Comparing GUI Functional System Testing with Functional System Logic Testing - An Experiment

Abdulaziz Alkhalid
Department of Systems and Computer Engineering
Carleton University
Ottawa, Canada
akhalid@sce.carleton.ca

Yvan Labiche
Department of Systems and Computer Engineering
Carleton University
Ottawa, Canada
labiche@sce.carleton.ca

***Abstract*— The practitioner interested in reducing software verification effort may found herself lost in the many alternative definitions of Graphical User Interface (GUI) testing that exist and their relation to the notion of system testing. One result of these many definitions is that one may end up testing twice the same parts of the Software Under Test (SUT), specifically the application logic code. To clarify two important testing activities for the avoidance of duplicate testing effort, this paper studies possible differences between GUI testing and system testing experimentally. Specifically, we selected a SUT equipped with system tests that directly exercise the application code; We used GUITAR, a well-known GUI testing software to GUI test this SUT. Experimental results show important differences between system testing and GUI testing in terms of structural coverage and test cost.**

Keywords—System testing; GUI testing; Entity-Control-Boundary design principle

I. 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 are capable of serving different types of users with different levels of abilities (e.g. ordinary user, user with disability, Web user, or Mobile user). These advances in technology produce challenges for software testers who are responsible for software verification of those GUI-based software. As a result, software testers find themselves in front of several testing types to choose and use, such as GUI testing and system testing.

A well accepted definition of software system testing is that it is a phase of software testing conducted on the complete software to evaluate its compliance with its requirements, be they functional or non-functional [1]. However, there is 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 [2]. They argue that GUI system testing is system testing of the entire software through its GUI while 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 [3]. We conclude that Memon's notion of GUI testing encompasses both notions of GUI testing and GUI system testing of Ammann and Offutt. As further shown by our study of literature on the topic (section II), we conclude that the reader interested in testing a GUI-based software may found herself 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 GUITAR [4] 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 [5], thereby testing the application logic twice.

The paper therefore attempts to answer the following research questions:

Research Question 1. What are available definitions of system and GUI testing and how they relate to each other?

Research Question 2. How system testing (bypassing the UI) and GUI testing compare in terms of structural coverage and test cost?

Fig. 1 shows the focus of this paper. It illustrates several software testing definitions by showing the software divided in its GUI layer and its application logic layer. It illustrates that system testing can focus on the functional aspects of the System Under Test (SUT), referred to as functional system testing, or the non-functional aspects of the SUT also sometimes referred to as the "alities", referred to as non-functional system testing. Both 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 dashed line and triggering the application logic layer) or only the application logic layer. It also shows that our scope, 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 logic application directly.

We focus on desktop applications since such applications typically require more robust UIs [6, 7]. Another motivation is the difficulty, to the point of impracticality, of GUI system testing for any SUT with non-trivial UI: for instance, using GUITAR [4] on Microsoft WordPad in Windows 7 [5], which contains over 50 GUI events, is extremely expensive (in terms of number of tests). This is confirmed by the experiment we discuss in this paper.

The rest of this paper is organized as follows. Section II surveys possible definitions of GUI testing and shows how

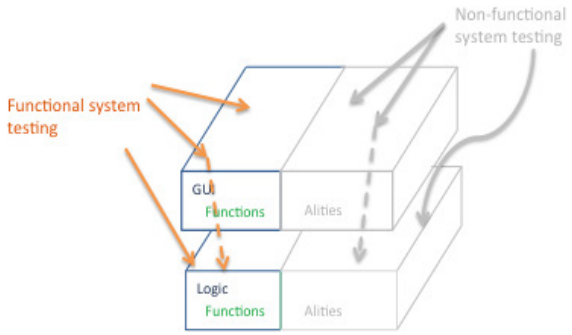

FIG. 1. Functional and non-functional system testing

they relate to one another. In doing so we reveal a risk of redundancies in testing effort between GUI testing and system testing and we introduce definitions we will rely on in the remainder of the paper. Section III presents the design of our experiment, including: the case study we will use, the measurement, the executions of tests. Results (Section IV) show differences between the two testing techniques in terms of test cost and structural coverage and confirm the risks of redundant efforts when conducting GUI testing and system testing. Section V presents related work. Section VI presents conclusions. Section VII presents an appendix.

II. DEFINITONS

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. The intent of this paper 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 available definitions of system and GUI testing and how they relate to each other?

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. Then, we identified chapters of 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". Section VII presents an appendix that contains the complete list of 52 books, both from the research laboratory and the Library. We believe that, for our search for definitions, looking into 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, 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 [3]. We used the dblp Computer Science Bibliography [8] to look for publications related the GUI testing when needed for a specific author.

The rest of this section is structured as follows. Subsection II.A reports on definitions about system testing. Subsection II.B describes definitions of GUI testing. Subsection II.C concludes and presents the definitions we will rely on in the reminder of this manuscript.

A. *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" [1]. This definition is in accordance with other authors' definitions [1, 9-13], with the IEEE definition of software system testing [14] as well as with the guide to the Software Engineering Body of Knowledge (SWEBOK guide) [15].

Ammann and Offutt define system testing as deriving tests from external descriptions of the software including specifications, requirements and design [2]. Naik and Tripathy define functional system testing as deriving tests that verify the system as thoroughly as possible over the full range of requirements specified in the requirements specification document, including requirements about the GUI [16]. Lewis defines black-box functional testing as a way to test conditions on the basis of the program or system's functionality [17] [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 [17] [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*, that decides how well the user is able to use and understand the application; *Documentation testing*, that 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; *Installation testing*, which verifies the ability to install the system successfully. This taxonomy of system test activities can be divided further [16, 17] but this is out of the scope of this paper.

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 [18] [page. 191].

Acceptance testing [1, 19] 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 [16]. The set of tests is usually a subset of the set of system tests [11, 17], 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 [10].

