input) ||
(expr == DNFComponent.verification_user_input) ||
(expr == VariableNegationControl.verification_user_input));
@*/
public TruthTableControl(String expr) throws ParseException, BinaryExpressionSolverException{```
TruthTableControl is being called in all three GUIs: Truth table, DNFComponent, and variable negation. Hence, the precondition of the constructor of TruthTableControl includes the verification variables from three different GUI classes.

For the DNF component functionality, the situation is similar as we need to handle only one input in the GUI of the DNF. When the user presses the Compute button, the text is taken from the text field and passed to the DNFComponentControl. Here is the code of the actionPerformed method:

```java
public void actionPerformed(ActionEvent evt) {
    if (evt.getActionCommand().equals(computeCmd)) {
        /*
         * The following two lines of code were added to track execution
         * August-4-2015
         */
        try {
            outputExpression.setText(""");
            DNFControl d = new DNFControl(inputExpression.getText());
            for (int y = 0; y < d.size(); y++){
                outputExpression.append("Term " +(1+y)+": " + d.get(y) + "\n");
            }
            parentFrame.pack();
            parentFrame.validate();
            computeButton.setEnabled(false); // disable the compute button
        } catch (Exception e) {
            JOptionPane.showMessageDialog(parentFrame, e.getMessage(), "Parsing Error", JOptionPane.ERROR_MESSAGE);
        }
    } else {
        System.err.println("ActionEvent unknown in DNFComponent: "+evt);
    }
}
```

One suggestion for the location of analysis is to add a local variable before the highlighted code. Here we have the following:

```java
String s = inputExpression.getText();
verification_user_input = s;
DNFControl d = new DNFControl(s);
```
As for the Control class, the JML code will be:

```java
//Constructor - Takes the logic expression as input
/*@ public normal_behavior
@ requires expr == DNFComponent.verification_user_input;
@*/
public DNFControl (String expr)
    throws ParseException, BinaryExpressionSolverException{

As for the variable negation component, the situation is different as there are three inputs by default from the GUI. The user may add more inputs as well. The input-output relation in this GUI is an example of N-1 relation. In fact, there is an array of inputs to be passed to the Control class. The array is still considered to be one argument to a method in the Control class. The values of this array come from different inputs from the user of the GUI. Here is the code of the actionPerformed method that we need to verify:

```java
public void actionPerformed(ActionEvent evt) {
    int p, k;
    if (evt.getActionCommand().equals(addProductTermButtonString)) {
        JTextField jtf = createTextField();
        productTermsPanel.add(jtf);
        listOfJTextField.add(jtf);
        parentFrame.pack();
        computeButton.setEnabled(false);
        showTestCaseButton.setEnabled(false);
        return;
    }
    if (evt.getActionCommand().equals(computeButtonString)) {
        JTextField jtf;
        Vector productTerms = new Vector();
        Enumeration enu = listOfJTextField.elements();
        while (enu.hasMoreElements()) {
            jtf = (JTextField) enu.nextElement();
            if (jtf.getText().length()==0)
                continue;
            productTerms.add(jtf.getText());
        }
        try {
            control = new VariableNegationControl(productTerms.elements());
            resultArea.setText(control.getRawResults());
        } catch (Exception e) {
            JOptionPane.showMessageDialog(parentFrame, e.getMessage(), "Parsing Error", JOptionPane.ERROR_MESSAGE);
        }
        computeButton.setEnabled(false);
        showTestCaseButton.setEnabled(true);
        return;
    }
    if (evt.getActionCommand().equals(showTestCaseButtonString)) {
        VariableNegationResultsTable vnTable = new VariableNegationResultsTable(control,"Variable Negation Results: Table"); // changes by b 15/09
        vnTable.setVisible(true);
        vnTable.requestFocus();
    }
}
```
Here is the line that we need to verify:

```java
control = new VariableNegationControl(productTerms.elements());
```

The `productTerms` is an array of Boolean expressions obtained from the variable negation window. Each element in the array has one Boolean expression from a text field. We need to define some verification variables to verify the highlighted line. We should define those static variables at specific points of the code. Hence, the ESC theorem prover analyzes those lines of code. We need to verify that each input entered by the user is a value in that array, so we change the code in the following way:

```java
public void actionPerformed(ActionEvent evt) {
    int p, k;
    if (evt.getActionCommand().equals(addProductTermButtonString)) {
        JTextField jtf = createTextField();
        productTermsPanel.add(jtf);
        listOfJTextField.add(jtf);
        parentFrame.pack();
        computeButton.setEnabled(false);
        showTestCaseButton.setEnabled(false);
        return;
    }
    if (evt.getActionCommand().equals(computeButtonString)) {
        JTextField jtf;
        Vector productTerms = new Vector();
        String[] productTermsStr = new String[listOfJTextField.size()];
        String verification_user_input[] = new String[listOfJTextField.size()];
        Enumeration enu = listOfJTextField.elements();
        int it=0;
        String userVal="";
        while(enu.hasMoreElements()) {
            jtf = (JTextField)enu.nextElement();
            if (jtf.getText().length()==0)
                continue;
            userVal=jtf.getText();
            productTerms.add(userVal);
            productTermsStr[it]=userVal;
            verification_user_input[it] = userVal;
            it++;
        }
        try {
            //control = new VariableNegationControl(productTerms.elements());
            control = new VariableNegationControl(productTermsStr);
            resultArea.setText(control.getRawResults());
        } catch (Exception e) {
            JOptionPane.showMessageDialog(parentFrame, e.getMessage(), "Parsing Error", JOptionPane.ERROR_MESSAGE);
        }
```

148
computeButton.setEnabled(false);
showTestCaseButton.setEnabled(true);
return;
}
if (evt.getActionCommand().equals(showTestCaseButtonString)) {
VariableNegationResultsTable vnTable = new VariableNegationResultsTable(control,"Variable Negation Results: Table"); // changes by b 15/09
vnTable.setVisible(true);
vnTable.requestFocus();
computeButton.setEnabled(false);
showTestCaseButton.setEnabled(false);
return;
}
System.err.println("Unknown ActionEvent in VariableNegationComponent: ");
System.err.println(evt);
}

We define two arrays in the highlighted code. The first array is to store the verification variables. We pass the second array to the Control class. The original code passes an iterator to the Control class. Iterator is not a primitive type and hence dealing with it in JML need some tricks. In the precondition of the Control class, we have to inspect every item inside the iterator as a verification procedure. We replace the iterator with an array.

The constructor of DNFComponent call three other methods:

```java
public VariableNegationControl(Enumeration DNFTerms) throws ParseException, BinaryExpressionSolverException{
    computeCubes(DNFTerms);
    computeUniqueTruePoints();
    computeNearFalsePoints();
}
```

The instructor does not use the input but pass it to another method. Hence, we do not write a precondition to the constructor.

We added a fake method called checkMap(). The precondition of this method checks the value when we want to use the code.

```java
/*@ public normal_behavior
@ requires term == VariableNegationComponent.verification_user_input[index];
@*/
public void checkMap(int index, String term )
{
}
```
We added a call inside the method `computeCubes()` to the method `checkMap()`. The method `checkMap()` has the verification precondition.

```java
private void computeCubes(Enumeration DNFTerms) throws ParseException, 
BinaryExpressionSolverException{
    String term;
    TruthTableControl ttc;
    ArrayList allTtc;
    allTtc = new ArrayList();
    allVariableNames = new ArrayList();
    int verification_i=0;
    while(DNFTerms.hasMoreElements()) {
        term = (String)DNFTerms.nextElement();
        // verification code
        this.checkMap(verification_i,term);
        verification_i++;
        ttc = new TruthTableControl(term);
        allTtc.add(ttc);
        allVariableNames.addAll(Arrays.asList(ttc.variableNames()));
    }
    //System.out.println(allVarNames);
    //Remove duplicates in allVarNames
    for(int i=allVariableNames.size()-1; i>=0; i--) {
        if (allVariableNames.indexOf(allVariableNames.get(i))<i)
            allVariableNames.remove(i);
    }
    //System.out.println(allVarNames);
    allCubes = new ArrayList();
    for(int i=0; i<allTtc.size(); i++) {
        ttc = (TruthTableControl)allTtc.get(i);
        allCubes.add(new Cube(ttc, allVariableNames));
    }
}
```

The use of fake method shown above is necessary when GUI passes the data to the Control through instance methods. In other words, the GUI does not pass data to the constructor of the Control classes. We repeat the same procedure on a correct and incorrect expression of our preconditions so that we obtain a set of output files. These output files are used to determine whether the precondition is violated.

ESC did not complain about all three valid preconditions and complained about all three invalid (incorrect) preconditions. Hence, we conclude that ESC is capable of fault detection in our context.
ESC takes around 2 to 3 minutes to analyze the code. The time is much less than the time consumed by GUITAR. GUITAR tests the underlying functionality only if we update the test cases with valid inputs. In both cases, GUITAR takes around 20 minutes to replay our test suite regardless if it is updated or not.

5.6.3 Threats to validity

As with any experiment, our work is subject to threats to validity [197]. Threats to construct validity relate to our choice of measurement as precondition violation. One may criticize our use of a precondition to conclude whether there is a fault in the code. In our experiment, we use invalid preconditions instead of using the faulty software. We make a conclusion based on that whether ESC can detect a faulty software. However, ESC is a verification tool by itself. Hence, studying the ability of ESC of discovering fault is out of the scope.

Conclusion validity is about the relation between what we manipulate and what we observe. Threats to conclusion validity are concerned with issues that affect the ability to draw the correct conclusion about relations between the manipulation and the observation of an experiment. We try to set up our experiments as systematically as possible using the ESC to avoid any bias. We study the input-output relation for only one direction, i.e., from the GUI to the Control, which may affect the conclusion validity as we did not study the other direction. However, we think that the same study can be applied on the relation from the Control to the GUI so this will not affect the conclusion.

Concerns about limitations of ESC may affect the conclusion that we made about the software having a fault or not. However, we observed an online discussion about the
limitations of ESC. The discussions assure that while ESC can handle simple equality check on strings, it cannot evaluate operations of strings processing. However, at the same time, these online discussions assure that practitioners still use ESC though it has limitations. Our point is that since these limitations did not stop practitioners from using ESC, we do not think that the limitations have a serious effect on the reliability of the results and conclusions about those results. The main paper published about ESC has more than a thousand citations.

Internal validity is about the set-up of our experiments. Threats to internal validity are influences that can affect the independent variable concerning causality, without our knowledge. We use the default ESC settings to avoid any threats of such type. For example, ESC uses the default number of counter-examples to disprove a theorem (10 examples). Hence, we kept this default setting without change.

Threats to external validity limit our ability to generalize results. We acknowledge that we use only one case study, which affects external validity. We explain results on a synthetic example and our case study so that they become less dependent on the case study to give our results a better chance of being generalizable. We couldn’t have a case study that shows all possible scenarios of input-output multiplicities. Hence, we used a synthetic example to simulate the three main types of multiplicities. We developed a GUI code in that synthetic example which implements those three main multiplicities. We showed how ESC works well in all of those three cases. We hope that this helps us to generalize our solution to apply to any scenario.
5.7 Scalability

In this section, we discuss potential issues when verifying large software using our solution. The objective of this section is to familiarize the reader with such issues. We do not explore solutions to these issues. We just highlight our expectations about them. Subsection 5.7.1 discusses time and effort issues associated with our solution. Subsection 5.7.2 discusses the formulation of a precondition. Subsection 5.7.3 discusses the location of the precondition and suggests solutions for issues related to scalability.

5.7.1 Execution time and effort

We tried our presented approach on a synthetic example, and small but real-life case study. Based on our case study, ESC combined with functional system logic tests is more efficient than GUI functional system testing in terms of time and efforts. The functional system logic test suite is able to test the underlying functionality (functionality of the logic) within a negligible time. In our Boolean logic case study, it takes few seconds. The use of the ESC to statically analyze the GUI code of our case takes around four minutes. Combining the verification part and the testing part, using our approach, our case study can be verified in less than five minutes. GUI functional system testing of our case study using GUITAR takes around 20 minutes (i.e. to replay the test suite). Generating the test suite takes around two minutes. Furthermore, GUITAR generates around 5000 test cases, out of which, 200 test cases are selected. As for manual efforts required to prepare the tests, GUI functional system testing needs updating test cases with valid inputs. This is a time-consuming procedure as described in Appendix I.
5.7.2 Formulating a precondition and adding a static variable

Our approach puts the responsibility on the software tester to formulate preconditions. This may affect the scalability of the approach. It may be more difficult to formulate a good precondition as the software grows in size. The formulation of the precondition depends mainly on the GUI code. The reason is that the precondition is written at the level of the Control as a representation of an input-output relation in the GUI code. Hence, it may be affected by the growth of the GUI. The formulation of a precondition may also be affected by the growth of the underlying functionality. As for the growth of the GUI, a software developer may write new lines of code inside a software function during corrective or preventive maintenance. The changes in code may lead to a new input-output relationship. Hence, the tester may have to redefine the precondition after such corrective maintenance.

We also discuss the scalability of our solution in the context of adapting the SUT to a new functional requirement. The adaptation causes an increase in the number of lines of code in software functions. Hence, the tester may need to understand longer parts of code and formulate more complex preconditions. This will affect negatively the scalability of our verification approach.

The software developer may decide to add new functions when implementing new functional requirements instead of updating existing functions. Hence, the new functions may implement new input-output relations instead of requiring updates to existing relations. Consequently, the tester needs to formulate more preconditions. This also may affect the scalability of our verification.
The software developer may add more classes and packages. This creates a large increase in the number of functions. Consequently, the tester needs to formulate many more preconditions when doing the verification.

As a result, if the number of functions or lines of code in a software increases, it may limit the scalability of our verification approach. This is due to the increases in the time and the difficulty of the task of formulating more preconditions and more complex preconditions.

When the software grows, it can accept more inputs. The new inputs may go to new functionalities or to old, modified ones. Hence, this may create new input-output relationships or modify existed ones. Consequently, the tester needs to add more static verification variables. This may make the verification difficult.

Maintenance to increase the software throughput by increasing the number of threads for example also increases the number of lines of code. Hence, our verification approach may not scale up when adapting new non-functional requirements of the software.

However, we think that our solution will be better than GUI functional system testing using GUITAR in terms of time and effort.

Based on our experience with ESC during the work in this thesis, we found it works at the function level. We think it may suffer issues when the SUT grows. Specifically, it may not understand some parts of the code.

We tried our solution on a much larger case study, namely Weka [207] but it did not work. Hence, we doubt that our solution scales up to handle a large case study.
Firstly, it is a large case study. It is impossible that we verify all the code. It is also impossible to verify only a part of it. This is due to the high dependence among the different code packages. The code has an extensive use of abstraction and interfaces. This extensive use of interfaces reduces the understandability of the software. Hence, it makes the formulation of the precondition harder.

Secondly, we wanted to experiment with our solution on a functionality that we can easily isolate from other functionalities. What the "Experiment" package implements was a good candidate. In order to apply our solution to the corresponding functionalities we needed to understand the machine learning algorithm implemented by Weka in this package. This turned out to be much more complex than anticipated. This was compounded by the fact that, contrary to our assumption that application logic is not solely implemented in Control classes, part of the logic of Weka appears to be implemented in the GUI. The GUI is also much more complex than anticipated, with several UI windows contributing data to the logic. It is therefore very difficult to identify which verification variables to add and where to add them. We did not find the place where the GUI calls the logic. So, we did not know where to define the precondition.

Thirdly, the ESC did not work on Weka because of dependencies Weka has with other packages. Even with some steps to solve the dependency (e.g., commenting out some code), the ESC has a difficulty to understand the code due to the use of inheritance. The same thing applies for some comments. For example, ESC does not understand the @override tag used as comment.

Finally, Weka is a software written by several developers each of them using his own style of writing code. There is a lot of abstraction due to that. The designer defined
interfaces and abstract classes. Hence, the developer of a new code needs to implement/extend the interface/abstract class. We found each classifier has a different author. It means even if we understand the programming style of one author it does not help us to understand the programming style of other authors. This makes the verification using our solution impossible as we need to understand all used coding styles.

5.7.3 The location of the precondition and solutions

We do not think that when the software scales up, it becomes much more difficult to find the place to put the precondition. It is easy to find the methods in the Control which receive output from the GUI. In case of any difficulty, Atlas can be used as suggested before.

In order to solve potential issues with large case studies, we recommend studying each package individually. In case of highly coupled classes inside the package, we recommend replacing some classes with other ones which have less code. Statements that confuse the ESC can be commented, as long as they do not impact the analysis. Identifying whether such statements do indeed not impact the analysis is no easy task. Some other blocks of code can be replaced with empty blocks. We may be able to help ESC to evaluate the code. This is possible by adding postconditions to software functions. Such a postcondition describes the value returned by the function. It may help ESC to make a judgment about the violation of a precondition to a software method which calls that software function.
5.8 Extensibility

Our approach can be extended to study the output from the Control to the GUI. However, the implementation of the approach may need to be adapted. For example, there might be a possibility to use postconditions. A method from the Control may pass a value to the GUI as an output. We can add a static variable in the Control class to store such a value. Another software method in the GUI may show that output value in a text field. We can add a postcondition to that method in the GUI. The postcondition may check if the value in the text field is equal to the value of the static variable. It may also check that the value in the text field is the same value of the static variable after a possible syntax modification (but with no semantics modification).

The limitation of ESC is that it does not handle all types of code. The tool is 10 years old so if there are new keywords for Java in future versions of the JDK, it may not be able to handle them. This may affect the extensibility negatively. If ESC cannot decide whether there is a fault, we think our solution will bring the tester's attention to a specific part of the error-prone code.

5.9 The implementation of the proposed solution using Julia

We give suggestions on the implementation of the approach using Julia static analyzer. Julia has limitations to solve our problem, however, the tool may evolve in the future to solve the problem. We show up to which level Julia is able to support researchers.

The objective of using Julia can be to obtain a mathematical formula that represents the code. The code handles the input to obtain an output. Specifically, Julia
can be used to extract a mathematical function from the code related to the input-output relation. Then, the obtained mathematical function can be compared with the expected one. The expected function is the predicted one. The software developer or tester predicts that this code (related to input-output relation) implements such a function. If the actual and predicated mathematical functions are the same, there is no fault in the code. Hence, we can verify the GUI. Subsection 5.9.1 describes an illustrative example using Julia. Subsection 5.9.2 describes ways for obtaining the invariant. Subsection 5.9.3 describes ways to write the invariant as Java code.

5.9.1 Simple, illustrative example

We use the same dummy case study used in section 5.4.5. We add a watchpoint (Line 3) to the code as shown in Table 22. Table 22 shows a part of the class CopyTextComponent after adding a watchpoint. The added line should be in the part of the code that we want to analyze (no execution). Usually, this is the part of GUI code that exists before the code that making a call to a Control class. It is actually at the intersection of the Control and GUI (section 5.2). The tester can use a tool such as Atlas [185] or as suggested in section 5.2 to identify this part in the code.

**Table 22. The adding of Julia watchpoint in the source code**

```
1 int userInput = Integer.parseInt(inputExpression.getText()); // get the user input from the text field
2 int callArgument = userInput + 1; // process the entered value by simply adding 1 to it.
3 Watchpoint.analyzeHere();
4 TextControl t = new TextControl(Integer.toString(callArgument)); // passing the value to a constructor of class TextControl in the Control package
5 t.printCopyMessage(evt.toString(),0);
```

Then, Julia, when checking the modified code of Table 22, generates an output log file with invariants (just one in this example). There is an invariant at the call to
analyzeHere() at line 3 but applied at the bytecode level. There, the constraint inferred by Julia is $\text{OL3} - \text{OL4} = -1$.

**TABLE 23. THE CONTENT OF THE FILE numericalInvariant.pl GENERATED BY JULIA**

<table>
<thead>
<tr>
<th>PathLengthAnalysis</th>
<th>public CopyTextComponent.actionPerformed(java.awt.event.ActionEvent):void *</th>
</tr>
</thead>
<tbody>
<tr>
<td>normal execution:</td>
<td>$\text{OL3} - \text{OL4} = -1$</td>
</tr>
<tr>
<td>exceptional execution:</td>
<td>$0 \geq 1, \text{OL3} - \text{OL4} = -1$</td>
</tr>
</tbody>
</table>

Specifically, Table 23 shows the content of a file called NumericalInvariant.pl generated by Julia. That is, the local variable 3 (OL3) is equal to the local variable 4 (OL4) minus 1. Here, Julia refers to local variables in the Java bytecode. Julia would help by finding the mathematical function that represents the input-output relation. In this case, the mathematical function is the difference between operations (OL3 and OL4) in a text file (Table 23). It is: $\text{OL3} - \text{OL4} = -1$. We can understand from the file NumericalInvariant.pl that the local variable three (OL3) stands for userInput and the local variable four (OL4) stands for callArgument. Julia does not tell us that OL3 represents userInput. We should find out that by ourselves by understanding the output text file NumericalInvariant.pl. We conclude that we verified the GUI for this particular scenario. The fault is discovered by reading and analyzing the text file. In other words, there is a fault for the variable callArgument (it does not have the same value entered by the user).

### 5.9.2 Obtaining the invariant

The illustrative example shown above is to obtain an invariant. Obtaining the invariant is done by using the termination checker. We try the termination checker on a
dummy case study following the steps that authors of Julia mentioned. However, we face issues / limitations we describe next when we apply static analysis using Julia on a real case study. In the dummy case study, there is a text field in the GUI. The text field receives an input from the user (Figure 23). Then, the GUI class changes the value of that input. The GUI class passes the modified input to the logic class of the software (non-GUI classes).

![Figure 23. Passing input from GUI to logic](image)

The following code shows the call to Julia at the intersection of a GUI class and a logic class. The code exists inside a GUI class. We make an instance of a Control class `TexControl` which is a logic class.

```java
int userInput=Integer.parseInt(inputExpression.getText()); // get the user input from the text field
int callArgument = userInput + 1; // process the entered value by simply adding 1 to it.
com.juliasoft.julia.checkers.Watchpoint.analyzeHere();
TextControl t = new TextControl(Integer.toString(callArgument)); // pass to the logic
```

We try to extend the dummy case study to handle two input variables instead of one. Hence, we have two text fields on the GUI instead of one. So instead of the old code, we have the following new block of code:

```java
int userInput=Integer.parseInt(inputExpression.getText()); // get the user input from the first text field
```
int userInput1 = Integer.parseInt(inputExpression1.getText()); // get the user input from the second
text field
int callArgument = userInput + userInput1; // process the entered value by simply adding 1 to it.
com.juliasoft.julia.checkers.Watchpoint.analyzeHere();
TextControl t = new TextControl(Integer.toString(callArgument)); // pass to the logic

We follow Julia online-help by setting the variable dumpNumericalInvariant to true. In the first case, Julia generates a file named
dumpNumericalInvariants.pl that has the following information: OL3 - OL5 = -1. This means that: userInput - callArgument = -1. We expect in this case to obtain: OL6 - OL5 = - OL3 which means that: userInput1 - callArgument = - userInput. Alternatively, an acceptable output will be OL3 - OL5 = - OL6 which means that: userInput - callArgument = - userInput1. We mention an acceptable output because when we compare the two codes, we think that is a rational output. We find that Julia cannot do the second analysis though it can do the first one. The file NumericalInvariants.pl generated by Julia do not include any invariant in the second case. We conclude that Julia has limitations when the code becomes complex.

One can try with other checkers to solve this issue. For example, we can try with the injection checker [208]. This checker is supposed to report issues related to taint analysis [208]. For example, if a parameter of a function is suspected to be tainted (e.g., changed by malicious code), the injection checker of Julia reports that part of the code which contains the parameter that produces a risk. Unfortunately, when using the injection checker, we get an error during the analysis. Julia does not continue the analysis and reports an error. The error message simply says an error while connecting to the server. The deployed version of the tool is not ready for such analysis.
One can try other options. For example, we can explore the use of an option in Julia called the polyhedra for small software to study the invariants. The authors of the Julia suggest the use of polyhedra option. The authors claim that polyhedra has a high computational ability. If one prepares a very simple Java class in which a method `a()` calls a method `b()`. If we define two integer variables `x = 1` and `y = 1` inside `a()` and then another integer variable `z = x + y` also inside `a()`, then we pass `z` to `b()`, Julia can find the invariant using `dumpNumericalInvariant = true`. However, if we move the declaration of `x` and `y` to the `main()` function, Julia cannot find the invariant using the `dumpNumericalInvariant`. So, we try to set `onlyPolyhedra` to `true` and run the analysis. We get the same error message that we obtained when we try the injection checker.

In summary, Julia can help with a specific case study, but one faces an issue on a complicated case study.

One can study numerical differences [209], which are representations of sets of points in some vector space. Bounded differences are constraints of the form:

```plaintext
var1 <= constant
var2 >= constant
var1 - var2 <= constant
```

Bounded differences are sometimes also called zones. Zone is the default domain used by Julia for termination analysis and the only one available online. Julia can also use an option called polyhedra, or both bounded differences and polyhedra together, falling back to bounded differences when polyhedra are not needed. This option is unavailable online due to internal issues related to the company that owns Julia. One possibility would be to use polyhedra and bounded differences for the application under analysis and
bounded differences only for the libraries used by the software developer such as Java Swing library. However, while Julia authors suggest this, they also confirm that the Julia team has never tested the polyhedra.

The fact that polyhedra does not scale up in some cases is due to the amount of code that Julia must check. The static analysis tries to analyze everything statically. It does so through approximations. If the code uses a lot of libraries, such as a decent Java program, then there is simply too much code (and especially code that is not necessarily accessible to the static analyzer) for the analyzer to finish in a decent amount of time (if at all).

One can search for ways to proceed in such circumstances (limitations) to help the static analyzer. In other words, to reduce the computations that the static analyzer need to do. For instance, some steps can be done to help analyzer by reducing computations. If the code we analyze uses a function from a library that we (humans) know returns an integer between 1 and 10, then we can inform the analyzer of such a situation. Hopefully, such steps could help the analyzer.

One can think of different ways to help Julia finds relationships between variables defined in the code. Specifically, we believe we could ourselves look at the code and identify information that could simplify Julia’s analysis. One can investigate whether other kinds of annotations could be of any help to us, including:

1. Annotations to specify bounded ranges of values for different types of variables (other than numerical variables).
2. Annotations to specify specific values for variables. These values could be enumerated or continuous.

3. Annotations to specify exceptional / illegal values. These values could be indicated easily as we already do that for handling invalid input. Hence, using annotation, Julia excludes such values from the set of possible values of a variable. If we have an option, then, we can express bounded ranges, using specific annotations.

4. Annotations to tell Julia to ignore some calls.

5. Annotations to tell Julia to ignore some variables.

6. Annotations to tell Julia that some calls / variables that we (humans) can tell would not affect results. Hence, again Julia can exclude them from computations. For instance, one possibility might be to tell Julia to ignore a specific call (e.g., a call to display the UI) in the analysis.

    There are no annotations in Julia to do the listed items above. That might be related to the analysis based on bounded difference: there are bounded differences that Julia can automatically infer from the code, and there are situations where Julia needs help to do so. We found annotations for a specific example that is not interesting for us. The example is related to a checker called GuardedBy checker. As for termination checker which is totally different from the GuardedBy, there is no way to use those annotations. There are no annotations in Julia to inform it that some calls have nothing to do with the analysis: for instance, if we have code (calling library functions) that construct the layout of the UI, then there is no point for Julia to try to analyze the code of
those called functions. There are only annotations that Julia understands for code synchronization issues.

Assuming that some annotations can help the termination checker, it is unclear how it is possible for the user to help the analyzer to avoid analyzing some code. For example, avoid a code such as the one in swing package (import java.swing.JFrame). There are no values within a specific range that we can tell the analyzer about them in our case study. If the analyzer knows that the value definitely belongs to a specific range, this may reduce computations.

Bounded abstractions only involve numerical values. One cannot define a bounded abstraction for other types. Even for some other types, one can express some constraints can as numerical values. For instance, if an input is a character, one can use the ASCII code, which is a value. If one deals with an array, one can specify a constraint on its length. Our objective is to provide information (perhaps under the form of constraints) about the Java GUI framework that is being used. Such tries will lead to success with the current status of technology.

5.9.3 Writing the invariant as Java code

The objective is to know whether a mathematical function does represent the code related to the input-output relation. Instead of asking Julia to obtain the invariant, one can ask Julia to tell us whether our predicated invariant is correct. We believe that one can reduce the computations that Julia needs to do when we write the invariant and ask Julia to verify it instead of asking Julia to find it. We justify the disability of Julia to find the invariant in complex cases due to the high computations required to achieve that. The
validation of an invariant is easier than finding that invariant. Hence, the answer of Julia will be yes or no. Yes: means the invariant (mathematical function/condition) is a valid one. No: means the invariant is invalid. Since our invariant is a predicated one, it may be a subset of the real invariant. This could be a limitation of this solution.

In this section, we present a way through which we write the invariant as a Java code. Recall that the invariant is a condition that is always satisfied regardless of the input of the software. One can write an if-statement in the Java code that has the invariant as a condition. If the condition is satisfied, then, the if-statement will call an infinite loop that we write inside the body of that if-statement. One can ask Julia to analyze the code for a possible non-termination of the software. When the program goes into infinite loop, it does not terminate. Julia should warn about a possible scenario. The non-termination usually happens due to bad practices by the software developer. The code out of such practices leads the software to freeze (i.e., stop working, hanging). We highlight the fact that if the condition of if-statement is satisfied, then this leads the program to stop (does not terminate), hence, Julia should warn about possible non-termination. It means that Julia should statically evaluate the code to decide whether the condition is satisfied.

Since our condition represents an input-output relation, we can use Julia to validate whether this mathematical function represents the code. In other words, we try to use the termination checker to detect if an invariant holds instead of generating the invariant.

We show next how one can add an invariant in the Java code. Hence, if the invariant does hold (satisfied all the time), the while(true) in the following code is called, and the termination checker gives us a warning. Unfortunately, we face a
limitation of Julia if the variables are uninitialized. We show this limitation through next steps.

We use an example to show up-to-which level Julia can help researchers. The example shows that Julia works on the simple case. When we update the example to have more complex code, Julia does not work well. We show this limitation in an incremental way: we start with a working copy of the example and end up with an unsuccessful one. For the following code, there is no warning reported by Julia. The condition of if-statement is not satisfied. The program does not proceed to the while loop. Julia works well with this example (it gives no warning).

```java
public class WhileTrue {
    public WhileTrue() {
    }
    public static void main(String[] args) {
        int x = 0;
        int y = 0;
        if (x == 1 || y== 1) while (true);
        System.out.println("Passed while ture");
    }
}
```

For the following code, all is fine as well, there is a warning reported by Julia. The termination checker finds that the program will go into an infinite loop. This is expected because the condition in the if-statement is satisfied. It means the program will not terminate. Termination checker warns about program possible non-termination.