B. 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 with those which are close to the notion of GUI testing like system testing. Few published documents defined GUI testing as system testing. We discuss below two views of GUI testing that are representative of what we have found in our search for definitions.

GUI testing can be defined as system testing for software that has a GUI [3, 5], that 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) [3]. Similarly, Grilo et al. defined GUI testing as an activity for increasing confidence in the SUT and its correctness by finding defects in the GUI itself or the whole software application [20].

Memon defined GUI testing as a process that consists of a number of steps [21] which are similar to any testing activity: e.g., creating tests, executing tests..

Assuming the standard, IEEE definition of system testing we already discussed, we argue that GUI testing as defined by Memon creates tests that do not address performance, usability, safety, installation, nor other "alities" (Section II.A). 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 and the logic of a GUI-based software behave as expected[1] includes usability testing and functional testing [2]. 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. They 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" [2]. 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. We assume it is the latter. Input validation testing aims to verify whether the GUI "recognize[s] the user input and respond[s] correctly to invalid input" [2]. 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" [22]. 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 UI controls work and allow the user of the UI to navigate between screens.

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 [23] tell us that exercising the SUT through its UI (GUI system testing) and verifying the UI itself (GUI testing) separately would not be sufficient to ensure the entire SUT behaves as expected. Some separate verification of the application logic itself would be necessary.

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

C. Conclusion

In line with the majority of the references 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 [1, 9, 10, 17] do not explicitly mention the GUI, in case 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, though slightly different to, Ammann & Offutt definition, as discussed earlier.

Fig. 2 illustrates the main definitions we have encountered in our survey and that we have discussed in previous sections. It shows 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 whether tests go through the UI layer (dotted 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 from the underlying application logic.

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

[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 write that usability testing and functional testing are the two activities involved in determining whether the GUI and the logic of a GUI-based software behave as expected.

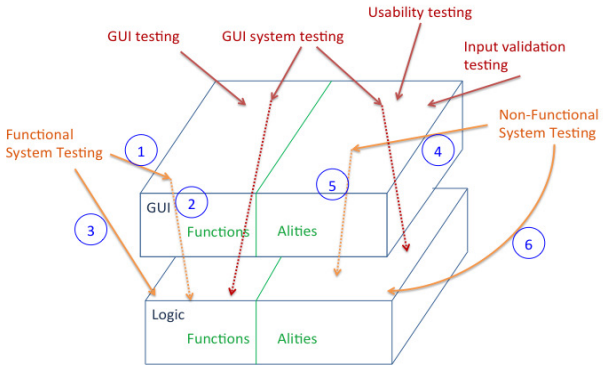

FIG. 2. RELATIONSHIP BETWEEN DIFFERENT TESTING TYPES

exercising the latter through the former (dotted 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 & Offutt's definitions. GUI testing is about the functional aspects of the GUI, focusing only on the UI layer, so the arrows 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).

To summarize, 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 Fig. 2), by isolating and focusing only on the UI (arrow 2), 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 system testing can be either functional or non-functional. So we use the term GUI functional system testing for our GUI testing experiments. As for GUI non-functional system testing, it is out of our scope. Functional system testing is one part of system testing, the other one being non-functional system testing, and they both include some form of GUI testing. In our experiments we refer to system test of the application logic code, so we use the term functional system logic testing in the rest of this paper. As for non-functional system logic testing, it is out of our scope.

III. EXPERIMENT DESIGN

In this section we discuss an experiment we conducted whereby we compare GUI functional system testing and functional system logic testing applied directly on the application logic, in an attempt to answer Research Question 2: How system testing (bypassing the UI) and GUI testing compare in terms of structural coverage and test cost? We first introduce the case study software we use in the experiment (section III.A). This software comes with functional system tests directly interacting with the application logic code. Next (section III.B) we discuss the measures we use in the experiment to answer the research question. Sections III.C and III.D discuss executing the functional system tests directly on the application logic code and the system tests created and executed through the GUI by GUITAR, respectively. Threats to validity are discussed in section III.E.

A. *Software Under Test (SUT)*

The GUI-based SUT provides three functionalities that work on Boolean expressions. The main window has three buttons. Clicking on any of the button(s) generates a child window to handle the corresponding functionality. The truth table window accepts a string representing a Boolean expression (TextField) 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 TextArea at the bottom of the window. The DNF Expression window computes ("Compute DNF" button) the disjunctive normal form of the Boolean expression provided in a text field and displays the result in a text area. The third window is an interface for an implementation of the variable negation testing technique [24]. The input must be provided as a series of terms of the DNF of a Boolean expression. The user must enter those terms in input TextField(s), one term per TextField. If the three default TextFields are not enough (i.e., the DNF has more than three terms), the user clicks the AddProductTerm button to add a new TextField. The user then can click the "Compute Var. Neg." button and the result appears in an output TextArea. The user can press on the "Show Text Cases" button, and the program shows a table with a set of entries satisfying the test objectives of the variable negation testing technique.

Each time a computation is asked (button), if the user has entered 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 and no computation is actually triggered.

The overall architecture of the SUT follows the Entity-Control-Boundary (ECB) design principle which divides classes over three main kinds of responsibilities [25]: boundary classes realize the UI, entity classes hold the data, control classes realize functionalities.

We had to update the GUI of the case study several times to fit the GUI testing tool, namely GUITAR. For instance, the original UI was made of tabbed panes, one pane for each of the three functionalities; Panes are not correctly handled by GUITAR and we changed that to new windows created upon clicking on buttons as discussed earlier. Thanks to the use of

the ECB design principle, those changes had no impact on the control and entity classes and therefore no impact on the system level JUnit tests.

B. Measurement

We measure structural coverage of each testing campaign, specifically line and branch coverage (i.e., the percentage of lines and branches executed by test runs) using COBERTURA [26]. In order to have a fair comparison between GUI system tests and functional system tests, since system tests interact directly with control classes, thereby bypassing the GUI, we do not measure coverage of GUI classes, system test classes, nor the main class. In other words we only measure coverage of individual control and entity classes which code should be exercised by both system tests and GUI tests, and total coverage for those classes. We measure coverage of the same 12 classes when executing each test suite.