```java
public class WhileTrue {
    public WhileTrue() {
    }
    public static void main(String[] args) {
        int x = 0;
        int y = 0;
        if (x == 0 || y== 0) while (true);
        System.out.println("Passed while ture");
    }
}
```
If we use a dummy case study which takes two inputs from the GUI, Julia throws a warning for the `while(true)` whether the condition (invariant) is stratified or not. It means that Julia cannot evaluate the code. Here is the example:

```java
public TextControl(String t) {
    // verification code
    int guiVariable=Integer.parseInt(t);
    int expectedGUIvariable=CopyTextComponent.getInputExpression() + CopyTextComponent.getInputExpression2();
    if (guiVariable == expectedGUIvariable) while (true);
    // end of verification code
    myText = new MyTextEntity(t);
}
```

The issue that we face is that Julia reports a warning in the two cases of invariant:

- `guiVariable == expectedGUIvariable`
- `guiVariable != expectedGUIvariable`

The reason for which Julia reported a warning in both scenarios is that Julia was unable to evaluate the condition. So, it warned in both cases. The reason for this disability with Julia is that the variables are uninitialized, and they will get their values only at the run-time. So, Julia cannot evaluate them. We conclude that while Julia works with a simple example. However, we cannot use Julia when we have several variables.

The termination analyzer can evaluate bounded difference constraints. The analyzer uses those constraints in the case with the line `if (condition) while(true)`. Everything which is not in the form `var1 - var2 <= constant` will just be discarded and abstracted away. However, when we have complicated code, the termination checker has limitations.
Consequently, one cannot move further in this direction. We communicated all these cases with Julia authors. The authors informed that they are working to enhance Julia in a future deployment.

5.10 Conclusion

In this chapter, we investigate the use of static analysis to verify the GUI. We describe the input-output relation at the GUI level. We describe the multiplicities of the input-output relation. We present an approach to verify the GUI without testing. We implement the approach using Extended Static Checker (ESC). We validate our solution. We discuss the possibility of implementing this approach using Julia. This solution overcomes other solutions of GUI functional system testing. Running ESC on our case study takes 2 to 3 minutes. For example, for our case study, GUITAR generates more than 5000 test cases that (out of which GUITAR selects 200 test cases). Running a test suite of GUITAR takes 20 minutes whether we update the test cases with valid input or not. The manual work of GUI functional system testing requires much more efforts than the manual work of our solution. For example, updating test cases with a valid input is much more effort consuming task than adding verification variables and preconditions to the code. The technology we suggest has limitations. For example, Julia does not generate a mathematical function that describes a software method all the time. However, this technology is evolving. We expect that the technology will overcome these limitations shortly [210]. For the time being, it is possible to use Java Modeling Language to reach the objective of GUI verification without testing [211].
Chapter 6   Conclusion

This PhD thesis explores the concept of system testing and GUI testing by first checking their relationships. Then, based on the equivalency test, the thesis bridges the gap between the two aforementioned software testing types by using static analysis for the GUI code. Section 6.1 summarizes the work in the thesis. Section 6.2 presents the future work.

6.1 Summary

Chapter 1 presents an introduction to the topic and shows the confusion in the literature about definitions of GUI testing and system testing. It shows how this leads to issues for practitioners. One of those issue is a confusion on which tool to use. Another issue is the redundancy in testing types.

Chapter 2 presents the background and a collection of important definitions from the literature of system testing and GUI testing and shows that it is critical to have a clear definition of GUI testing. For example, having a clear definition of GUI system testing allows software practitioners to have a consensus on stopping criteria for testing of the business logic of GUI software applications and hence avoid the effort of testing the underlying functionality of the software twice: once when testing the software with desktop GUI and a second time when testing the software with web GUI implementations. Chapter 2 answers Research Question 1: What are the available definitions for system testing/GUI testing and how do they relate to each other? Then, it
presents a taxonomy of definitions. Then, it classifies a part of published works based on that taxonomy.

Chapter 3 presents background-related work for static analysis. We describe selected techniques of static analysis. For instance, we describe abstract interpretation, symbolic execution and Extended Static Checker. We then move to summarize published work on static analysis for software verification.

Chapter 4 presents an experimental investigation of the concept of GUI testing by making a comparison of structural coverage for functional system logic testing and GUI functional system testing on a case study. It answers Research Question 2: Is GUI system testing of the software logic equivalent to functional system testing applied to the logic by bypassing the GUI in terms of structural coverage? We argue that if the structural coverage obtained by those two types of testing is equal, then those two types of testing are doing the same thing and hence it is enough to have one of them to verify the application logic.

The experiments use GUITAR to perform GUI functional system testing for a GUI-based software. The objective of the experiments was to understand the difference between GUI testing and system testing. The experiments show that functional system logic testing is better than GUI functional system testing in terms of cost-effectiveness. Although more studies are necessary to precisely understand the phenomenon we have encountered, this may suggest that GUI system testing and functional system testing are two different testing activities, i.e., that GUI system testing may not be system testing through the GUI of a GUI-based software. Repeating those experiments on another case study or using another GUI test suite may help to gain a better understanding.
Chapter 5 focuses on a different contribution of this thesis by bridging the gap between GUI functional system testing and functional system logic testing using static analysis. We answer Research Question 3: Is it possible to verify a GUI-based software using static analysis of the GUI code combined with system tests on the application logic? In our experiments, we show that the use of static analysis can help verify the GUI code without doing testing. We use the static analysis because we do not want to execute the GUI code. The reason is that the execution of the GUI code is time consuming. By adding that to functional system logic tests, it would then be sufficient to conclude that a software application is adequately verified. In other words, there is no need to GUI functional system testing.

Then, we show that using functional system logic testing combined with static analysis of GUI classes of such large software is enough to adequately verify that software. This can be done in less computation time than GUI functional system testing.

We study what we called the input-output relationship. That is the relation between input variables in the GUI and arguments of methods in the Control. We study one direction of this relation. The same study can be applied on the other direction from the Control to the GUI.

6.2 Future work

A future work can be the use of a capture/replay tool instead of GUITAR to do the comparison described in Chapter 4. Future experiments may include the quantitative and qualitative assessment for the manual effort attached to the techniques described in the thesis and measurements used in the thesis, respectively. The manual efforts include
the modification of the GUI test suite with valid inputs, finding widgets in the Event Flow Graph, finding coordinates of text fields and other input widgets, specifying which test cases related to which button, etc. The time that tester spends in such activities can be quantified to have better understanding on how much time GUI functional system testing takes. As for qualitative assessment, the experiments may include different kinds of measurements such as fault detection, different kinds of coverage, the use of postconditions instead of preconditions, etc. The experiments may include repeating/replicating experiments on several case studies. It will also give results with higher quality if experiments of functional GUI system testing can be repeated on different operating systems with different versions of Java development kits (JDKs). We defined a taxonomy, which is a contribution. We applied that contribution in the context of our thesis on examples selected from literature. A future work we can mention is to do further studies on the applicability of this taxonomy.
Bibliography or References


66. Yusifoglu, V.G., Y. Amannejad, and A.B. Can, 


69. Garousi, V. and E. Yildirim, *Introducing automated GUI testing and observing its benefits: an industrial case study in the context of law-practice management software,* in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW).* 2018, IEEE.


177. Yuan, X. and A.M. Memon. *Using GUI run-time state as feedback to generate test cases.* in *29th International Conference on Software Engineering (ICSE'07)*. 2007: IEEE.


204. Snyder, C., *Paper prototyping: The fast and easy way to design and refine user interfaces*. 2003: Morgan Kaufmann.

205. Tidwell, J., *Designing interfaces: Patterns for effective interaction design*. 2010: "O'Reilly Media, Inc.".


Appendix A  The complete list of books, both from the research laboratory and the Library

- Black, R., Advanced software testing. 2009, Santa Barbara, Calif: Rocky Nook.
- Bruegge and Dutoit, Object-Oriented Software Engineering: Using UML, Patterns and Java. 2000.
- Miguel Sales Dias, Sylvie Gibet, Marcelo M. Wanderley, Rafael Bastos, Gesture-Based Human-Computer Interaction and Simulation. 7th International Gesture Workshop. 2009, Lisbon: Springer Science & Business Media
• Koirala, S., Sheikh, S., Software testing interview questions. 2008, Sudbury, Mass: Jones and Bartlett.
• Li, K. and M. Wu, Effective software test automation: developing an automated software testing tool. 2006: John Wiley & Sons.
• Mathur, A. Foundations of software testing: fundamental algorithms and techniques. 2013, New Delhi: Dorling Kindersley (India).
• McGregor, J.D. and D.A. Sykes, A practical guide to testing object-oriented software. 2001: Addison-Wesley Professional.
• Mitchell, J.L., R. Black, Advanced software testing. 2015, Santa Barbara, Rocky Nook.
• Pezze, M. and M. Young, Software testing and analysis: process, principles, and techniques. 2007: John Wiley & Sons.
• Roper, M., Software testing. 1994, New York; London; McGraw-Hill.
Rubin, J. and D. Chisnell, Handbook of usability testing. How to plan, design, and conduct effective tests. 2008: Google Books.


Sharma, M., R. Padmanaban, and C.R.C. Press, Leveraging the wisdom of the crowd in software testing. 2015, Boca Raton: CRC Press.


Appendix B  Steps of GUI testing (Demo of using GUITAR on our case study)

This appendix illustrates the use of GUITAR for GUI-based testing [3]. GUITAR contains a set of tools for testing applications written using Java Foundation Classes (JFC). JFC is a term used to name a graphical framework for building portable Java-based graphical user interfaces (GUIs). JFC consists of the Abstract Window Toolkit (AWT), Swing and Java 2D to generate user interface for Java programs that can run independently from the operating system. GUITAR can be used to automatically test a JFC applications by using five components. The following subsections describe the Ripper work, the use of Convertor, the use of Visualizer to get a graph representation, the use of generator to create test cases, the Replayer's work when running the test cases on the application’s GUI.

**Traversing GUI to build GUI-Tree**

Ripper is the tool that is capable of extracting GUI structural information from GUI applications through interacting with GUI widgets. The default GUI events such as left click on clickable widgets (e.g., push button, radio button check box) and type-in-text on editable widgets (e.g., text box, text area) are recognized and captured. In addition, the Ripper uses an algorithm that needs specific guidance to discover relationships among GUI events and widgets (e.g., which widget cause which event). This information is encoded in the tool’s adapters. Ripper includes custom adapters for ripping containers and then move to contained widgets. For the Boolean logic software, it starts from Java
Frame class as it is the base class that contains all other widgets. Then it moves to Container class, then to panel and then to the text field or button on top of that panel. Some non-standard (e.g., those defined by user) widgets and their properties may be missed during ripping but this is not the situation with our case study as we use standard widgets. Table 24 shows the tags of the first level representing GUI.

**Table 24. XML for the first level of GUI of the Boolean logic application**

```xml
<?xml version="1.0" encoding="UTF-8" standalone="true"?>
<GUIStructure>
  <GUI>
    <Window>
      <Attributes>
        <Property>
          <Name>ID</Name>
          <Value>Main Exe Frame</Value>
        </Property>
        <Property>
          <Name>Modal</Name>
          <Value>false</Value>
        </Property>
        <Property>
          <Name>Rootwindow</Name>
          <Value>true</Value>
        </Property>
      </Attributes>
    </Window>
    +<Container>
  </GUI>
</GUIStructure>
```

The representation of a GUI starts from the main frame which contains all other components. The properties of the main frame are represented as tags and values are given to those properties. One of the XML tags is the `<Container>`. Since the application is written in Java, Java frame has a container over it which usually contains a panel or directly contains the individual widgets. Expanding the container tag shows the properties of that container and the set of nested groups of XML tags. Each of these groups represent a widget. Table 25 shows the XML tag representing widget after expanding the `<Container>` tag.
The main tag is the ID tag. Each widget is given an ID. This ID is used in traversing GUI components. For each widget on the GUI, a group of XML tags is formulated. The tag again represents a set of properties for the widget and their values.

**Converting the GUI-Tree to a graph**

Converter (Graph convertor) converts the GUI Tree generated by the Ripper into an Event Flow Graph (EFG), based on an event-flow model. An EFG is a directed graph representing the event interactions on a GUI. Each node in an EFG represents a GUI
event (e.g., click-on-Create, click-on-OK). An edge from node \( v \) to node \( w \) represents a follows relationship between \( v \) and \( w \), indicating that event \( w \) can be performed immediately after event \( v \). An EFG is analogous to a control-flow graph, in which vertices represent program statements and edges represent execution flows between the statements. The Convertor takes the output of the Ripper and transform it into an XML file that represent the EFG.

Table 26 shows three events represented as XML: \( e_0 \), \( e_1 \) and \( e_2 \). The three major tags for each event are: EventId which is a unique ID given to the event, WidgetId which is the identifier of the widget that triggers this event, and Type which is the tag that describes the type of event.

**Table 26. XML representation for EFG of the Boolean Logic application**

```xml
<?xml version="1.0" encoding="UTF-8" standalone="true"?>
<EFG>
  <Events>
    <Event>
      <EventId>e0</EventId>
      <WidgetId>w11</WidgetId>
      <Type>EXPAND</Type>
      <Initial>true</Initial>
      <Action>edu.umd.cs.guitar.event.JFCSelectionHandler</Action>
    </Event>
    <Event>
      <EventId>e1</EventId>
      <WidgetId>w21</WidgetId>
      <Type>SYSTEM INTERACTION</Type>
      <Initial>true</Initial>
      <Action>edu.umd.cs.guitar.event.JFCEditableTextHandler</Action>
    </Event>
    <Event>
      <EventId>e2</EventId>
      <WidgetId>w25</WidgetId>
      <Type>SYSTEM INTERACTION</Type>
      <Initial>true</Initial>
      <Action>edu.umd.cs.guitar.event.JFCActionHandler</Action>
    </Event>
  </Events>
</EFG>
```

For example, event could be some movement on the GUI like expand event or interaction with underlying functionality like \( e_1 \).
Table 27. Part of Event Flow Graph's XML file for Boolean Logic Application

<table>
<thead>
<tr>
<th>Event Graph</th>
<th>&lt;EventGraph&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Row&gt;</td>
<td>&lt;Row&gt;</td>
</tr>
<tr>
<td>&lt;E&gt;1&lt;/E&gt;</td>
<td>&lt;E&gt;1&lt;/E&gt;</td>
</tr>
<tr>
<td>&lt;E&gt;2&lt;/E&gt;</td>
<td>&lt;E&gt;2&lt;/E&gt;</td>
</tr>
<tr>
<td>&lt;E&gt;2&lt;/E&gt;</td>
<td>&lt;E&gt;2&lt;/E&gt;</td>
</tr>
<tr>
<td>&lt;E&gt;2&lt;/E&gt;</td>
<td>&lt;E&gt;2&lt;/E&gt;</td>
</tr>
<tr>
<td>&lt;E&gt;2&lt;/E&gt;</td>
<td>&lt;E&gt;2&lt;/E&gt;</td>
</tr>
<tr>
<td>&lt;E&gt;2&lt;/E&gt;</td>
<td>&lt;E&gt;2&lt;/E&gt;</td>
</tr>
<tr>
<td>&lt;E&gt;2&lt;/E&gt;</td>
<td>&lt;E&gt;2&lt;/E&gt;</td>
</tr>
<tr>
<td>&lt;E&gt;0&lt;/E&gt;</td>
<td>&lt;E&gt;0&lt;/E&gt;</td>
</tr>
<tr>
<td>...........</td>
<td>...........</td>
</tr>
<tr>
<td>&lt;E&gt;2&lt;/E&gt;</td>
<td>&lt;E&gt;2&lt;/E&gt;</td>
</tr>
<tr>
<td>&lt;Row&gt;</td>
<td>&lt;Row&gt;</td>
</tr>
<tr>
<td>&lt;/EventGraph&gt;</td>
<td>&lt;/EventGraph&gt;</td>
</tr>
</tbody>
</table>

Table 27 represents for relationships between different events. This relationships create numerical representation used by GUITAR to represent the communication among the different events. Graph convertor by itself is unable to show the EFG visually. The XML format used by Graph convertor is not readable by graph visualizer software. However, the distribution of GUITAR comes with a jar file that can convert the XML files to another format that is readable by some visualizers like Gephi. Figure 24 shows visual representation of the EFG.

**Generating test cases**

Generator is the test cases generator of the GUI application. The resulting suite contains one test case for each event sequence of length L in the input EFG, where L = 1 results in a suite that covers every event in the EFG and L = 2 results in a suite covers every pair of events occurring in the EFG, and so on for longer lengths.
Each individual test case covers one sequence of events with length \( L \). A prefix is applied to the sequence of events to make it accessible from the application’s initial state and hence the actual length of test cases may exceed the value of \( L \). For Boolean logic application, Generator 1.2 generated a test suite with 82 test case when length of test case is equal to 2. Table 28 shows an example of test case which is test case 5 that contains
two events \(e_0\) and \(e_1\). These two events are extracted from the EFG built by the Convertor. The structure of the XML files representing a test case consists of a set of steps each of them is specialized for one event. The generator does not create a test case for each objective (i.e. as many test cases as test objective). However, the generator has a set of setting to be made inside a properties file used with XML build file as part of ant technology to build the application. Maximum number of test cases and length of test case are two examples of those options.

**Replaying test cases**

The Replayer executes test cases on a GUI-based software. The Replayer implements TestMonitors that affect the state of the GUI after executing each event in the test case and store that information in an XML file. Replayer executes a test case in isolation rather than in the exploratory fashion of the Ripper. Thus, an observed state may be unique, having never occurred during the ripping process. This XML GUI state file can be analyzed to determine if a test case passed. The analysis may include comparing the state with the state stored from a previous execution, for example, during regression testing. Authors mentioned that a simple shell script, acting as a test oracle, examines log files generated by Replayer. The script also detects exceptions and error strings from execution logs, which human testers can then manually inspect. Our understanding is that this script/oracle can be used in general testing, however, the script is unavailable in GUITAR distribution.

Replaying a test cases is a time consuming process as it requires high computational resources, and hence Replayer asks the user for the number of test case (s) that user wants to replay. A special version of GUITAR can be used on a cluster of
machines. Replayer replays the test cases on the GUI and generates XML files that contain in addition to the <Step> tag, the <GUIStructure> tag which describes the state of the GUI before and after each event happens. Table 29 shows the output of replaying test case 5 showed in Table 28.

**Table 29. The state trace file content**