We are also interested in test cost. 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 paper, an automatically generated GUITAR test does not necessarily click on a "Compute" button and therefore not every GUITAR test actually triggers a functionality of the SUT. To obtain a cost measure of GUITAR test suites that is comparable to the test suite cost measure of system test suites interacting directly with the application logic, we also count the number of GUITAR tests that contain a click on a "Compute" button as a measure of a GUITAR test suite.

C. Experiments—Functional system testing to the application logic code

The system test suite exercising the application logic code consists of 11 JUnit black-box tests. Six of them test the truth table functionality and hence they are applied on class TruthTableControl. Two test the DNF generation functionality and therefore exercise the DNFControl class. Three test the variable negation functionality and therefore exercise the VariableNegationControl class. Functional system 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.

Using this test suite we devised the following experiments. Experiment F only uses the truth table system test cases, which each use one Boolean expression as an input, specifically: `x AND (y OR Z)`, `a xor b`, `a or (not b)`, `a XNOR (b nand R)`, `a NOR b`, `(A AND d) XOR (not(C XNOR (NOT b)))`. Experiment G only uses the DNF system test cases, which each use one Boolean expression: `(a or b) and c`, and `(a AND b) XOR c`. Experiment H only uses the three test cases exercising the variable negation functionality. The first test case has two input Boolean expressions: `a and b and not c`, and `a and d`. The second test case has four input Boolean expressions: `x and not y and z`, `y and w`, `not x and w`, and `y and not z`. The third test case has two input Boolean expressions: `x and not y and z`, and `y and w`. Experiment I contains all 11 test cases.

D. Experiments—GUI functional system testing

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 [5]. We obtained 200 test cases by using the default value L=3. Each of those tests is a traversal of a graph (so-called the Event Flow Graph—EFG) representing the entire GUI of the SUT and that GUITAR's Ripper creates made of a triplet (L=3) of GUI widgets with a shortest path prefix: each triplet of GUI widgets of the EFG is exercised at least once. 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 a text field used in the truth table functionality and then a text input for a 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) to the variable negation functionality. The different numbers of tests in simply due to the larger number of widgets of the latter window (recall several text fields are needed). We also note that GUITAR does not know when, in a test case, i.e., in a sequence of events, to actually trigger a functionality, i.e., when to click on a "compute" button: a user would click only after filling the required text field(s). 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 actually 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 as a fact, after having checked the 200 tests, is that in the 17 (resp., 11, 129) tests[2] 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, ensuring that the application logic code of each functionality is at least triggered once. 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. A compute button triggering some functionality of the application logic was clicked eight times within the replay of 200 test cases.

Last we note that GUITAR uses, during the replaying process, its own set of strings for text fields and these are not Boolean expressions. Using the defaults strings provided by GUITAR would not trigger functionalities but would result in parsing errors: The application code starts to parse the input

[2] 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. So it would be wrong to sum up these numbers and compare the result to the total number of tests (200).

string and reports a parsing error (in an error window) if the string is not a Boolean expression. One can specify test inputs (i.e., Boolean expressions) for GUITAR tests in many different ways. In an attempt to be as systematic as possible and obtain coverage results that can be compared to those obtained by system tests, we devised a number of experiments each one having a specific test input selection procedure.

Experiment A for truth table. This experiment tests the truth table functionality only. A Boolean expression is used each time a TextField 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 TextField. The question is however: which Boolean expression to use when a GUITAR test case needs one to exercise the truth table functionality? To have a fair comparison with system tests we used their input Boolean expressions and split Experiment A into seven experiments. Since the system 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 TextField for the truth table functionality needs an input. Specifically, in experiment A.1, we use input "`x AND (y OR Z)`" each time we need such an input for a TextField that is needed for truth table. In experiment A.2, we use "`a xor b`". In A.3, we use "`a or (not b)`". In A.4, we use "`a XNOR (b nand R)`". In A.5, we use "`a NOR b`". In A.6, we use "` (A AND d) XOR (not(C XNOR (NOT b)))`". A seventh variation of experiment A randomly assigns those Boolean expressions to text fields for the truth table functionality, ensuring that each Boolean expression is used at least once.

Experiment B for DNF. This experiment tests the DNF functionality only. A Boolean expression is used each time a TextField for the DNF functionality appears in a test case, and we use GUITAR's default strings for any other TextField. Similarly to experiment A, and since the system test suite uses two inputs for this functionality, 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 inputs ensuring that each Boolean expression is used at least once.

Experiment C for variable negation. Similarly to previous A and B, a Boolean expression is used each time a TextField for the variable negation functionality appears in a test case, and we use GUITAR's default strings for any other TextField. Again, we split this experiment, reusing inputs we used for functional system testing. In experiment C.1, each time the variable negation functionality needs an input (i.e., a series of Boolean expressions) we use the following three terms: "`x and not y and z`", "`y and w`", and "not x and w". Notice that this is the second system 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. The variable negation GUI provides three TextFields to the user and a button to add more TextFields if needed. GUITAR systematically and only uses three. The Add product term button is sometimes clicked, thereby adding a fourth text field, but when this is the case, the fourth text field is not used to provide a value. The second experiment (C.2) uses the first system test input systematically each time the GUITAR test cases requires an input for variable negation. Since this input has two terms and GUITAR systematically fills the three TextFields, we use GUITAR's default input for the third input. The third experiment (C.3) uses the third system test input, with two terms, similarly to C.2. 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.

Experiment D for the three functionalities. This experiment works on all three functionalities together. We performed two sub-experiments. In experiment D.1, each time an input is needed for truth table (resp., DNF, Variable negation) we used that of experiment A.1 (resp. B.1, C.1). In the second sub-experiment, we used the randomized selection of the previous experiments, i.e., as in A.7, B.3, and C.4. 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 and selection was picked up in random way.

Experiment E using expressions from the Internet. This experiment works on the three functionalities together. However, instead of using GUITAR's default values or the values used during functional system testing, we used Boolean expressions we collected from the Internet [27-34] 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 test inputs. The Boolean expressions were mostly available in websites related to academic mathematical topics and computer circuits design. In doing so we attempt to simulate what a user may do with GUITAR test cases when testing our case study.

E. Threats to validity

Similarly to any experiment, our work is subject to threats to validity [35].

Threats to construct validity relate to our choice of measurement as a way to compare system testing and GUI testing. Although many different criteria can be considered when comparing two different testing techniques, structural coverage is a well-know 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 [23]. 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 test [36] or GUI tests [37], the cost of a test suite as the number of test cases of that test suite.

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, we believe, an as fair as possible way.

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 highly depend 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 this 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 system test suite that we did not create and that achieves a very decent level of coverage; We used the default GUITAR settings which have been shown by others to work well; We systematically designed our experiments, prior to conducting them, to ensure a fair comparison.

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

IV. RESULTS

We first discuss the results of experiments F, G, H and I about functional system testing directly applied on the application logic (i.e., control) classes (section IV.A) and then the results of experiments A, B, C, D and E on functional system testing through the GUI with GUITAR (section IV.B). We summarize the results in section IV.C.

A. *Results about functional system logic testing (F, G, H, I)*

TABLE I shows coverage levels achieved 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 in experiment F. 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: e.g., in experiments G and H, the coverage of TruthTableControl is not null. We confirmed that without those specific services, i.e., methods in class TruthTableControl that are only used by other control classes, experiment F 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. Not surprisingly, the test suite does not trigger the third functionality: 0 line and branch coverage of VariableNegationControl.

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 specific format for program debugging. Similarly the previous experiment, it is not surprising that the test suite does not cover at all class DNFControl, and we observe that TruthTableControl is somewhat covered.

The union of the three previous test suites (experiment I) achieves 100% line and branch coverage for TruthTableControl and DNFControl, and only 84% line and 87% branch coverage of VariableNegationControl.

B. *Results about GUI functional system testing*

In experiment A (TABLE II), results show how coverage increases with the increase of complexity of the test input Boolean expression: as a simple measure of complexity, we consider that the more terms and variety of Boolean expressions the higher the complexity of the Boolean expression). For example, in experiment A.6 line and branch coverage are maximum, whereas they are minimum in experiment A.2. In experiment A.7, the coverage values are at their minimum, which may look like a contradiction since we

TABLE I. LINE AND BRANCH COVERAGE FOR EXPERIMENTS F, G, H AND I (N/A WHEN NO BTRANCH TO MEASURE)

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

use all test inputs. However, remember that not every filled TextField for truth table is followed by a click on the "Compute" button in test cases; In experiment A.7, it just happened that when a click happens, the Boolean expression we selected at random for the TextField was always the simplest one (`a xor b`).

We make similar observations for experiment B, regarding the low coverage value obtained with the random selection of inputs. We note that experiment B1 and B2 use expressions of similar complexity, resulting in similar coverage values.

For experiment C, it is noticeable that in experiments C.2 and C.3, we obtained low coverage values for each of line and branch coverage. We justify this by the lack of a third valid (Boolean) input in both experiments. The SUT parses all inputs, discovers that one is not a valid Boolean expression (parsing error), brings this to the attention of the user and stops: the application logic does not execute. This observation shows how the input affects the coverage in our experiment and hence justifies our controlled test selection procedure.

For experiment D, the overall results are better than when testing each functionality separately, as expected since all functionalities are exercised with some Boolean expressions, but coverage only reaches 58% (line) and 52% (branch) even though we used the same test inputs as with functional system logic testing. This is again due to the fact that not all TextField input is followed by a button click in GUITAR tests. The random selection 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 and this indicates more sensitivity of branch coverage to the test input. In other words, it is a different observation from experiment A where we got the same coverage in two experiments (random and normal) when those experiments depended on the same input (i.e., the same Boolean expression).

With retrospect, in light of the results of GUI functional system testing coverage, another interesting combination would have been A.6, B.2 and C.1 because this maximizes coverage for each functionality separately. We did not a posteriori consider this combination because we did not expect drastically different (improved) coverage results.

For experiment E (TABLE III), 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; and the parser is only triggered through the control classes which causes problems of controllability of its code. Unit tests of the parser would help us increase this coverage.

TABLE III. LINE AND BRANCH COVERAGE FOR EXPERIMENT E

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

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 [2]. For example, it is easy to control a piece of software for which all inputs are values entered from a keyboard [2, 38, 39]. 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, 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.

TABLE II. TOTAL LINE AND BRANCH COVERAGE FOR EXPERIMENTS A, B, C AND D

Experiment A				Experiment B				Experiment C		
Exp #	Boolean Expression	Line Coverage	Branch Coverage	Exp #	Boolean Expression	Line Coverage	Branch Coverage	Exp #	Line Coverage	Branch Coverage
A.1	x AND (y OR Z)	0.35	0.25	B.1	(a or b) and c	0.36	0.25	C.1	0.52	0.43
A.2	a xor b	0.33	0.23	B.2	(a AND b) XOR c	0.37	0.27	C.2	0.16	0.10
A.3	a or (not b)	0.36	0.26	B.3	Random Boolean expressions	0.36	0.25	C.3	0.16	0.10
A.4	a XNOR (b nand R)	0.37	0.28	Experiment D				C.4	0.52	0.43
A.5	a NOR b	0.34	0.24	Exp #	Line Coverage	Branch Coverage				
A.6	(A AND d) XOR (not(C XNOR (NOT b)))	0.40	0.33	D.1	0.58	0.50				
A.7	Random Boolean expressions	0.33	0.23	D.2	0.58	0.52				

C. Results Analysis—System testing vs. GUI functional system testing

As alluded to earlier when we discussed the setup of GUITAR and the tailoring of the tests it generates to our SUT (i.e., the test input selection), using GUITAR involved a lot of effort (e.g., looking at the details of the EFG and the generated tests) and therefore time. It took significantly less effort and time to the original author of the code to generate system tests. Although we do not have precise measurements of these two time efforts, we conjecture there is a difference of more than an order of magnitude between the two. It terms of test suite execution, executing the entire system test suite is almost instantaneous; 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 system test suite has eleven tests; the GUI test suite has 200 tests. All tests for functional system logic testing call the functionality while GUI system tests call the functionality eight times.