```xml
<?xml version="1.0" encoding="UTF-8" standalone="true"?>
<TestCase>
  <Step>
    <EventId>e5</EventId>
    <ReachingStep>false</ReachingStep>
    <GUIStructure>
      <GUI>
        <Window>
          <Attributes>
            <Property>
              <Name>ID</Name>
              <Value>Main Exe Frame</Value>
            </Property>
            <Property>
              <Name>Modal</Name>
              <Value>false</Value>
            </Property>
            <Property>
              <Name>Rootwindow</Name>
              <Value>false</Value>
            </Property>
          </Attributes>
        </Window>
        <Container>
          <GUI>
            <GUIStructure>
              <Step>
                <Step>
                </Step>
              </Step>
            </GUIStructure>
          </GUI>
        </Container>
      </GUI>
    </GUIStructure>
  </Step>
</TestCase>
```
Appendix C  Measuring coverage of automated GUI-testing

Measuring coverage of GUI-testing is different from measuring coverage in the general case of software testing. An integration step is required between some components of a GUI-testing tool like Replayer and a coverage tool like COBERTURA in order to measure the coverage of test cases used in GUI-testing for GUI-based SUT. In this appendix, we use the term framework to represent both of: (a) A structure of folders to contains all code and artifacts related to experimental work like: SUT, instrumented code, output, GUITAR tool for GUI-testing, COBERTURA tool for measuring coverage and (b) A set of shell scripts adapted from open source solutions to control the flow of experiments related to GUI-testing and coverage measurement. Both the structure of folders and shell scripts work as framework to solve the mentioned integration issue. The following subsections describes the framework.

**Folders structure and shell scripts**

The framework includes two main parts: A set of folders to organize experiments and a set of shell scripts to control the flow of experiments. We arranged folders in convenient structure that facilitates the work of user to conduct GUI-testing on SUT and then measure the coverage after that. Figure 25 shows the arrangement of folders.
The main folder in the framework is called “framework” that has the following contents:

1. A set of jar files, shell scripts and batch files related to COBERTURA
2. lib: A folder contains a set of jar files related to COBERTURA
3. jfc-guitar-v1.1.1: A folder contains jar files and other artifacts related to GUITAR
4. myExample: Slightly modified copy of an older myExample folder contains a set of files related to SUT and shell scripts to control the flow of experiment. For myExample folder we kept the same old structure of folders but dropped one folder called “class” and added one folder called “Demo” and hence the myExample folder has the following contents:
   a. instr: A folder contains the instrumented code by COBERTURA
   b. source: A folder contains the java classes of SUT
c. results: A folder contains the output of coverage when running COBERTURA on SUT in html format
d. Demo: A folder contains output of GUI-testing when running GUITAR on SUT This folder contains the following files and subfolders: Demo.GUI: An XML file that includes the tree-structure representation of GUI for SUT as XML after the ripping process performed by jfc-ripper; Demo.GUI: An XML file contains the EFG for SUT after converting the Demo.GUI by jfc-convertor; testcases: A folder contains a set of test cases extracted from EFG by jfc-generator; states: A folder contains XML files describe the state of GUI after replying the test cases on SUT; logs: A folder contains the logs of executions when running GUITAR on SUT

We dropped the folder “class” created which is supposed to contain the bytecode of SUT. Instead, we used the folder jfc-aut inside the jfc-guitar-v1.1.1 to contain bytecode of the SUT. The reason for that is jfc-aut folder should be as shared folder between COBERTURA and GUITAR for any SUT. For an SUT called X, a folder called X should be created inside jfc-aut that contains two subfolders:

1. bin: a folder contains byte code of X
2. guitar-config: A folder contains configuration files that GUITAR uses when testing the SUT (e.g. GUITAR needs to know what is the main java class)

The set of shell scripts are distributed over the different folders to control a process of two phases: (a) GUI testing and (b) replaying and measuring coverage. The folder myExample contains the script “runCoberturaOnMyExample-EnvVar-guitar.sh” which is an updated copy of script “runCoberturaOnMyExample-EnvVar.sh” to use the bytecode in the new folder of bytecode “jfc-aut” instead of the
old one “classes”. The script “runBareHello.sh” was updated to have calls to another scripts called “jfc-sample-workflow-MyFrame-Replayer.sh” existed in jfc-guitar-v.1.1.1 folder. jfc-sample-workflow-MyFrame-Replayer.sh is a modified copy of original script included in GUITAR package called “jfc-sample-workflow.sh” that operates the four main components of GUITAR: jfc-ripper, jfc-convertor, jfc-generator and jfc-replayer. The update was done to disable all operation unrelated to jfc-replayer because when measuring coverage we assume that GUI-testing is already done and test cases have been generated and hence no need to run components like jfc-generator again. In other words, COBERTURA communicates with only one components of GUITAR which is the jfc-replayer. The structure of folders has been upgraded later.

**Primitve case study using GUITAR v1.1 and COBERTURA 2.0**

The purpose of this simplified case study is to understand the suitability of the proposed framework to measure coverage of an SUT after doing GUI testing. To do that, we created simple java class that has GUI. The code is available in the framework folder under MyExmple folder in source folder. The name of the class is MyFrame.java. Figure 26 shows the output of MyFrame program and Table 30 shows source code.

![GUI of MyFrame Program](image)

**FIGURE 26. GUI OF MYFRAME PROGRAM**
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class MyFrame extends JFrame {
    private JTextArea t1,t2;
    private JButton copy;

    public MyFrame()
    {
        super("TextArea Demo");
        Box b = Box.createHorizontalBox();
        String s = "This is a demo string to 'n"+
                "illustrate copying text 'n"+
                "from one TextArea to 'n"
                "external event'\n",
        t1 = new JTextArea (s,10,15);
        b.add(new JScrollPane (t1));
        copy= new JButton("Copy >>>");
        copy.addActionListener(new ActionListener () {
            public void actionPerformed(ActionEvent e)
            {
                t2.setText(t1.getSelectedText());
            }
        });
        b.add(copy);
        t2= new JTextArea(10,15);
        t2.setEditable(false);
        b.add(new JScrollPane(t2));
        Container c = getContentPane();
        c.add(b);
        setSize(424,200);
        show();
    }

    public static void main(String[] args)
    {
        MyFrame mf=new MyFrame();
        mf.addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent e)
            {
                System.exit(0);
            }
        });
    }
}

However, the source code is unnecessary to measure coverage in our experiment but the bytecode. The bytecode file MyFrame.class is available inside MyFrame folder in jfc-aut folder in jfc-guitar-v1.1.1 folder in framework folder. Before measuring coverage using the framework. It is necessary to run the GUI-testing process first. In order to do that, it is necessary to run the script “jfc-sample-workflow-MyFrame.sh” available
inside jfc-guitar-v1.1.1. This script is a modified copy of script called jfc-sample-workflow.sh available in GUITAR package. The modification was minor one and is only related to configuration of GUITAR to test MyFrame program. As mentioned before that a set of configuration files should be included inside guitar-config folder in MyFrame folder in jfc-aut folder in jfc-guitar-v1.1.1 folder. However, if there is a conflict between jfc-sample-workflow.sh and other configuration files in guitar-config folder, the ones in sample-workflow.sh will be applied. In our case study, the ones in jfc-sample-workflow-MyFrame.sh will be applied as we don’t use jfc-sample-workflow.sh. Running jfc-sample-workflow-MyFrame.sh produces 200 test cases in the Demo folder inside framework folder. Each test case is represented in a single file with tst extension. In this case study we used only test case 1 when collecting coverage measurements. After finishing GUI-testing using GUITAR, it would be possible to measure the coverage. In order to that, it is enough the run the script runCoberturaOnMyExample-EnvVar-guitar.sh in the myExample folder. The output of COBERTURA will be available in the results folder inside MyExmaple folder. Table 31 and Table 32 show the results and detailed results of using COBERTURA on our MyFrame program and other non-gui code used before.

### Table 31. Coverage report generated by COBERTURA

<table>
<thead>
<tr>
<th>Package</th>
<th>#Classes</th>
<th>Line Coverage %</th>
<th>Line Coverage</th>
<th>Branch Coverage %</th>
<th>Branch Coverage</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>(default)</td>
<td>5</td>
<td>95%</td>
<td>47/49</td>
<td>100%</td>
<td>6/6</td>
<td>2.333</td>
</tr>
<tr>
<td>All Packages</td>
<td>5</td>
<td>95%</td>
<td>47/49</td>
<td>100%</td>
<td>6/6</td>
<td>2.333</td>
</tr>
</tbody>
</table>
### Table 32. Detailed coverage report generated by COBERTURA

<table>
<thead>
<tr>
<th>Classes in this Package</th>
<th>Line Coverage %</th>
<th>Line Coverage</th>
<th>Branch Coverage %</th>
<th>Branch Coverage</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hello</td>
<td>100%</td>
<td>18/18</td>
<td>100%</td>
<td>4/4</td>
<td>2.5</td>
</tr>
<tr>
<td>MyFrame</td>
<td>100%</td>
<td>20/20</td>
<td>N/A</td>
<td>N/A</td>
<td>0</td>
</tr>
<tr>
<td>MyFrame$1</td>
<td>100%</td>
<td>3/3</td>
<td>N/A</td>
<td>N/A</td>
<td>0</td>
</tr>
<tr>
<td>MyFrame$2</td>
<td>33%</td>
<td>1/3</td>
<td>N/A</td>
<td>N/A</td>
<td>0</td>
</tr>
<tr>
<td>NewHello</td>
<td>100%</td>
<td>5/5</td>
<td>100</td>
<td>2/2</td>
<td>2</td>
</tr>
</tbody>
</table>

The results in myExample folder are shown as html files and describe the coverage on the source code of SUT for test case 1.

**Changes to shell scripts**

After arranging the structure of folders and shell scripts, it was observed that COBERTURA was unable to collect coverage measurement for MyFrame.java but it was able to do that for other java files like Hello.java and NewHello.java. The only difference between MyFrame.java and other two java files is that MyFrame.java uses GUI but Hello.java and NewHello.java do not and hence COBERTURA was not in need to communicate with jfc-replayer in order to run Hello.java or NewHello.java. Specifically, though COBERTURA was able to trigger the jfc-replayer to replay selected test cases on MyFrame.java, result of coverage was 0 all the time for MyFrame.java. To have an interpretation for this observation, we examined the shell scripts used by jfc-replayer. We found that the script jfc-sample-workflow-MyFrame-Replayer.sh does not trigger jfc-replayer jar files directly but it calls another scripts called “jfc-replayer.sh” that triggers the jar files of jfc-replayer. While calling the last script, the
former script passes the value of “class_path” variable using -cp option for java command. We had to add the line used before:

    export _JAVA_OPTIONS='-
Dnet.sourceforge.cobertura.datafile=coberturaAll.ser'

to the script jfc-sample-workflow-MyFrame-Replayer.sh. Then we defined variable:

    aut_classpath1=${CLASSPATH}:${aut_classpath}

Finally, we updated the command to use aut_classpath1 instead of aut_classpath as the following:

    cmd="$SCRIPT_DIR/jfc-replayer.sh -cp $aut_classpath1 -c $mainclass -g $gui_file -e $efg_file -t $testcase -i $intial_wait -d $relayer_delay -l $logs_dir/$test_name.log -gs $states_dir/$test_name.sta -cf $configuration -ts"

This was the only change that has been done to scripts associated with GUITAR package. After doing this change, COBERTURA was able to collect coverage measurements when jfc-replyer replayed test case 1 on MyFrame program. The next step in this research is to use boolean logic application as SUT. Also it could be possible to develop user interface for the framework to facilitate its use.
Appendix D  Steps for conducting GUI-testing on Boolean logic software using Replayer v.1.1

1. In a Linux platform, download the software package from [212]

2. Open the following sequence of folders: Cobertura-Ant, jfcguitar-v1.1.1

3. Do the configuration by filling configuration options that are available in the execution script file jfc-sample-work-theMix.sh which overrides the configuration file available in SUT folder (named jfc-aut)

4. Remove the folder Demo that contains results of all previous experiments

5. Run the shell script: jfc-sample-work-theMix.sh

6. When experiments start and Replayer asks for number of the number of test cases, type 200

7. When experiments finish, go to jfcguitar-v1.1.1, copy folder Demo which contains results of GUI testing
Appendix E  Integrating GUITAR and COBERTURA with Ant for GUI-testing on Boolean logic software using GUITAR 1.2

GUITAR V1.2 uses an Ant [201] as a way to build the different components. In order to integrate it with COBERTURA we had to use Ant to build the COBERTURA files as well. Using Ant, COBERTURA coverage reports can be built using two different ways. The first one is by collecting results of coverage in XML files and then use those XML files to generate coverage files. The second way is to store the measurement in SER file and then build the report using that SER file. The first one works fine with JUnit test. When using JUnit, there is an option of storing coverage measurements in XML files. However, GUITAR test are not JUnit tests. GUITAR generates test cases in XML format. Thus, we were unable to benefit from option provided by JUnit library to store information in XML files. When returning the first method of generating SER file. We updated the build file used in the sample example to overcome a bug there that always generated a problem in running COBERTURA even with JUnit tests. Then, we updated the XML build files and properties files in order to collect measurement during replaying and storing those measurements in SER files. The structure of folded has been changed.

The steps of GUI testing using the Ant are:

1. Open the following sequence of folders: Cobertura-Ant, cobertura 2.1, examples-booleanLogic, basic [212]

2. Perform configuration settings which are unavailable in the running script files as ANT technology has been used for building. Alternatively, configuration is done
through three other configuration files in the basic folder that contains main files of
GUITAR and all have the extension properties. Fix the configuration in each of:
jfcripper.properties, TestCaseGenerator.properties (default values of maximum
number of test cases and length of test case have been changed), and finally
jfcreplayer.properties. TestCaseConvertor component does not need any properties
file.

3. Delete previous execution files of results: Project.GUI.xml, Project.EFG.xml and
   Project.log

4. Remove the content of folders: TC, Projectsta (the name of this folder has been
   changed as the default name does not work because of a bug in GUITAR)

5. Run script: runAll.sh

6. When experiments start and Replayer never asks for number of test cases as we
   changed the configuration of jfcrepalyer.properties to go over all of test cases- tabs
   are not triggered but we found that it still be valid

7. When experiments finish, go to basic folder and copy the folders: TC, Projectsta and
   files: Project.GUI.XML, Project.EFG.xml and Project.log as all contain the results of
   execution
Appendix F  Steps for measuring coverage when using
Replayer v.1.1

1. Go through the sequence of folders: Cobertura-Ant, then TheMix [212]

2. Run the script: runCoberturaOnMyExample-EnvVar-TheMix-guitar.sh

3. When Replayer asks about number of test cases, enter: 200

4. Replayer starts, tabs stat being triggered, random values will be generated and used as
an input to SUT

5. When replaying phase finish, go to folder: CoberturaAnt, theMix, copy folder named
results that contains results of coverage
Appendix G  Measuring coverage using GUITAR 1.2 and COBERTURA 2.1

Steps to measure coverage

1. Go through the sequence of folders: Cobertura-Ant [212], cobertura-2.1, examples-booleanlogic, basic

2. Type ant, this will create a set of folders and files: classes and instrumented. If the folders are there, no need for this command

3. Remove all files inside report folder but keep structure of folders.

4. Run script runReplayerAnt.sh (Replyaer never asks about number of test cases you need to replay as we configured jfcreplayerCobertura.properties to replay 81 test cases)

5. When replaying finish, go to report folder and copy coverage results are being collected in basic_coverage.ser or cobertura-html folder
Appendix H  Clicking the push button on the SUT in GUI testing

GUITAR did not cover Control classes in Boolean logic SUT case study though the constructors of those classes are available in the GUI classes. We tried to check if the code of TruthTableComponent class which is GUI class calls TruthTableControl first-which is Control class- or calls other classes from the parser package. We found that there is no import statements for parser package in the TruthTableComponent class. Thus, TruthTableComponent calls TruthTableControl and TruthTableControl calls the parser classes. Consequently, we believe that there is processing happing on Control classes at run-time but even though it was not covered by GUITAR. GUI testing covered classes (MainExeGUI, DNFComponent, MainExeFrame, TruthTableComponent, VariableNegationComponent) and system testing covered all except non-GUI classes. Table 33 and Table 34 show the results of this investigation of GUI testing using simple case study vs. GUI testing using our Boolean logic case study.

**Table 33. Observations with a simple GUI app, test case length = 3, button does not get enabled till text is entered in text field**

<table>
<thead>
<tr>
<th>Tool</th>
<th>Clicking push button</th>
<th>Clicking radio button</th>
<th>Generating text for text field</th>
<th>Copying text from text field to label</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ripper</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Ripper</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>No</th>
<th>Try</th>
<th>Tool</th>
<th>Clicking push button</th>
<th>Clicking radio button</th>
<th>Generating text for text field</th>
<th>Generating truth table</th>
<th>Coverage information</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Default case: compute button does not get enabled till text is entered in text field and no default text entered in text field</td>
<td>Ripper</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Only GUI classes</td>
</tr>
<tr>
<td></td>
<td>Ripper</td>
<td>No (here it differs from simple GUI App)</td>
<td>Yes</td>
<td>Yes</td>
<td>No (here it is different from simple GUI app- we don't think this is related to input)</td>
<td>Only GUI classes (proves that it is not related to input as no coverage for Control or parser classes)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>We edited all test cases and entered valid input in test cases XML files as Dr. Memon explained</td>
<td>Replayer</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Low coverage for parser and Control classes</td>
</tr>
<tr>
<td></td>
<td>Exactly like Try 1, Replayer does not click the button regardless if valid input entered or not the text field. I updated 200 test cases but the button was not clicked.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>We interfered in the work of Replayer and entered text and clicked on compute button</td>
<td>Replayer</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>?? (not measured)</td>
</tr>
<tr>
<td></td>
<td>Low coverage for parser and Control classes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>We interfered with work of Ripper but not Replayer in the same way of try 3</td>
<td>Ripper</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>?? (not measured)</td>
</tr>
<tr>
<td></td>
<td>Low coverage for parser and Control classes (prove the difference from Try 1 as here we have coverage for Control and parser classes though input is invalid)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>We updated source code to have Boolean expressions in the text field and enabled all buttons</td>
<td>Ripper</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Covers Control and parser classes with different percentages depends on the used Boolean expression and max achieved is 51 %</td>
</tr>
</tbody>
</table>
Appendix I  Automated GUI testing using GUITAR - lessons learnt

Automated GUI testing requires the use of testing tools and integration of those tools with other tools for measuring coverage. GUITAR is a well-known GUI testing tool that automates the generation of test cases and executing them [8]. However, based on our experience, GUITAR shows deficiency in doing a complete GUI testing. Specifically, Replayer did not click push compute button in our Software Under Test (SUT) with several tabs. So we got coverage for only 6 classes out of 33. In this appendix, we go over potential solutions like: Passing test input in the source code of SUT, changing the default configuration settings of GUITAR, and change the GUI of SUT. We show how changing the GUI of SUT is the only effective solution to overcome that deficiency and how the use SUT equipped using Entity-Control-Boundary design principle facilitates the completion of GUI testing when GUITAR did not handle the GUI well.

Description of the case study

Our SUT performs some computation on Boolean operations through three main functionalities: computing and displaying the truth table of a Boolean expression, computing and displaying the Disjunctive Normal Form (DNF) of a Boolean expression, and deriving tests according to the variable negation strategy [186] for a Boolean expression. Table 35 shows the class content of each package of the SUT.

The overall architecture of the SUT followed the Entity-Control-Boundary (ECB) design principle which divides classes over three main categories [10]: Entity classes
represent the information the software needs to manipulate; Control classes realize the use cases, implement the logic of the software; Boundary classes realize the interactions between the software and the actors (e.g., Human, hardware, other software). Gomma refined the ECB taxonomy for the special case of distributed and real time systems [4]: e.g., Boundary classes are either one of device interface classes (further refined in other types), user interface classes, and system interface classes.

**Table 35. Description of source code of the case study**

<table>
<thead>
<tr>
<th>Package</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean Variables</td>
<td>BooleanVariable, SetOfBooleanVariables</td>
</tr>
<tr>
<td>Controls</td>
<td>CheckTestCase, Cube, DNFControl, TruthTableControl, VariableNegationControl</td>
</tr>
<tr>
<td>Gui</td>
<td>DNFComponent, MainExeFrame, TruthTableComponent, VariableNegation_TableModel, VariableNegationComponent, VariableNegationResultsTable</td>
</tr>
<tr>
<td>Parser1</td>
<td>LogicalExpressionParser, LogicalExpressionParserConstants, LogicalExpressionParserTokenManager, ParseException, SimpleCharStream, Token, TokenMgrError</td>
</tr>
<tr>
<td>Parser2</td>
<td>BinaryExpressionSolver, BinaryExpressionSolverConstants, BinaryExpressionSolverException, BinaryExpressionSolverTokenManager, SimpleCharStream, Token, TokenMgrError</td>
</tr>
<tr>
<td>Default</td>
<td>MainExeGUI</td>
</tr>
</tbody>
</table>

Our case study has one main Frame with three tabs. The captions of these tabs are: Truth table, DNF Expression, and variable negation. Clicking on any of these tabs will show a panel to handle the corresponding functionality.

The truth table tab has one text field, three Radio button(s) and one Push button as shown in Figure 27. The text field accepts the Boolean expression for which a truth table needs to be computed and displayed. Radio button(s) controls the format of the truth table content for the Boolean expression: Using either (0,1), (false, true) or (f,t). When the user enters a Boolean expression and clicks the Compute Push button, the truth table of that Boolean expression will be shown in the text area under the buttons. If the user does not
enter a Boolean expression in the input text field, an error message appears in a different pop-up window that contains a text showing the error message (typically a parsing error).

![Figure 27. Truth Table Tab of Boolean Logic Application](image)

**Figure 27. Truth Table Tab of Boolean Logic Application**

The DNF Expression tab has one text field, one text area and one Push button:

![Figure 28. DNF Expression Tab of Boolean Logic Application](image)

**Figure 28. DNF Expression Tab of Boolean Logic Application**

When the user enters a Boolean expression in the upper text field, clicks the Compute Push button, the program shows the DNF of the expression on a text area as a series of terms. If there is a problem in the format of the input (e.g., not a recognizable Boolean expression), the SUT's parser triggers a window that contains the error message.
This window is an interface for an implementation of the variable negation testing technique [186]. The input must be provided as a series of terms of the DNF of a Boolean expression. The user must enter those terms in the input text field(s), one term per text field: Figure 29. If the three default text fields are not enough (i.e., the DNF has more than three terms), the user clicks the AddProductTerm button to add a new text field. The user then can click the Compute Push button and the results appear in the output text area. The results include: Cube variants, unique true points and near false points. The user can press on the Show Text Cases Push button, and the program shows a table with a set of entries satisfying the test objective of the variable negation testing technique.

Integration of GUITAR and COBERTURA

In order to do GUI testing, we assembled a framework to facilitate our work by integration of two open source tools GUITAR [8] and COBERTURA [187]. We chose GUITAR as it is a well-known automated GUI testing tool that provides a higher level of automation than simpler capture/replay tools. In capture/replay the values of input, window coordinates, caption and others are hardcoded in test script which makes editing
these scripts as challenging task since these values may change in the SUT and this leads to the test script to fail during test execution [95]. Furthermore, this is still naïve approach and most of the time it is necessary to register the coordinates of screen when dealing with custom controls [96] and not automated as having capture/replay tool is not enough to say "automated testing" [97]. Though we noticed that replaying a test suite generated by GUITAR on Fedora OS generates different results when doing that on Ubuntu, still GUITAR is superior over other testing tools. Using GUITAR, GUI testing can be done using four components: Ripper, Convertor, Generator and Replayer [3].

Ripper interacts with GUI widgets when triggering events for the purpose of extraction of structural information related to GUI. Editable GUI widgets like text fields may have several attached events like text change. Events related to buttons such as clicking are also handled. Ripper starts from the containers like panels or frames and then explores widgets on top of them. The Ripper output is an XML file represents the GUI as tree hierarchy [3].

Convertor takes the output of Ripper and generates another XML files describes the Event Flow Graph (EFG) [3].

Generator uses the XML file generated by Convertor and produces a test suite consists of a specific number of test cases determined by the user. Each of those test cases is a sequence of events in the EFG with a specific length determined by user as well [3].

Replayer runs test cases on the SUT one by one with specific delay between two consecutive runs. For each test case, Replayer produces a state XML file that describes
the GUI after replaying that test case [3]. In order to determine whether a test case is passed, the state file is compared by another one from previous run.

We have also three stages for measuring coverage starting with instrumenting and ending with Replaying. Measuring coverage starts with instrumenting the code. Then, there is a need to bind the Replayer with the coverage files so it stores the coverage frequencies in that file. The next process is to replay test cases. The generated coverage file after replaying can be used by the reporting process to generate a coverage report.

**Measuring coverage using Replayer v.1.1 and COBERTURA**

For conducting GUI-testing on Boolean logic software, a set of steps should be considered: (1). In a Linux platform, download the software package for integration [212]; (2). Open the following sequence of folders: Cobertura-Ant, jfcguitar-v1.1.1; (3). Do the configuration by filling configuration options that are available in the execution script file jfc-sample-work-theMix.sh which overrides the configuration file available in SUT folder (named jfc-aut); (4). Remove the folder Demo that contains results of all previous experiments; (5). Run the shell script: jfc-sample-work-theMix.sh; (6) When experiments start and Replayer asks for number of the number of test cases, type 200; (7). When experiments finish, go to jfcguitar-v1.1.1, copy folder Demo which contains results of GUI testing. The results will be: Output folder: Demo, # logs: 199, # test cases: 200, # states: 199, windows: 1, GUI file: ./Demo/Demo.GUI, EFG file: ./Demo/Demo.EFG, log file: ./Demo/Demo.log, Ripping Elapsed: 5:19, Total time (Ripping, converting to EFG, test case generation, and replaying): less than 30 minutes- replayer took the majority of time.
For measuring coverage, a set of steps can be considered: (1). Go through the sequence of folders: Cobertura-Ant, then TheMix; (2). Run the script: runCoberturaOnMyExample-EnvVar-TheMix-guitar.sh; (3). Replayer starts, tabs stat being triggered, random values will be generated and used as an input to SUT; (4). When experiments start and Replayer asks for number of the number of test cases, type 200; (5). When replaying phase finish, go to folder: CoberturaAnt, theMix, copy folder named results that contains results of coverage.

The results will be in coverage report that will appear to show code and branch coverage will be generated. In our case, we got 0.77 for line coverage and 0.4 for branch coverage. We found that only 6 classes were covered.

**Measuring coverage using Replayer v.1.2 and COBERTURA**

The steps of GUI-testing on Boolean logic software will be: (1). Open the following sequence of folders: Cobertura-Ant, cobertura 2.1, examples-booleanLogic, basic [212]; (2). Perform configuration settings which are unavailable in the running script files as ANT technology has been used for building. Alternatively, configuration is done through three other configuration files in the basic folder that contains main files of GUITAR and all have the extension properties. Fix the configuration in each of: jfcripper.properties, TestCaseGenerator.properties (default values of maximum number of test cases and length of test case have been changed), and finally jfcreplayer.properties. TestCaseConvertor component does not need any properties file; (3) Delete previous execution files of results: Project.GUI.xml, Project.EFG.xml and Project.log; (4). Remove the content of folders: TC, Projectsta (the name of this folder has been changed as the default name does not work because of a bug in GUITAR); (5). Run script:
runAll.sh; (6). When experiments start and Replayer never asks for number of test cases as we changed the configuration of jfcrepalyer.properties to go over all of test cases- tabs are not triggered but we found that it still be valid; (7). When experiments finish, go to basic folder and copy the folders: TC, Projectsta and files: Project.GUI.XML, Project.EFG.xml and Project.log as all contain the results of execution

Here are some results related to GUI-testing of Boolean logic software: Output folders and files: TC, Proejectsta, Project.EFG.xml, and Project.log, # logs: 1, # test cases: 81, # of states: 81, # windows: 1, GUI file: ./Project.GUI.xml, GUI file: Project.EFG.xml, total time Elapsed: 4:25.

For measuring coverage using GUITAR 1.2 and COBERTURA 2.1, a set of steps should be followed: (1). Go through the sequence of folders: Cobertura-Ant, cobertura-2.1, examples-booleanlogic, basic; (2). Type ant, this will create a set of folders and files: classes and instrumented. If the folders are there, no need for this command; (3). Remove all files inside report folder but keep structure of folders; (4). Run script runReplayerAnt.sh (Replyaer never asks about number of test cases you need to replay as we configured jfcreplayerCobertura.properties to replay 81 test cases); (5). When replaying finish, go to report folder and copy coverage results are being collected in basic_coverage.ser or cobertura-html folder. In our case study, we got the same results we obtained in A.

**Manual GUI testing**

However, we expected coverage to be better than that but we did not know how much better it should be. So, we went through manual GUI testing with valid/invalid
input. When practicing the GUI with valid input, we got 21 classes covered with 59% coverage. When practicing the GUI with invalid input, we got 21 classes covered and 57% coverage. So probably GUI testing using GUITAR should reach coverage on the same rank.

**Enabling buttons and passing input in the source code of SUT**

Passing input at the source code of SUT which is a set of steps that can be done to modify the SUT that help GUITAR. In order to pass input to GUITAR at run time, we checked out GUITAR trunk using the terminal command:

```
```

It takes around 3 minutes to extract the trunk from repository. The current revision of GUITAR that was checked out is the revision number 3643 and last time was updated on 30th of April 2015. However, authors of GUITAR - currently active author on level of code is Ishan Banerjee - didn't separate jfc-guitar that we are interested in from other GUITAR versions. Thus, when we download the GUITAR package, it downloads components like WebGuitarSetup, WebGuitarTesting and others. The code is all mixed up with each other and it is difficult to separate the packages from each other. Running the ant command - as the instructions available in the download mentioned - did some processing at the background but we could not recognize the identity of this processing. No available information in the downloaded package on how to run GUITAR. Thus, we used trial and error way. We found a shell script called init-setup.sh. When we run it did some processing but no information is available about what is going on the terminal window when running that shell scripts. However, it generated a folder called dist inside
the distribution and placed in that folder another folder called guitar that contains a copy very similar to jfc-guitar v1.1.1 that we tried before but the folder contains a set of Python files. We believe that those are the files that authors used as an Oracle to test whether test case is passed. They can be also those file which convert XML-test cases used by GUITAR to JFCUnit test cases that is runnable from Java immediately. The code is large as there are around 1000 Java files and the size is around 5 MB. No reading from file statements in them. We do not think authors implemented a way to enter data to Replayer at run time as ReplayTC class does not include any reading from file statements, however this not significant that input can provided through the test cases.

We present first the results of coverage when providing the input to GUITAR in test cases, then we describe how we obtain those results. The results were: # covered classes: 30, line coverage: 63%, 1142/1787l; and branch coverage: 52%, 458/873. The input that was used in this experiment is like the following: For truth table "x AND (y OR Z)", for DNF: "(a or b) and c", for variable negation: The following terms were used: "x and not y and z", "y and w", and "not x and w". The problem that we faced with Boolean logic software is that when doing GUI testing without interfering with the work of GUITAR, truth table was not generated. The reason for that is GUITAR didn't click on the compute button. In other words, in the GUI application we are testing, we have a "compute" button that only becomes enabled (clickable) when a string is provided in a text field. Ripper does recognize there is such a button (i.e., we can see it in the generated tree) but does not click on it. We suspected this is because the button remains disabled because Ripper does not provide any string to the text field. We had a set of questions: Is this normal? Is there any way we can tell Ripper to use specific strings? We changed the
code of the GUI to enable the compute button by default. Consequently, clicking compute button without entering any text in the related text field generates pop-up window showing parsing error: "Encountered EOF... Was expecting NOT ...". We run Ripper, then it generated the same pop-up window with the same error message while ripping. We run TC generator, then it created 199 test cases (We got almost the same number of test cases when compute button was disabled). We run Replayer on only one specific test case (The one which generates the error), we got a coverage for each of parser and Control packages. The feedback that we got from authors of GUITAR says that: (1) It is normal that GUI Ripper does not click on a disabled button (2) About text input used in the Ripper: The default input is defined in the gui-model-jfc project.

In class edu.umd.cs.guitar.event.JFCEditableTextHandler, it is a hard-coded parameter in current distributed version. GUITAR's authors mentioned that they do not think the current version will support configurable input for the Ripper. Based on that we did the following investigation: We updated Boolean logic software to generate a text file compute.txt when compute button is clicked and radio.txt if a radio button is clicked. We found the ripper to click radio button but not the compute button. The reason is that compute button is disabled till a text is entered in text field. As explained in before, We enabled compute button and found coverage for parser and Control. Later, we added a third modification to Boolean logic software: (1). We added a listener to the text field in such way that the listener generates a document called text field.txt in case the text is changed and copy the content of the text field to the generated file. In case the text changed again, the listener overrides the file and updates the new file with a new content of text field; (2). We run the Ripper, then both files compute.txt and radio.txt were
generated but not text field.txt; (3). We deleted compute.txt and radio.txt after finishing Ripping phase; (4). We converted the GUI tree and generated an EFG; (5). We generated 200 test cases; (6). We run Replayer, then all three text files were generated; (7). We had a look on the Java file JFCEventHandler that GUITAR's authors described, it is a class inside the package event. There is no evidence that Ripper will use this class; (8). We run GUITAR on the example included in GUITAR package, but we couldn't see Ripper adds any text in the text field. The text in text field is generated by the source code of example itself when clicking radio button; (9). We run GUITAR on the simple example that we developed (The case study is one JFrame with two text areas and copy Push button) to understand GUITAR. Clicking the button moves the selected text in the text of the text area 1 to text area 2. We didn't see Ripper generating the hardcoded text while ripping. However, we saw the hardcoded text being added to the text area during Replaying.

The conclusion that we got is Ripper does not use any text for editable widgets during ripping. Replayer does that. Our initiative was that we don't think that it is a bug in Ripper when it didn't generate a text in the text field. The reason is that even if adding text to text field is covered by Memon's team, some other issues will appear. For example, in our simple SUT that we developed, even if there is a text in text area 1, clicking the button will not move that text to text area 2 unless the text is selected (highlighted). We believe it is hard to cover all options. To this end we run a second investigation. We thought it would be better that we conduct the experiments using the same approach that Memon et al. followed in their experiments [3]. For their SUT, as noticed, they pass input to SUT at ripping by the SUT itself. Specifically, in their SUT, when Ripper goes over different options in the drop down menu, each of them add input
automatically to the text field. So, we did the following: We changed code of Boolean logic software to have default text in all text fields. Then, we enabled all buttons. We conducted GUI testing without any interference in the process neither in the work of Ripper nor Replayer. We collected coverage for Replaying only. We got 42% in general and 16 classes were covered. One may raise question: Why did we enable the buttons since we already assigned a default text to all text fields? The answer is: Keeping the buttons disabled and assigning default text to text fields will not accomplish anything. The reason is: our SUT has been programmed to enable the buttons only when text in text field changes. In other words, when one runs the SUT, even if text field contains a default text, the button will be still disabled. Hence, button will not be recognized by GUITAR's Ripper as GUITAR authors mentioned. So, the only way in order to have those buttons clicked during ripping is to enable them in source code. Then, in order to show truth table and not an error message, we use the `setText()` method to assign a default text to the text fields. Consequently, Ripper will click the button and shows truth table.

The same will be repeated for buttons and text fields in other tabs. Off course, Replayer may change our default text while replaying test cases by replacing it with Replayer's default random text that is hardcoded in GUITAR's source code. We measured the coverage with different Boolean expressions used as an input to the Boolean logic software. The source code of the Boolean logic software was updated in order to have an input in the text fields. All text fields have valid input by using the method `setText()` in the source code of the Boolean logic software. The source code were changed as well to enable all buttons. The difference among the following experiments is only the Boolean expression in the first tab which related to truth table. The input in all other tabs is the
same in all experiments. In these experiments, it is noticeable that when number of
classes covered is equals to 17, then ParserException class is being added. Here are some
examples of results-The results are ordered as Boolean expression used, number of
covered classes, line coverage and finally branch coverage: (1). Boolean expression: X,
17, 44%, 635/1413, 25%, 177/691; (2). Boolean expression: X and Y, 16, 45%,
614/1346, 26%, 172/657; (3). Boolean expression: X or Y, 16, 45%, 607/1346, 25%,
166/657; (4). Boolean expression: Not X, 17, 47%, 677/1413, 31%, 219/691; (5).
Boolean expression: X nor Y, 17, 48%, 684/1413, 32%, 224/691; (6). Boolean
expression: X and (Y or Z), 17, 49%, 704/1413, 32%, 225/691; (7). Boolean expression:
(X and Y) or (X and (not Z)), 17, 51%, 730/1413, 36%, 253/691; (8). Boolean
expression: (X or Y) and ((X and Z) or (X and (not Z))) and (X and Y) or Y, 16, 49%,
670/1346, 32%, 215/657; (9). Boolean expression: X xor Y, 16, 45%, 613/1346, 26%,
173/657.

**Results of coverage during Ripping**

In this experiment, we measure the code coverage while doing the ripping on the
Boolean logic application using GUITAR. Ripping in our context is a kind of reverse
engineering at to the GUI of SUT. In case of GUITAR, it is being done on byte code
without the need for source code of the SUT. The objective of Ripper is to build a
hierarchy of GUI like a tree by extracting those GUIs from the executable binary of SUT.
The procedure starts by taking the GUI as an input and extracting top level windows.
Then for all those top level windows, a ripping procedure will be applied recursively. The
recursive procedure goes over each window of the set of windows and extract all widgets
on that window and their properties. Then, it identifies which widgets are executables
from that set of windows. The reason for the previous step is that the recursive algorithm will execute all of those widgets and get all the windows invoked by them. Finally, the algorithm update the set of windows and call itself recursively on the new set of windows that consists of old set plus the new founded windows. The algorithm stops when there is no more additions to the set of windows. The coverage while ripping the Boolean logic application was: # covered classes: 6, line coverage: 70%, 217/309, branch coverage: 18%, 6/32. Table 36 shows detailed results.

<table>
<thead>
<tr>
<th>Classes in this Package</th>
<th>Line Coverage</th>
<th>Branch Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNFComponent</td>
<td>66%</td>
<td>0%</td>
</tr>
<tr>
<td>MainExeFrame</td>
<td>100%</td>
<td>N/A</td>
</tr>
<tr>
<td>MainExeGUI</td>
<td>83%</td>
<td>N/A</td>
</tr>
<tr>
<td>MainExeGUI$1</td>
<td>100%</td>
<td>N/A</td>
</tr>
<tr>
<td>TruthTableComponent</td>
<td>74%</td>
<td>50%</td>
</tr>
<tr>
<td>VariableNegationComponent</td>
<td>61%</td>
<td>6%</td>
</tr>
</tbody>
</table>

**Table 36. Coverage of all classes during ripping on the Boolean logic application**

**Trying with dummy case study**

After finishing the work with passing input in source code of SUT, we came to the following initiative: In order to confirm or not the finding, create a simply, dummy GUI application in Java with text fields, buttons, and radios. It does not need to have much logic in Control classes, just perhaps some reporting on the console so that we know what has executed. Then, to try Ripper on that simple software. If really experiments confirm our observation, then our though that we can ship this application to GUITARS's authors and ask them specifically whether we are doing something wrong. Based on that, we prepared simple GUI application. We tried to keep similar design of
our original case study because we wanted to understand the reason for this problem. The first dummy case study that we tried with simulates the first tab of our case study. It contains one tab that has one text field, two radio buttons, one Label and one Push button. The Push button does not get enabled till user enters a text in the text field. When clicking the Push button, the text is copied from text field and pasted in the Label. The color of copied text matches the selected radio button Red/Blue. For this specific case study, GUITAR did not show any problem.

In the details of that work, we went through the following: We reused parts of Boolean Logic software. The application consists of the following classes: (1). Main class: MainExeGUI; (2). GUI classes: MainExeFrame and CopyTextComponent; the application has a text field, a Push button, two radio buttons and a Label. The Push button is disabled till a text is entered in the text field. Clicking the Push button copies text from text field and pasts it in the Label colored by blue or red based on the selected radio button. The Push button gets disabled again as soon the text placed in the Label. Clicking on another radio button removes text from Label and enables the Push button again so user can click the button to copy text again and pasts it with the new color. CopyTextComponent class makes an instance of TextControl Control class in a case of: The text in text field is changed, Push button is pushed or radio button is clicked. Then, it calls one of three writing methods in the Control class based on the triggered event; (3). Control class: TextControl, it creates an instance of MyTextEntity Entity class as soon as TextControl's constructor is called. It has three printing methods each of them prints a message to console and calls a log method in MyTextEntity class; (4). Entity class: MyTextEntity, it has a log method that creates one of three files: a.
copy.txt, if Push button is clicked; b. radio.txt, if one of radio buttons is clicked; c. text field.txt, if text in text field changed and it adds the text entered in text field to those files in addition to information about triggered event.

When we run GUITAR, Ripper generated radio.txt but neither copy.txt nor text field. This result coincides with our previous results that Ripper does not generate any text for text fields. Later we decided to contact the authors with the following information explaining that when running Ripper, execution traces show that the radio button is clicked but neither the text in the text field is changed nor the Push button is clicked. We noticed that Ripper does not generate any text for text field. The app generates the radio.txt but neither the copy.txt nor the textFiled.txt.

We found that Replayer generates, for a text field, the hardcoded that GUITAR authors mentioned, but this is not the case of the Ripper. However, our intention is to use different strings in the text field at ripping as some test cases would not be generated unless a text is entered to the text field. Based on Memon et al. feedback, we checked the GUI tree and EFG. We found the following: The Ripper does not give a complete GUI file. The label widget is missing. As mentioned before clicking the button copies the text from the text field and pastes it in the Label with font's color based on the selected radio button. All other important components are there (i.e. Jbutton: W222917567, text field: W3743474262, radio buttons: W3736128464, W3734251194 and W4002222236). The EFG contains all events for generated widgets: Jbutton: e222917567, text field: e3743474262, radio buttons: e3736128464, e3734251194 and e4002222236. Of course, The EFG does not have an event for the label as it is missing in the GUI file. Thus, we confirmed a that test case which should have the following events is missing: Entering
text in text field, copy button gets enabled, user clicks on copy button, and then text is getting copied to the Label with the color selected with the radio button.

GUITAR authors replied that if we search for the text of Labels (e.g., “Enter text and click copy”), we will find they are actually titles of panels, and are included in the GUI file. However, “Enter text and click copy” is not a text of a Label. It is a text of panel. The output Label has nothing to do with this text. GUITAR author added that we are correct that they are not included in the EFG or test cases because a click event on a panel is not considered relevant in the implementation of GUITAR. However, we are not clicking on the panel itself but on the Push button on that panel. In further feedback, GUITAR author added that GUITAR will not use the default input in the Ripper.

The second dummy case study that we used consists of two tabs, the first tab is the same tab used in the first dummy case study. The second tab contains two text areas and one Push button. There is some default text in the first text area and it is editable by user. When user clicks the Push button, the selected part of text in the first text area is moved to the second text area. For this specific case study with two tabs, GUITAR had no problem as well. Similarly to first case study in which Replayer had no problem, for two simple SUTs each with 2 tabs, we found that with test case length = 3 or test case length = 10, Replayer clicked buttons in each tab (no problem).

Then we tried with a simple GUI application with 3-tabs, in which the third tabs coincides with the first tab. For this specific case study, Replayer did not click the copy button (in the first tab). We changed the contents of the tabs in the previous case study, then, GUITAR did not generate any problem. We returned back to modifying our Boolean logic case study by dropping the third tab to have only 2 tabs. For this Boolean
logic SUT with two tabs, Replayer clicked one compute button on one tab but not both of them in a single test run.

**Providing input to Replayer at runtime**

GUITAR authors recommended that when the interested component (compute button) is included in the GUI and EFG, then we may have test cases that encode the desired event sequence. They added, to be specific in a simple case study, the “Copy” button (w222917576) and event (e222917576) are included in the GUI and EFG. This event is included in two test cases though neither of them construct the “edit-copy” sequence. Hence we had two options: (1) The first option is to generate longer event sequences. When we use the flag “-l 3” to generate length-3 test suite, we have the following test cases: “t_e3765634550_e1510542628_e222917576.tst” (edit - click on main panel - copy). With such test cases, the Replayer will enter a default text and click copy. Of course, we can specify the text as well. If we want to add parameter to a test case, we will firstly need to find the widgets that are supported by this handler. For example, if we have such a widget from the GUI file: w4009000504, and corresponding event in EFG, e4009000504, then in the test case file, we can add a “Parameter” tag to specify the input during replay. Table 37 shows an updated test case.

**TABLE 37. ADDING DATA INPUT TO TEST CASE**