The overall coverage results of our experiments are reported in TABLE IV: since we have sub-experiments for GUI 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; except for experiment (G and B) in which they 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 test because a test does not have to fill all Boolean terms).

For comparison purpoess we excluded some classes from the instrumentation. In particular we excluded classes that were not covered (0% line coverage) by system 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 functional 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 system tests also include robustness test, these classes would also be covered.

TABLE IV. LINE AND BRANCH COVERAGE FOR ALL THE EXPERIMENTS

	ID	Main Experiment	Line Coverage	Branch Coverage
			Average Total (Standard Deviation)	
GUI	A	Truth table	0.35 (0.03)	0.26 (0.04)
	B	DNF	0.36 (0.01)	0.26 (0.01)
	C	Variable negation	0.34 (0.21)	0.27 (0.19)
	D	All three operations	0.58 (0.01)	0.51 (0.02)
	E	Expressions from Internet	0.57	0.51
System	F	Truth table	0.42	0.39
	G	DNF	0.36	0.27
	H	Variable negation	0.49	0.40
	I	All three operations	0.60	0.57

TABLE V. LINE AND BRANCH COVERAGE FOR CONTROL CLASSES

	Experiment E		Experiment I		Simulated-I	
Class#	Line	Branch	Line	Branch	Line	Branch
DNFControl	1	1	1	1	1	1
TruthTableControl	0.96	0.84	1	1	1	1
VariableNegationControl	0.94	0.95	0.84	0.87	94	98.9

The difference of coverage between functional system logic testing and GUI functional system testing shows the values of difference is greater than or equal to zero except for two outlier classes; which we discuss next. The first outlier is branch coverage for class SetOfBooleanVariables in the parser. 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 system test inputs, but this happens in Boolean expressions we collected from the Internet.

The second outlier is for class VariableNegationControl. By inspecting the code we found a method in this class that is covered by GUI tests but not by system tests. The method breaks the Boundary-Control-Entity 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 tests do 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 tests, and the first outlier would not take place if a Boolean expression with twice the same Boolean variable were used as test input.

We simulated, by considering the lines and branches these methods contribute, the coverage one would obtain if (1) the Entity-Boundary-Control principle were adequately followed, i.e., the code missed by original system 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 system test inputs. TABLE V 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. 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. We conclude that GUI functional system testing is more expensive than functional system logic testing.

V. RELATED WORK

Though we have not found any study like ours in the literature, we can relate to some related work on GUI testing. Then, we move to discuss tools for GUI testing and hence we justify our choose of GUITAR.

Memon's PhD thesis [40] presents a framework for GUI testing, called GUITAR [5], that generates, runs, and assesses GUI tests [6]. Descriptions of the main components of that framework with further optimizations and improvements of the process may be found for reverse engineering [41], coverage analysis of test cases [42], test oracle generation [43], and regression testing [44, 45]. GUITAR is a tool that performs GUI system testing per authors definitions, but GUI functional system testing and input validation testing based on our experience with it. Another approach for GUI functional system testing is to represent the behavior of the GUI as a state model, possibly with technology to avoid the state explosion problem [46], to generate tests [47]. Just as in every level of testing, GUI tests must consider both valid and invalid inputs and hence we tried to apply both inputs when doing GUI testing. From a tool perspective, many tools exist for capturing manually entered sequences. In this "capture/replay" paradigm, test case selection involves entering every input sequence of events manually. Model-based testing (MBT) approaches for GUI-testing present a test case selection process which constructs test cases based on the model [48, 49].

Script-based tools are widely used such as JFCUnit, Selenium WebDriver, Robotium, Abbot, and SOAtest [5]. As another example, the Sikuli testing framework [50] employs computer vision techniques to develop a visual language for writing test scripts. MBT approaches employ tool support for automated test case generation. Several tools exist for generating test cases automatically. Automated test planning [51] uses AI planning to generate test cases based on the state of a GUI before and after executing a user-defined operation [5]. GUI variants [52] enable testers to convert business logic test cases into presentation logic test cases. The PETTool [53] identifies patterns in GUIs and generates generic testing solutions based on the patterns.

Beside Memon's work, reverse engineering approaches include Silva and colleagues' [48] which automatically reverse engineers a behavioral model of the GUI from the source code of Java Swing-based GUI applications, and Pavia and colleagues' [54] which reverse engineers a GUI into a specification model which can be used by Spec Explorer [55] to generate test cases [5]. Amalfitano et al. presented a tool [56] that is a similar MBT tool using a reverse engineering technique to automatically construct the GUI model. The tool 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 [49] and Revangie [57] employ similar techniques for web applications.

REST [58] enables a user to evolve test scripts when the GUI changes. The tool detects differences between the original and modified versions of a GUI and generates a warning if a script needs correction.

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 over other tools in a comparative study [5] that included a comparison between: (1) GUITAR [4], a research tool that handles the complete life cycle of automated GUI testing from ripping the SUT to replaying test cases; (2) NModel [59], a model-based testing framework for C# programs; (3) Quick Test Pro [60], a popular, proprietary, multi-platform tool for test automation; (4) Selenium [61] WebDriver, a popular API for browser automation. It was also found to be superior to (6) Marathon [62], a capture/reply tool that uses Paython, (7) Sikuli [50], a reverse engineering tool that uses image recognition techniques (8) JAutomate [63], a tool for GUI testing based on image recognition.