```xml
<TestCase>
  ...
  <Step>
    <EventId> e4009000504 </EventId>
    <ReachingStep>false</ReachingStep>
    <Parameter>TheTextToInputDuringReplay</Parameter>
  </Step>
  ...
</TestCase>
```
Based on previous lines obtained from GUITAR author, we did a couple of new experiments here: (A). We run Replayer again on simple GUI app, it clicked the button. Absolutely, Ripper does not do that. We were looking for a solution at the level of Ripper. However, since Replayer does that task, really it does not matter for us to understand how Memon et al. designed the Ripper so it does not click the button while Replayer does that; (B). We double checked our previous finding on Boolean logic software just to make sure. With test case length =3, Replayer will not click the disabled button in the Boolean logic software regardless of anything else. We enabled the button then: (1). With modified test cases, Replayer uses the text in XML files; (2). With original test cases, Replayer uses default random text. In both cases 1 and 2, Replayer clicked the button.

**Change The Configuration of GUITAR**

A set of steps that can be done on configuration of experiments leads Convertor to do more processing in GUI testing like increasing the length of generated test cases, changing the test case generation algorithm, or increasing the number of generated test cases. Because of existing work for Gao et al. [213] which suggests the change of configurations of GUITAR to have better testing results, we conducted a set of experiments with several configurations. Memon et al. studied the issue of making GUI testing repeatable by investigating possible relationship between a set of variables related to code coverage [3]. In their experiment, the independent variables include the operating system used as platform, initial state of GUI of software under test (SUT), GUITAR's configuration, GUITAR's waiting time between two replays of test cases and Java Development Kit (JDK) version. The dependant variables include code coverage and
oracle properties. In this appendix, we complement results obtained in aforementioned study by showing the deficiency of changing GUITAR's configuration to have a complete GUI testing using GUITAR.

**Apx_1. Increasing length of test case**

The resulting test suite of Generator contains one test case for each event sequence of length L (determined by user) in the input EFG. L = 1 results in a suite that covers every event in the EFG and L = 2 results in a suite that covers every pair of events occurring in the EFG, and so on for longer lengths. Each individual test case covers one sequence of events with length L. A prefix is applied to the sequence of events to make it accessible from the application’s initial state and hence the actual length of test cases may exceed the value of L [3]. Based on recommendation of GUITAR's authors, next, we tried changing GUITAR configuration by increasing the length of generated test cases. Default value is 3, but when we made length of tc= 5, no change of results (same issues faced). So, we made length of tc= 10, then, Ripper generated the GUI tree so Convertor generated the EFG, but Generator failed to generate test cases.

**Apx_2. Changing test case generation algorithm**

An EFG is a directed graph representing the event interactions on a GUI. Each node in an EFG represents a GUI event like clicking on compute button [3]. An edge from node v₁ to node v₂ represents a follows relationship between v₁ and v₂, indicating that event v₂ can be performed immediately after event v₁ [3]. An EFG is analogous to a control-flow graph, in which vertices represent program statements and edges represent execution flows between the statements [3]. In next step, we tried to change the test case
generation algorithm as we are interested in the scenario when Generator traverses the graph and comes to a node in which there are two options for consequent event (node). Which option (branch) the Generator will choose? And why it does that specifically that maximum length of the test case is fixed for example =3? If we run Generator several times, do we get the same test cases every time? If yes, do they have the same names? What is the objective function when it generates those test cases? What is the stopping criteria? We investigated this topic and found that the test case is created by selecting random path from an initial event that has no predecessor to any other event and hence different test suite can be created in each test run. By selecting an event of the EFG, the event forms the start of the test case. The algorithm works in the following way: It picks a fixed number n of events from the EFG. The events are selected by one of two ways: (a) Random strategy, it selects the events randomly and (b) Fixed-length strategy, it picks one event then all events on the path of length n-1 starting in this event. When using SequenceLength generator, the SequenceLength Generator provides a guarantees of event coverage in the EFG. However, using longer sequence lengths could discover more interesting bugs. The number of maximum possible test cases increases exponentially with an increase in the length of test cases. Therefore, a tester might consider random sampling as an alternate method for test case generation. There are default settings in the distribution package GUITAR. For the algorithm type, we were able to find those settings. For objective function (plugin), we didn’t find any settings. Changing the test case generation algorithm for which default is sequence length coverage generated thousands of test cases. Replaying is a time and computation consuming task. When replaying 25 % of those test cases, no change in results (i.e. coverage).
Apx_3. Increase number of test cases

We thought in increasing the number of generated test cases for which the default value=200. However, GUITAR generates useful test suite [3]. In other words, test cases of test suite are equally distributed over the sample space of test cases. So, increasing the number of test cases over the default value is not promising. Using the non-default values of test case length or EFG traversal algorithm didn’t work well before.

Apx_4. Visualization of EFG

Because of the control mechanism that we use in our GUI is represented as rare event in EFG (enter text, button get enabled, and click button), we wanted to know what happens when generator comes to a node and there are two different edges going out of that node. In other words, which edge the generator will select? We tried to use visualization for each of our Boolean logic case study and simple case study. The question was: Does visualizing EFG helps to understand the graph traversal process? When trying to visualize some EFGs they were complicated. We obtained EFG of simple GUI with 3-tabs and 2-tabs which work well without problem with GUITAR and a third one for Boolean logic software. We noticed that decreasing number of tabs simplifies the EFG massively, however, in all obtained EFGs, it was impossible to follow edges due to large number of nodes. We used software called Gephi [104] to obtain the graph after executing a script in GUITAR package to convert from XML file of EFG to DL extension which is readable by Gephi. However, we had to update each of the script and configuration file of Gephi to obtain the graph.
The second option suggested by GUITAR authors is to generate desired event sequences by use the OME* method [214] through an iterative process: When the Ripper recognizes that “Copy” will be enabled in one execution where the text is edited, then it will continue ripping based on this specific observation to explore more event spaces. However, we have not seen any necessity as we can either use text in case study itself or enable button and update the 200 test cases manually. In addition, Ripper execution traces sent to us by GUITAR authors were little different from ours. We were not sure that we are using the same version that use of GUITAR. Number of test cases is different as well.

**Change GUI of SUT**

A set of steps that enables us to reach our goal by completing the testing without helping GUITAR nor making GUITAR do more steps and takes in account that our objective is not to contribute to GUITAR. In this type of solutions, we changed the GUI of SUT. Since the SUT equipped with Entity-Control-Boundary design principle, we had no difficulty in changing the GUI. In other words, it was possible to change the GUI without changing the business logic.

In first try, we changed the GUI by putting all components in one interface but GUITAR did not work well with this GUI (same issues faced before). In the second try, we used menus in the GUI, but GUITAR did not click the Push buttons.

Figure 30 shows the Main window new GUI implementation with which GUITAR had no problem. Clicking on any Push button generates sub-window equivalent to one of the corresponding tabs.
Apx_5. Thanks to ECB design principle

The implementation of new GUI was done by adding three classes to the GUI package: TruthTableExeFrame, DNFExeFrame, and VariableNegationExeFrame. The class MainExeFrame was modified to make calls to the previous three GUI classes instead of making direct calls to existing classes: TruthTableComponent, DNFComponent, and VariableNegationComponent, respectively. Thus, the three new added classes extends the class JFrame and at the same time receive calls from DNFExeFrame and pass them to the three Component classes. In other words, they work as middle layer inside the GUI tier. Finally, the modification the class MainExeFrame were minor as well. Three internal classes were added to this class to handle the calls from this class to the three new added classes. The addition of these new internal classes minimize modifications the existing structure of the class MainExeFrame.

Apx_6. Updating test cases for modified SUT

GUITAR uses button caption as an identifier but there is no such one for text fields. So we used coordinates as identifiers in the EFG. The reason for this is that we need to add an input at run time for text fields. Here is an example, for first functionality text field (truth table): X=6, Y=26, Width=282, Height=25. For second functionality text
field (DNF component): X=6, Y=26, \textit{Width}=270, \textit{Height}=25. For third functionality (variable negation): X=6, \texttt{Y}=24, \textit{Width}=205, \textit{Height}=28. X=6, \texttt{Y}=52, \textit{Width}=205, \textit{Height}=28.X=6, \texttt{Y}=80, \textit{Width}=205, \textit{Height}=28. Underlined properties were used as identifier in the EFG.XML. X, Y coordinates in the EFG were different from those when running the case study a side. In a total number of test cases=200, we found first functionality text field: W2729604298 in 17 test cases, second functionality text field: W3012009522 in 11 test cases, third functionality text fields: W1756625574 in 41 test cases, W1756636920 in 44 test cases, W3631670506 in 44 test cases.

In summary, using the new GUI, the new results are better than previous ones in the following aspects: We got coverage for 30 classes out of 33 and 63 \% coverage of the code. In previous runs when we used the default settings we covered 6 classes and 70 \% of code. In previous runs with input hardcoded in the source code of case study we got 17 classes covered with 50 \% coverage of the code.

\textbf{Conclusion}

Based on our experience, testing GUI application using GUITAR was not a straight forward process. We had to change the SUT in order to have this process go smoothly. In this appendix, we reported our own experience in GUI testing using GUITAR. We showed obstacles and our proposed solutions to handle them.
Appendix J  The use of Marathon

Capture/replay testing attempts to reduce the cost of testing re-execution by capturing the interactions with the AUT during one test execution session and then replaying those interactions during later test execution session [215].

![Diagram of test process](image)

**Figure 31. Manual test process (left) and capture/replay testing process (right) [215]**

The right side of Figure 31 shows how this is different from manual test execution. The interaction with the system under test is managed by an interaction tool called a capture/replay tool. This records all the inputs sent to the system under test and the outputs that result (e.g., procedural return results, screen snapshots, data files, etc).
Then when a new component related to the AUT must be tested, the capture replay tool can attempt to return all the recorded tests and report which ones fail. To rerun each recorded test, the tool sends the recorded inputs to the AUT and then compares the new outputs with the recorded outputs from the original test execution. The main problem with capture/replay testing is that it is very fragile. For example, a change to the layout of a window in the AUT (such as changing from a combo box to a radio button) or a small change to the interface of one procedure in an Application Programming Interface (API) can cause a large number of the recorded tests to fail. These have to be tested manually again and recorded for future sessions. This inability to adapt to small changes in the AUT creates a huge maintenance problem with the recorded test and often leads to the capture/replay method being abandoned after several AUT releases. This problem comes from a lack of abstraction in the recorded tests. That is, it is the low-level details of the actual AUT input and output. The verification of the correctness of the outputs is also usually too low-level and based on comparing screen snapshots or strings rather than checking just the relevant higher level features of the output. The key issue of automating the test execution in only partially solved by the capture/replay approach. This is due to the extreme sensitivity of the approach to any changes in the AUT. This approach is usually used to automate the GUI testing of AUT which is only part of the desired functional test suit.

AJ.1 Application of experiment of Capture/Replay using Marathon software on our case study

The paragraph summarizes a set of tries on testing the boolean logic software using a software called Marathon. This section summarizes our try to run Marathon on the Boolean Logic Mix provided to me. In order to run Marathon, before start creating a
project, we collected the following information about application: Main Class Name: The name of the application class that contains the main method (MainExeGUI), Class Path: The class path for the application which includes folders and archives that provide to the Java command to start the application. We can write the jar file name in case we have jar file, however, we didn't use the jar file as we have source code. Directory: The directory in which we created Marathon project. Launcher: Default launcher script as Marthon uses Ruby or Paython (we used Paython). Object Lookup: Using object map provides the most flexible solution for creating tests. When this lookup strategy is used, Marathon records the generated component names and their associated recognition properties in text files. These object map files are formatted in AML format and can be edited with any text editor (including the Marathon's editor). Then, we made the following steps: we start Marathon by using the included script files. Marathon opened the Project Selection dialog shown in Figure 32.

![Figure 32. Marathon start window](image)

In the Project Selection dialog, we clicked on New button, the Project Settings dialog shown in Figure 33 was opened.
**Figure 33. Project settings**

**Figure 34. Marathon control center**
We filled the previous mentioned information about the project. For example, the Project Settings dialog is split across multiple tabs. In the language tab, we selected Python a Script from the drop down list. we didn't enter any values for the Program Arguments and VM Arguments fields. Finally, we clicked on Save button to save the project and close this window. We created an empty test script, by creating an empty test case using "File" then "New" then "Test Case" menu option. Marathon created an untitled test case and opens it in the editor. We recorded a test script using Marathon in-built recorder by clicking on the toolbar button to start the recording. Marathon main window is minimized and a floating Marathon Control Center window is displayed as shown in Figure 34. We perform some operations on the application and hence these operations are recorded by Marathon. We stopped recording after performing some operations by clicking on the button in the Control Center. Marathon close the application and returned to the Main window.

We saved the recorded script as our myTestCases.py folder of the project. Figure 35 shows the generated test case
Figure 35. Saved test case