GUITAR generates a set of XML files when testing a GUI-based application. Based on our experience, it is possible to use Gephi [64] in order to visualize the Event-Flow-Graph of the SUT. This is an advantage of using GUITAR as visualization allows manual verification of models.

VI. CONCLUSION

We presented an experimental investigation of the concept of GUI testing in term of definitions, steps, requirements, design and capture/replay. The paper investigated relationships between GUI testing and other types of testing such as system testing. We noted disagreements about what GUI testing is, in comparison with system testing. We therefore decided to conduct an experiment whereby we study the differences in terms of structural coverage of the application logic code between system tests and GUI tests.

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 more time and computation cost than system testing. Although replications of our experiments are necessary to precisely understand the phenomenon we have encountered, our results empirically prove the existence of duplicate effort when using GUI testing and system testing simultaneously. Our investigation suggests that the use of system testing on the application logic code would be a less costly verification technique of the application logic of the SUT than GUI testing.

The validity threats were evaluated. This is important to do upfront to ensure that the threats are minimized [35]. It is close to impossible to avoid all threats [35]. But all threats in our experiment were identified and whenever possible mitigated. Based on our evaluation, we were hopefully ready to run the experiment and it is possible to repeat the experiment several times when we had any doubt in coverage results when conducting the experiment. For example, many of our GUI testing experiments were repeated many times using two versions of GUITAR and with two types of integration with COBERTURA with ANT [65] and with shell script. We believe our results are valid and generalizable.

To summarize, we believe the software testing research community and testing practitioners need to better define what GUI testing is in comparison to system testing and when GUI testing "à la" GUITAR should be used.

VII. APPENDIX

- Abbott, J., Software testing techniques. 1986.
- Ammann, P., J. Offutt, Introduction to software testing. 2008, Cambridge; New York;: Cambridge University Press.
- Beydeda, S. and V. Gruhn, Testing Commercial-off-the-Shelf Components and Systems. 2005: Springer Berlin Heidelberg.
- Binder, R., Testing Object-oriented Systems: Models, Patterns, and Tools. 2000: Addison-Wesley.
- Black, R., Advanced software testing. 2009, Santa Barbara, Calif: Rocky Nook.
- Bruegge and Dutoit, Object-Oriented Software Engineering: Using UML, Patterns and Java. 2000.
- Chip, D., Software Test Engineering with IBM Rational Functional Tester: The Definitive Resource. 2009: Pearson Education India.
- Desikan, S. and G. Ramesh, Software testing: principles and practice. 2006, India: Pearson Education
- 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
- Dustin, E., Garrett, T., Gauf, B., Implementing automated software testing: how to lower costs while raising quality. 2009, Upper Saddle River, N.J: Addison-Wesley
- Elfriede Dustin, Jeff Rashka, John Paul, Automated software testing: introduction, management, and performance. 1999, Addison-Wesley.
- Farrell-Vinay, P., Manage software testing. 2008, Boca Raton: Auerbach Publications
- Fewster, M. and D. Graham, Software Test Automation: Effective Use of Test Execution Tools. ACM Press Series. 1999: Addison-Wesley
- Gomaa, H., Designing Concurrent, Distributed, and Real-Time Applications with UML. 2000: Addison-Wesley Professional
- Hierons, R.M., J.P. Bowen, and M. Harman, Formal Methods and Testing: An Outcome of the FORTEST Network. Revised Selected Papers. illustrated ed. Lecture Notes in Computer Science, ed. M. Harman. 2008: Springer Berlin Heidelberg.
- Homes, B., Fundamentals of software testing. 2012.
- IEEE, Standard Glossary of Software Engineering Terminology (ANSI). 1991, The Institute of Electrical and Electronics Engineers Inc.
- Jorgenson, P.C., *Software Testing: A Craftsmans Approach*, in *Taylor & Francis Group*. 2008: New York.
- Pfleeger, S.L. and J.M. Atlee, Software engineering: theory and practice. 1998: Pearson Education India.
- Jorgenson, P.C., Software Testing: A Craftsmans Approach, in CRC Press. 1995: New York.
- Kaner, C., Fiedler, R., Foundations of software testing: a BBST workbook. 2014: Context Driven Press.
- Kaner, C., B. Pettichord, and J. Bach, Lessons learned in software testing: a context-driven approach. 2002, New York: Wiley.
- King, J.C., Symbolic Execution and Program Testing. Communications of the ACM 1976.
- Koirala, S., Sheikh, S., Software testing interview questions. 2008, Sudbury, Mass: Jones and Bartlett.
- Lewis, W.E., Software testing and continuous quality improvement. CRC press, 2004.
- Li, K. and M. Wu, Effective GUI testing automation: Developing an automated GUI testing tool. John Wiley & Sons, 2006.
- Li, K. and M. Wu, Effective software test automation: developing an automated software testing tool. 2006: John Wiley & Sons.
- Majchrzak, T.A., Improving software testing: technical and organizational developments. 2012, New York; Berlin: Springer.
- Mathur, A. Foundations of software testing: fundamental algorithms and techniques. 2013, New Delhi: Dorling Kindersley (India).
- Mathur, A.P., Foundations of Software Testing. 2008: Pearson Education.
- McGregor, J.D. and D.A. Sykes, A practical guide to testing object-oriented software. 2001: Addison-Wesley Professional.
- Memon, A.M., A comprehensive framework for testing graphical user interfaces, in Computer Science. 2001, University of Pittsburgh.
- Mili, A.,Tchier, F., Software testing: concepts and operations. 2015: Wiley.
- Mitchell, J.L., R. Black, Advanced software testing. 2015, Santa Barbara, Rocky Nook.
- Myers, G.J., C. Sandler, and T. Badgett, The art of software testing. 2011: John Wiley & Sons.
- Naik, S. and P. Tripathy, Software testing and quality assurance: theory and practice. 2011: John Wiley & Sons.
- Notenboom, E., Testing Embedded Software. 2003: Addison-Wesley.
- Commission of Ieee-Standards Board Ieee Xplore International Organization, I.o.E.a.E.E.a.C.o.I.-S.B.I.X.I., Software and systems engineering: software testing. 2013, New York; Geneva; ISO.
- Patton, R., Software testing. 2001, Indianapolis, Ind: Sams.
- Pezze, M. and M. Young, Software testing and analysis: process, principles, and techniques. 2007: John Wiley & Sons.
- Pries, K.H. and J.M. Quigley, Testing complex and embedded systems. 2011: CRC Press.
- 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.
- Schutz, W., The testability of distributed real-time systems. Vol. 245. 1993: Springer.
- Sharma, M., R. Padmanaban, and C.R.C. Press, Leveraging the wisdom of the crowd in software testing. 2015, Boca Raton: CRC Press.
- Singh, S., G. Singh, and S. Singh, Software Testing. International Journal of Advanced Research in Computer Science, 2010. 1(3): p. 403-406.
- Utting, M. and B. Legeard, Practical model-based testing: a tools approach. 2010: Morgan Kaufmann.
- Vance, S., Quality code: software testing principles, practices, and patterns. 2013, Upper Saddle River, NJ: Addison-Wesley.
- Werner, S., The Testability of Distributed Real-Time Systems. 1993: Kluwer Academic Publishers. 160.
- Whittaker, J.A., Exploratory Software Testing: Tips, Tricks, Tours, and Techniques to Guide Test Design. 2009: Pearson Education.
- Colin Willcock, Thomas Deiß, Stephan Tobies, Stefan Keil, Federico Engler, Stephan Schulz, Anthony Wiles, An Introduction to TTCN-3. 2011: WILEY.
- Pressman, R., Software Engineering: A practitioner approach, 6th edition, McGrawHill, 2005.

REFERENCES

[1] Desikan, S. and G. Ramesh, Software testing: principles and practice. 2006, India: Pearson Education

[2] Ammann, P. and J. Offutt, Introduction to Software Testing. Vol. 1. 2008, New York: Cambridge University Press.

[3] Banerjee, I., B. Nguyen, V. Garousi, and A. Memon, Graphical user interface (gui) testing: Systematic mapping and repository. Information and Software Technology, 2013. 55(10): p. 1679-1694.

[4] Memon, A., GUITAR. 2015, https://sourceforge.net/projects/guitar/.

[5] Nguyen, B.N., B. Robbins, I. Banerjee, and A. Memon, GUITAR: an innovative tool for automated testing of gui-driven software. Automated Software Engineering, 2014. 21(1): p. 65-105.

[6] Ganov, S.R., C. Killmar, S. Khurshid, and D.E. Perry. Test generation for graphical user interfaces based on symbolic execution. in Proceedings of the 3rd international workshop on Automation of software test. 2008: ACM.

[7] Forrester, J.E. and B.P. Miller. An empirical study of the robustness of Windows NT applications using random testing. in Proceedings of the 4th USENIX Windows System Symposium. 2000: Seattle.

[8] dblp. Computer Science Bibliography. 2016 [cited 2014; Available from: http://dblp.uni-trier.de/.

[9] Abbott, J., Software testing techniques. 1986.

[10] Homes, B., Fundamentals of software testing. 2012.

[11] Myers, G.J., C. Sandler, and T. Badgett, The art of software testing. 2011: John Wiley & Sons.

[12] Pries, K.H. and J.M. Quigley, Testing complex and embedded systems. 2011: CRC Press.

[13] Schutz, W., The testability of distributed real-time systems. Vol. 245. 1993: Springer.

[14] Std, I., IEEE Standard for Software and System Test Documentation. 2008: p. 1-150.

[15] Alain, A. and W.M. James, Guide to the Software Engineering Body of Knowledge - SWEBOK, ed. A. Alain, et al. 2004: IEEE Press. 228.

[16] Naik, S. and P. Tripathy, Software testing and quality assurance: theory and practice. 2011: John Wiley & Sons.

[17] Lewis, W.E., Software testing and continuous quality improvement. CRC press, 2004.

[18] Jorgenson, P.C., Software Testing: A Craftsmans Approach, in Taylor & Francis Group. 2008: New York.

[19] Pezze, M. and M. Young, Software testing and analysis: process, principles, and techniques. 2007: John Wiley & Sons.

[20] Grilo, A.M., A.C. Paiva, and J.P. Faria, Reverse engineering of gui models for testing, in 2010 5th Iberian Conference on Information Systems and Technologies (CISTI). 2010. p. 1-6.

[21] Memon, A.M., A comprehensive framework for testing graphical user interfaces, in Computer Science. 2001, University of Pittsburgh.

[22] IEEE, T.I.o.E.a.E.E., Standard Glossary of Software Engineering Terminology (ANSI). 1991.

[23] Binder, R., Testing object-oriented systems: models, patterns, and tools. 2000: Addison-Wesley Professional.

[24] Weyuker, E., T. Goradia, and A. Singh, Automatically generating test data from a Boolean specification, in IEEE Transactions on Software Engineering. 1994. p. 353-363.

[25] Bruegge and Dutoit, Object-Oriented Software Engineering: Using UML, Patterns and Java. 2000.

[26] Lee, C., COBERTURA. 2015, http://cobertura.github.io/cobertura/.

[27] AllAboutCircuits. Boolean Expressions. 2015 [cited; Available from: http://www.allaboutcircuits.com/textbook/digital/chpt-7/demorgans-theorems/.

[28] BasicGatesandFunctions. Boolean Expressions. 2015 [cited; Available from: http://www.ee.surrey.ac.uk/Projects/CAL/digital-logic/gatesfunc/index.html#example.