```python
# {{ Marathon
from default import *
#}}{{ Marathon

def test():

    set_java_recorded_version("1.7.0_25")
    if window('Main Exe Frame'):
        select('TextField', 'True and False')
        click('Compute')
        select('false, true', 'true')
        click('Save Truth Table')
    close()

    pass
```

Figure 36. Playing the script

We played our individual script by selecting a script from the navigator and select a test case file and double click it. Marathon opened the file in a new editor. We clicked on the toolbar button. Marathon launched the application and replayed the actions
recorded in the script. In most cases the script plays fast (because the script does not wait for keyboard or mouse input from you). It is possible to reduce the speed of replay. Figure 36 shows the results. In order to play multiple scripts, we created another test case and used Test Runner view called JUnit and it is next to the Navigator view. We ran play all the scripts. Then, we generated a report as by clicking on report button as shown in Figure 37.

**Figure 37. The generated report**

Based on our reading for book [97] and the previous experiment we found that capture/replay is not automated way to do GUI testing. Moreover, the claims by many authors in literature to have a tool to conduct automated GUI-testing when just own capture/replay tool is not true.
## Appendix K Functional test suite description

### Table 38. Description of functional system logic test suite

<table>
<thead>
<tr>
<th>functionality</th>
<th>Test</th>
<th>Test input</th>
<th># variables</th>
<th>Expected results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Truth Table</td>
<td>1</td>
<td>x AND y OR Z</td>
<td>3</td>
<td>[0 1 0 1 0 1 1 1]</td>
</tr>
<tr>
<td>Truth Table</td>
<td>2</td>
<td>a and (b or F)</td>
<td>3</td>
<td>[0 1 0 1 0 1 1 1]</td>
</tr>
<tr>
<td>Truth Table</td>
<td>4</td>
<td>a xor b</td>
<td>2</td>
<td>[0 1 1 0]</td>
</tr>
<tr>
<td>Truth Table</td>
<td>5</td>
<td>a or not b</td>
<td>2</td>
<td>[1 0 1 1]</td>
</tr>
<tr>
<td>Truth Table</td>
<td>6</td>
<td>a xnor (b nand c)</td>
<td>3</td>
<td>[0 0 0 1 1 1 0]</td>
</tr>
<tr>
<td>Truth Table</td>
<td>7</td>
<td>a NOR b</td>
<td>2</td>
<td>[1 0 0 0]</td>
</tr>
<tr>
<td>Truth Table</td>
<td>8</td>
<td>(A AND d) XOR (NOT (C XNOR (NOT b)))</td>
<td>4</td>
<td>[0 1 1 0 0 1 1 0 1 0 0 1]</td>
</tr>
<tr>
<td>Truth Table</td>
<td>9</td>
<td>a and b</td>
<td>2</td>
<td>[1 0 0 0]</td>
</tr>
<tr>
<td>DNFControlTest</td>
<td>testSize</td>
<td>(a or b) and c</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>DNFControlTest</td>
<td>testGet1</td>
<td>(a or b) and c</td>
<td>-</td>
<td>&quot;NOT(a) AND b AND c&quot;,&quot;a AND NOT(b) AND c&quot;,&quot;a AND b AND c&quot;</td>
</tr>
<tr>
<td>DNFControlTest</td>
<td>testGet2</td>
<td>(a AND b) XOR c</td>
<td>-</td>
<td>&quot;NOT(a) AND b AND c&quot;,&quot;NOT(a) AND b AND c&quot;,&quot;a AND NOT(b) AND c&quot;,&quot;a AND b AND NOT(c)&quot;</td>
</tr>
<tr>
<td>VariableNegationControlTest</td>
<td>testVariableNegation1</td>
<td>&quot;a and b and not c&quot;,&quot;a and d&quot;</td>
<td>expectedCubes=</td>
<td>expectedTruePoints=</td>
</tr>
<tr>
<td>VariableNegationControlTest</td>
<td>testVariableNegation2</td>
<td>&quot;x and not y and z&quot;,&quot;y and w&quot;,&quot;not x and w&quot;,&quot;y and not z&quot;</td>
<td>expectedCubes=&quot;[1 0 1 -1&quot;,&quot;-1 1 -1&quot;,&quot;0 0 -1 -1 &quot;,&quot;-1 1 -1 &quot;; expectedTruePoints=&quot;[1 0 1 0 , 1 0 1 1 ]&quot;,&quot;[1 1 1 0 ]&quot;,&quot;[1 1 0 0 , 1 0 1 0 , 1 1 1 0 ]&quot;); expectedAllVariables=&quot;x&quot;,&quot;y&quot;,&quot;z&quot;,&quot;w&quot;; expectedNearFalsePoints=&quot;([0 0 1 0 ]&quot;,&quot;[1 0 1 0 ]&quot;); expectedAllVariables=&quot;x&quot;,&quot;y&quot;,&quot;z&quot;,&quot;w&quot;;</td>
<td></td>
</tr>
</tbody>
</table>
| VariableNegationControlTest    | testPrepareArrayResult | "x and not y and z","y and w" | expectedCubes="[1 0 1 -1","-1 1 -1","0 0 -1 -1 ","-1 1 -1 "; expectedTruePoints="[1 0 1 0 , 1 0 1 1 ]","[1 1 1 0 ]","[1 1 0 0 , 1 0 1 0 , 1 1 1 0 ]"); expectedAllVariables="x","y","z","w"; expectedNearFalsePoints="([0 0 1 0 ]","[1 0 1 0 ]"); expectedAllVariables="x","y","z","w"; | expLabelsCaption = "[(1) = UTP for \"x and not(y) and z\""," + "(2) = NFP for \"x and not(y) and z\" negating \"x\""," + "(3) = NFP for \"x and not(y) and z\" negating \"y\""," + "(4) = NFP for \"x and not(y) and z\" negating \"z\""," + "(5) = UTP for \"x and not(y) and z\" negating \"x\"," + "(6) = UTP for \"x and not(y) and z\" negating \"y\"," + "(7) = UTP for \"x and not(y) and z\" negating \"z\"," + "(8) = UTP for \"x and not(y) and z\" negating \"x\"," + "(9) = UTP for \"x and not(y) and z\" negating \"y\"," + "(10) = UTP for \"x and not(y) and z\" negating \"z\"," + "(11) = UTP for \"x and not(y) and z\" negating \"x\"," + "(12) = UTP for \"x and not(y) and z\" negating \"y\"," + "(13) = UTP for \"x and not(y) and z\" negating \"z\"," + "(14) = UTP for \"x and not(y) and z\" negating \"x\"," + "(15) = UTP for \"x and not(y) and z\" negating \"y\"," + "(16) = UTP for \"x and not(y) and z\" negating \"z\"," + "(17) = UTP for \"x and not(y) and z\" negating \"x\"," + "(18) = UTP for \"x and not(y) and z\" negating \"y\"," + "(19) = UTP for \"x and not(y) and z\" negating \"z\"," + "(20) = UTP for \"x and not(y) and z\" negating \"x\"," + "(21) = UTP for \"x and not(y) and z\" negating \"y\"," + "(22) = UTP for \"x and not(y) and z\" negating \"z\"," + "(23) = UTP for \"x and not(y) and z\" negating \"x\"," + "(24) = UTP for \"x and not(y) and z\" negating \"y\"," + "(25) = UTP for \"x and not(y) and z\" negating \"z\"," + "(26) = UTP for \"x and not(y) and z\" negating \"x\"," + "(27) = UTP for \"x and not(y) and z\" negating \"y\"," + "(28) = UTP for \"x and not(y) and z\" negating \"z\"," + "(29) = UTP for \"x and not(y) and z\" negating \"x\"," + "(30) = UTP for \"x and not(y) and z\" negating \"y\"
\( y \) and \( w \), \( y \) + \( w \) = NFP for \( y \) and \( w \) negating \( y \) \( w \);

expVariableName="(1),(2),(3),(4),(5),(6),(7)", "Test Case", "Your Test Case";
expCellValue=

<table>
<thead>
<tr>
<th>0 0 0 0</th>
<th>0 0 0 1</th>
<th>0 0 1 0</th>
<th>0 0 1 1</th>
<th>0 1 0 0</th>
<th>0 1 0 1</th>
<th>0 1 1 0</th>
<th>0 1 1 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td>0 0 0 1</td>
<td>0 0 1 0</td>
<td>0 0 1 1</td>
<td>0 1 0 0</td>
<td>0 1 0 1</td>
<td>0 1 1 0</td>
<td>0 1 1 1</td>
</tr>
<tr>
<td>(0 0)</td>
<td>(0 1)</td>
<td>(1 0)</td>
<td>(1 1)</td>
<td>(0 0)</td>
<td>(0 1)</td>
<td>(1 0)</td>
<td>(1 1)</td>
</tr>
<tr>
<td>(0 0 0 0)</td>
<td>(0 0 0 1)</td>
<td>(0 0 1 0)</td>
<td>(0 0 1 1)</td>
<td>(0 1 0 0)</td>
<td>(0 1 0 1)</td>
<td>(0 1 1 0)</td>
<td>(0 1 1 1)</td>
</tr>
</tbody>
</table>
## Appendix L  GUI testing experimental results

### TABLE 39. TRUTH TABLE COVERAGE INFORMATION

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BinaryExpressionSolver</td>
<td>x AND (y OR Z)</td>
<td>0.27</td>
<td>0.24</td>
<td>0.23</td>
<td>0.21</td>
<td>0.27</td>
<td>0.24</td>
<td>0.27</td>
<td>0.24</td>
<td>0.23</td>
<td>0.21</td>
<td>0.31</td>
<td>0.26</td>
<td>0.23</td>
<td>0.21</td>
<td>0.26</td>
<td>0.23</td>
<td>0.26</td>
<td>0.23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>BinaryExpressionSolverTokenManager</td>
<td>a XOR b</td>
<td>0.37</td>
<td>0.18</td>
<td>0.34</td>
<td>0.15</td>
<td>0.40</td>
<td>0.21</td>
<td>0.41</td>
<td>0.23</td>
<td>0.35</td>
<td>0.17</td>
<td>0.49</td>
<td>0.34</td>
<td>0.34</td>
<td>0.15</td>
<td>0.39</td>
<td>0.20</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>BooleanVariable</td>
<td>a OR (NOT b)</td>
<td>0.80</td>
<td>N/A</td>
<td>0.80</td>
<td>N/A</td>
<td>0.80</td>
<td>N/A</td>
<td>0.80</td>
<td>N/A</td>
<td>0.80</td>
<td>N/A</td>
<td>0.80</td>
<td>N/A</td>
<td>0.80</td>
<td>N/A</td>
<td>0.80</td>
<td>N/A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Cube</td>
<td>a XNOR (b NAND R)</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>DNFControl</td>
<td>a NOR b</td>
<td>0.80</td>
<td>0.93</td>
<td>0.93</td>
<td>0.77</td>
<td>0.93</td>
<td>0.77</td>
<td>0.93</td>
<td>0.77</td>
<td>0.93</td>
<td>0.77</td>
<td>0.93</td>
<td>0.77</td>
<td>0.93</td>
<td>0.77</td>
<td>0.93</td>
<td>0.77</td>
<td>0.93</td>
<td>0.77</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>LogicalExpressionParser</td>
<td>x AND (y OR Z)</td>
<td>0.51</td>
<td>0.55</td>
<td>0.48</td>
<td>0.53</td>
<td>0.54</td>
<td>0.57</td>
<td>0.51</td>
<td>0.55</td>
<td>0.48</td>
<td>0.53</td>
<td>0.54</td>
<td>0.57</td>
<td>0.48</td>
<td>0.53</td>
<td>0.51</td>
<td>0.55</td>
<td>0.51</td>
<td>0.55</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>LogicalExpressionParserTokenManager</td>
<td>a XOR b</td>
<td>0.52</td>
<td>0.34</td>
<td>0.49</td>
<td>0.31</td>
<td>0.53</td>
<td>0.37</td>
<td>0.56</td>
<td>0.41</td>
<td>0.50</td>
<td>0.34</td>
<td>0.60</td>
<td>0.50</td>
<td>0.49</td>
<td>0.31</td>
<td>0.53</td>
<td>0.37</td>
<td>0.53</td>
<td>0.37</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>SetOfBooleanVariables</td>
<td>a XOR b</td>
<td>0.93</td>
<td>0.77</td>
<td>0.93</td>
<td>0.77</td>
<td>0.93</td>
<td>0.77</td>
<td>0.93</td>
<td>0.77</td>
<td>0.93</td>
<td>0.77</td>
<td>0.93</td>
<td>0.77</td>
<td>0.93</td>
<td>0.77</td>
<td>0.93</td>
<td>0.77</td>
<td>0.93</td>
<td>0.77</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>SimpleCharStream</td>
<td>a AND (y OR Z)</td>
<td>0.30</td>
<td>0.23</td>
<td>0.30</td>
<td>0.25</td>
<td>0.30</td>
<td>0.23</td>
<td>0.30</td>
<td>0.23</td>
<td>0.30</td>
<td>0.25</td>
<td>0.30</td>
<td>0.23</td>
<td>0.30</td>
<td>0.25</td>
<td>0.30</td>
<td>0.25</td>
<td>0.30</td>
<td>0.24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Token</td>
<td>a XOR b</td>
<td>0.75</td>
<td>1.00</td>
<td>0.75</td>
<td>1.00</td>
<td>0.75</td>
<td>1.00</td>
<td>0.75</td>
<td>1.00</td>
<td>0.75</td>
<td>1.00</td>
<td>0.75</td>
<td>1.00</td>
<td>0.75</td>
<td>1.00</td>
<td>0.75</td>
<td>1.00</td>
<td>0.75</td>
<td>1.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>TruthTableControl</td>
<td>a XOR b</td>
<td>0.80</td>
<td>0.68</td>
<td>0.80</td>
<td>0.68</td>
<td>0.80</td>
<td>0.68</td>
<td>0.80</td>
<td>0.68</td>
<td>0.80</td>
<td>0.68</td>
<td>0.80</td>
<td>0.68</td>
<td>0.80</td>
<td>0.68</td>
<td>0.80</td>
<td>0.68</td>
<td>0.80</td>
<td>0.68</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>VariableNegationControl</td>
<td>a XOR b</td>
<td>0.03</td>
<td>0.00</td>
<td>0.03</td>
<td>0.00</td>
<td>0.03</td>
<td>0.00</td>
<td>0.03</td>
<td>0.00</td>
<td>0.03</td>
<td>0.00</td>
<td>0.03</td>
<td>0.00</td>
<td>0.03</td>
<td>0.00</td>
<td>0.03</td>
<td>0.00</td>
<td>0.03</td>
<td>0.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>Classes in this Package</td>
<td>Boolean Expression</td>
<td>(a or b) and c</td>
<td>(a AND b) XOR c</td>
<td>Random Boolean expressions</td>
<td>Average</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>----</td>
<td>----------------------------------</td>
<td>--------------------</td>
<td>---------------</td>
<td>---------------</td>
<td>--------------------------</td>
<td>---------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Line Coverage</td>
<td>Branch Coverage</td>
<td>Line Coverage</td>
<td>Branch Coverage</td>
<td>Line Coverage</td>
<td>Branch Coverage</td>
<td>Line Coverage</td>
<td>Branch Coverage</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>BinaryExpressionSolver</td>
<td>0.27</td>
<td>0.24</td>
<td>0.27</td>
<td>0.24</td>
<td>0.27</td>
<td>0.24</td>
<td>0.27</td>
<td>0.24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>BinaryExpressionSolverTokenManage</td>
<td>0.38</td>
<td>0.18</td>
<td>0.40</td>
<td>0.21</td>
<td>0.38</td>
<td>0.18</td>
<td>0.39</td>
<td>0.19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>BooleanVariable</td>
<td>0.80</td>
<td>N/A</td>
<td>0.80</td>
<td>N/A</td>
<td>0.80</td>
<td>N/A</td>
<td>0.80</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Cube</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0/38</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>DNFControl</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>LogicalExpressionParser</td>
<td>0.51</td>
<td>0.55</td>
<td>0.51</td>
<td>0.55</td>
<td>0.51</td>
<td>0.55</td>
<td>0.51</td>
<td>0.55</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>LogicalExpressionParserTokenManage</td>
<td>0.50</td>
<td>0.32</td>
<td>0.53</td>
<td>0.37</td>
<td>0.50</td>
<td>0.32</td>
<td>0.51</td>
<td>0.34</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>SetOfBooleanVariables</td>
<td>0.93</td>
<td>0.77</td>
<td>0.93</td>
<td>0.77</td>
<td>0.93</td>
<td>0.77</td>
<td>0.93</td>
<td>0.77</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>SimpleCharStream</td>
<td>0.30</td>
<td>0.25</td>
<td>0.30</td>
<td>0.25</td>
<td>0.30</td>
<td>0.25</td>
<td>0.30</td>
<td>0.25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Token</td>
<td>0.75</td>
<td>1.00</td>
<td>0.75</td>
<td>1.00</td>
<td>0.75</td>
<td>1.00</td>
<td>0.75</td>
<td>1.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>TruthTableControl</td>
<td>0.60</td>
<td>0.40</td>
<td>0.60</td>
<td>0.40</td>
<td>0.60</td>
<td>0.40</td>
<td>0.60</td>
<td>0.40</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>VariableNegationControl</td>
<td>0.03</td>
<td>0.00</td>
<td>0.03</td>
<td>0.01</td>
<td>0.03</td>
<td>0.00</td>
<td>0.03</td>
<td>0.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Table 41. The Variabel Negation Coverage Information

<table>
<thead>
<tr>
<th>Number</th>
<th>Experiment</th>
<th>1 Line Coverage</th>
<th>1 Branch Coverage</th>
<th>2 Line Coverage</th>
<th>2 Branch Coverage</th>
<th>3 Line Coverage</th>
<th>3 Branch Coverage</th>
<th>4 Line Coverage</th>
<th>4 Branch Coverage</th>
<th>Average Line Coverage</th>
<th>Average Branch Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BinaryExpressionSolver</td>
<td>0.25</td>
<td>0.23</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.25</td>
<td>0.23</td>
<td>0.13</td>
<td>0.12</td>
</tr>
<tr>
<td>2</td>
<td>BinaryExpressionSolverTokenManager</td>
<td>0.39</td>
<td>0.22</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.39</td>
<td>0.22</td>
<td>0.20</td>
<td>0.11</td>
</tr>
<tr>
<td>3</td>
<td>BooleanVariable</td>
<td>0.80</td>
<td>N/A</td>
<td>0.20</td>
<td>N/A</td>
<td>0.20</td>
<td>N/A</td>
<td>0.80</td>
<td>N/A</td>
<td>0.50</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Cube</td>
<td>0.95</td>
<td>0.94</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.95</td>
<td>0.94</td>
<td>0.48</td>
<td>0.47</td>
</tr>
<tr>
<td>5</td>
<td>DNFCtrl</td>
<td>0.09</td>
<td>0.00</td>
<td>0.09</td>
<td>0.00</td>
<td>0.09</td>
<td>0.00</td>
<td>0.09</td>
<td>0.00</td>
<td>0.09</td>
<td>0.00</td>
</tr>
<tr>
<td>6</td>
<td>LogicalExpressionParser</td>
<td>0.52</td>
<td>0.57</td>
<td>0.42</td>
<td>0.48</td>
<td>0.42</td>
<td>0.48</td>
<td>0.52</td>
<td>0.57</td>
<td>0.47</td>
<td>0.53</td>
</tr>
<tr>
<td>7</td>
<td>LogicalExpressionParserTokenManager</td>
<td>0.52</td>
<td>0.37</td>
<td>0.42</td>
<td>0.23</td>
<td>0.42</td>
<td>0.23</td>
<td>0.53</td>
<td>0.38</td>
<td>0.47</td>
<td>0.30</td>
</tr>
<tr>
<td>8</td>
<td>SetOfBooleanVariables</td>
<td>0.93</td>
<td>0.77</td>
<td>0.21</td>
<td>0.09</td>
<td>0.21</td>
<td>0.09</td>
<td>0.93</td>
<td>0.77</td>
<td>0.57</td>
<td>0.43</td>
</tr>
<tr>
<td>9</td>
<td>SimpleCharStream</td>
<td>0.30</td>
<td>0.25</td>
<td>0.26</td>
<td>0.20</td>
<td>0.26</td>
<td>0.20</td>
<td>0.30</td>
<td>0.25</td>
<td>0.28</td>
<td>0.23</td>
</tr>
<tr>
<td>10</td>
<td>Token</td>
<td>0.75</td>
<td>1.00</td>
<td>0.75</td>
<td>1.00</td>
<td>0.75</td>
<td>1.00</td>
<td>0.75</td>
<td>1.00</td>
<td>0.75</td>
<td>1.00</td>
</tr>
<tr>
<td>11</td>
<td>TruthTableControl</td>
<td>0.70</td>
<td>0.44</td>
<td>0.08</td>
<td>0.00</td>
<td>0.08</td>
<td>0.00</td>
<td>0.70</td>
<td>0.44</td>
<td>0.39</td>
<td>0.22</td>
</tr>
<tr>
<td>12</td>
<td>VariableNegationControl</td>
<td>0.94</td>
<td>0.95</td>
<td>0.03</td>
<td>0.00</td>
<td>0.03</td>
<td>0.00</td>
<td>0.94</td>
<td>0.96</td>
<td>0.49</td>
<td>0.48</td>
</tr>
<tr>
<td>Experiment</td>
<td>Number</td>
<td>Classes in this Package</td>
<td>1 Line Coverage</td>
<td>1 Branch Coverage</td>
<td>2 Line Coverage</td>
<td>2 Branch Coverage</td>
<td>Average Line Coverage</td>
<td>Average Branch Coverage</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------------------------------------</td>
<td>--------</td>
<td>-------------------------</td>
<td>-----------------</td>
<td>-------------------</td>
<td>-----------------</td>
<td>-------------------</td>
<td>------------------------</td>
<td>-------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BinaryExpressionSolver</td>
<td></td>
<td></td>
<td>0.29</td>
<td>0.25</td>
<td>0.35</td>
<td>0.29</td>
<td>0.32</td>
<td>0.27</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BinaryExpressionSolverTokenManager</td>
<td></td>
<td></td>
<td>0.46</td>
<td>0.29</td>
<td>0.51</td>
<td>0.38</td>
<td>0.49</td>
<td>0.34</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BooleanVariable</td>
<td></td>
<td></td>
<td>0.80</td>
<td>N/A</td>
<td>0.80</td>
<td>N/A</td>
<td>0.80</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cube</td>
<td></td>
<td></td>
<td>0.95</td>
<td>0.94</td>
<td>0.95</td>
<td>0.94</td>
<td>0.95</td>
<td>0.94</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DNFControl</td>
<td></td>
<td></td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LogicalExpressionParser</td>
<td></td>
<td></td>
<td>0.55</td>
<td>0.58</td>
<td>0.37</td>
<td>0.25</td>
<td>0.46</td>
<td>0.42</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LogicalExpressionParserTokenManager</td>
<td></td>
<td></td>
<td>0.59</td>
<td>0.47</td>
<td>0.62</td>
<td>0.53</td>
<td>0.61</td>
<td>0.50</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SetOfBooleanVariables</td>
<td></td>
<td></td>
<td>0.93</td>
<td>0.77</td>
<td>0.93</td>
<td>0.77</td>
<td>0.93</td>
<td>0.77</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SimpleCharStream</td>
<td></td>
<td></td>
<td>0.30</td>
<td>0.25</td>
<td>0.30</td>
<td>0.25</td>
<td>0.30</td>
<td>0.25</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Token</td>
<td></td>
<td></td>
<td>0.75</td>
<td>1.00</td>
<td>0.75</td>
<td>1.00</td>
<td>0.75</td>
<td>1.00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TruthTableControl</td>
<td></td>
<td></td>
<td>0.96</td>
<td>0.84</td>
<td>0.96</td>
<td>0.84</td>
<td>0.96</td>
<td>0.84</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VariableNegationControl</td>
<td></td>
<td></td>
<td>0.94</td>
<td>0.95</td>
<td>0.94</td>
<td>0.96</td>
<td>0.94</td>
<td>0.96</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
TABLE 43. THE EXPRESSIONS FROM INTERNET COVERAGE INFORMATION

<table>
<thead>
<tr>
<th>Number</th>
<th>Classes in this Package</th>
<th>Line Coverage</th>
<th>Branch Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BinaryExpressionSolver</td>
<td>0.29</td>
<td>0.25</td>
</tr>
<tr>
<td>2</td>
<td>BinaryExpressionSolverTokenManager</td>
<td>0.50</td>
<td>0.34</td>
</tr>
<tr>
<td>3</td>
<td>BooleanVariable</td>
<td>0.80</td>
<td>N/A</td>
</tr>
<tr>
<td>4</td>
<td>Cube</td>
<td>0.95</td>
<td>0.94</td>
</tr>
<tr>
<td>5</td>
<td>DNFControl</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>6</td>
<td>LogicalExpressionParser</td>
<td>0.37</td>
<td>0.26</td>
</tr>
<tr>
<td>7</td>
<td>LogicalExpressionParserTokenManager</td>
<td>0.62</td>
<td>0.53</td>
</tr>
<tr>
<td>8</td>
<td>SetOfBooleanVariables</td>
<td>0.93</td>
<td>0.81</td>
</tr>
<tr>
<td>9</td>
<td>SimpleCharStream</td>
<td>0.30</td>
<td>0.25</td>
</tr>
<tr>
<td>10</td>
<td>Token</td>
<td>0.75</td>
<td>1.00</td>
</tr>
<tr>
<td>11</td>
<td>TruthTableControl</td>
<td>0.96</td>
<td>0.84</td>
</tr>
<tr>
<td>12</td>
<td>VariableNegationControl</td>
<td>0.94</td>
<td>0.95</td>
</tr>
<tr>
<td></td>
<td>Total coverage</td>
<td>0.57</td>
<td>0.51</td>
</tr>
</tbody>
</table>
Appendix M  Using COBERTURA in the case study

The data flow diagram of Figure 38 illustrates our use of GUITAR-COBERTURA integration by doing Ripping, Converting from tree to EFG, generating of test cases and replaying of test cases [3] in four vertical processes starting with Ripping and ending with Replaying, as well as the integration of COBERTURA for measuring structural coverage (three horizontal stages for measuring coverage starting with instrumenting and ending with GUITAR’s Replaying stage).
**Figure 38. Our framework for GUI functional system testing by integrating GUITAR [8] and COBERTURA [187]**