[29] Carter, J. Boolean Expressions. 2015 [cited; Available from: http://www.coe.uncc.edu/~jcarter/Elet3285/pageicon.gif.

[30] Dunn, K. Boolean Expressions. 2015 [cited; Available from: http://district.bluegrass.kctcs.edu/kevin.dunn/files/Simplification/4_Simplification_print.html.

[31] ElectronicsTutorials. Boolean Expressions. 2015 [cited; Available from: http://www.electronics-tutorials.ws/boolean/bool_8.html.

[32] IndiaBix. Boolean Expressions. 2015 [cited; Available from: http://www.indiabix.com/digital-electronics/boolean-algebra-and-logic-simplification/.

[33] NationalInstruments. Boolean Expressions. 2015 [cited; Available from: http://www.ni.com/example/14493/en/.

[34] Sandbox. Boolean Expressions. 2015 [cited; Available from: http://sandbox.mc.edu/~bennet/cs110/boolalg/simple.html.

[35] Wohlin, C., P. Runeson, M. Host, M.C. Ohlsson, B. Regnell, and A. Wesslen, Experimentation in software engineering. 2012: Springer Science & Business Media.

[36] Hutchins, M., H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria. in Proceedings of the 16th international conference on Software engineering. 1994: IEEE Computer Society Press.

[37] Brooks, P.A. and A.M. Memon. Automated GUI testing guided by usage profiles. in Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering. 2007: ACM.

[38] Freedman, R.S., Testability of software components. Software Engineering, IEEE Transactions on, 1991. 17(6): p. 553-564.

[39] Gao, J. Component testability and component testing challenges. in Proceedings of International Workshop on Component-based Software Engineering (CBSE2000, held in conjunction with the 22nd International Conference on Software Engineering (ICSE2000). 2000.

[40] Memon, A.M., A comprehensive framework for testing graphical user interfaces. 2001, University of Pittsburgh.

[41] Memon, A.M., I. Banerjee, and A. Nagarajan. GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing. in WCRE. 2003.

[42] McMaster, S. and A. Memon, Call-stack coverage for gui test suite reduction. IEEE Transactions on Software Engineering, 2008. 34(1): p. 99-115.

[43] Memon, A., I. Banerjee, and A. Nagarajan. What test oracle should I use for effective GUI testing? in Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on. 2003: IEEE.

[44] Memon, A., I. Banerjee, N. Hashmi, and A. Nagarajan. DART: a framework for regression testing" nightly/daily builds" of GUI applications. in Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on. 2003: IEEE.

[45] Memon, A.M. Using tasks to automate regression testing of GUIs. in IASTED International Conference on Artificial Intelligence and Applications-AIA. 2004.

[46] White, L. and H. Almezen. Generating test cases for GUI responsibilities using complete interaction sequences. in Software Reliability Engineering, 2000. ISSRE 2000. Proceedings. 11th International Symposium on. 2000: IEEE.

[47] Shehady, R.K. and D.P. Siewiorek. A method to automate user interface testing using variable finite state machines. in Fault-Tolerant Computing, 1997. FTCS-27. Digest of Papers., Twenty-Seventh Annual International Symposium on. 1997: IEEE.

[48] Silva, J.L., J.C. Campos, and A.C.R. Paiva, Model-based user interface testing with Spec Explorer and ConcurTaskTrees. Electronic Notes in Theoretical Computer Science, 2008. 208: p. 77-93.

[49] Mesbah, A. and A. Van Deursen. Invariant-based automatic testing of AJAX user interfaces. in Proceedings of the 31st International Conference on Software Engineering. 2009: IEEE Computer Society.

[50] Chang, T.-H., T. Yeh, and R.C. Miller. GUI testing using computer vision. in Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. 2010: ACM.

[51] Memon, A.M., M.E. Pollack, and M.L. Soffa, Hierarchical GUI test case generation using automated planning. IEEE transactions on software engineering, 2001. 27(2): p. 144-155.

[52] Nguyen, D.H., P. Strooper, and J.G. Suess. Model-based testing of multiple GUI variants using the GUI test generator. in Proceedings of the 5th Workshop on Automation of Software Test: ACM.

[53] Cunha, M., A.C.R. Paiva, H.S. Ferreira, and R. Abreu. PETTool: a pattern-based GUI testing tool. in Software Technology and Engineering (ICSTE), 2010 2nd International Conference on. 2010: IEEE.

[54] Paiva, A.C.R., J.o.C.P. Faria, and P.M.C. Mendes. Reverse engineered formal models for GUI testing. in International Workshop on Formal Methods for Industrial Critical Systems. 2007: Springer.

[55] Veanes, M., C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson, Model-based testing of object-oriented reactive systems with Spec Explorer, in Formal methods and testing. 2008, Springer. p. 39-76.

[56] Amalfitano, D., A.R. Fasolino, and P. Tramontana. A gui crawling-based technique for android mobile application testing. in Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on. 2011: IEEE.

[57] Draheim, D., C. Lutteroth, and G. Weber. A Source Code Independent Reverse Engineering Tool for Dynamic Web Sites. in CSMR. 2005.

[58] Grechanik, M., Q. Xie, and C. Fu. Creating GUI testing tools using accessibility technologies. in Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on. 2009: IEEE.

[59] CodePlex. NModel. 2016 [cited; Available from: http://nmodel.codeplex.com/.

[60] HP. Quick Test Pro. 2016 [cited; Available from: http://www.hp.com/QuickTestPro.

[61] WebDriver, S. Selenium. 2016 [cited; Available from: http://www.seleniumhq.org/projects/webdriver/.

[62] Co, M., Marathon. 2015, https://marathontesting.com/.

[63] Alegroth, E., M. Nass, and H.H. Olsson, JAutomate: A Tool for System- and Acceptance-test Automation, in IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST). 2013. p. 439-446.

[64] Gephi, Gephi. 2015, https://gephi.org/.

[65] Apache. ANT. 2015 [cited; Available from: http://ant.apache.org/.