On Graphical User Interface Verification

Abdulaziz Alkhalid and Yvan Labiche
Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada

Keywords: System Testing, Graphical User Interface (GUI), GUI Testing, Verification.

Abstract: Graphical User Interface (GUI) testing, for instance by means of capture and replay tools, is computationally expensive. In this paper, we present an approach for GUI verification that is not GUI (verification) testing. Using this approach, we study the input provided by an actor to the GUI and the output of the GUI to the underlying functionality. We also verify relations between those inputs and outputs. We describe the approach and discuss some first steps towards its validation in terms of fault detection using a real, though simple GUI-based software as well as a synthetic GUI-based software.

1 INTRODUCTION

GUI-based software is more and more prevalent and verifying they function as expected is therefore more and more important, especially since users are less and less willing to accept failures. Our own experience (Alkhalid and Labiche, 2017) with the verification of GUI-based software, specifically with a capture and replay tool such as GUITAR (Nguyen et al., 2014) (as a representative example of what is available in the field), indicates that capture and replay tests do not necessarily entirely exercise the application logic functionality. This may suggest that GUI testing is not functional system testing applied directly on the UI. Another observation we made is that using a capture and replay tool such as GUITAR is extremely expensive, to the point that this may not be practical on real, large-scale GUI-based software. These observations led us to rethink the verification of a GUI-based software.

In this paper, assuming a GUI-based software is designed according to the Entity-Control-Boundary (ECB) design principle (Bruegge and Dutoit, 2000), we suggest to decompose the verification of that software into the (static) verification of its UI part (i.e. no testing) combined with the (verification) testing of its application logic code at the system level (thereby bypassing the UI). For the (static) verification of the UI we rely on user-defined contracts that can be verified statically by a checker, in our case the Extended Static Checker (Flanagan et al., 2013) for Java; for the functional, system level tests, we rely on high-level functional requirements to derive tests. We evaluate our proposed solution on a real, though simple GUI-based software as well as a synthetic GUI-based software.

The rest of the paper is structured as follows. Section 2 discusses related work. Section 3 describes our solution in details. Section 4 describes an initial attempt to validate our approach. We conclude in section 5.

2 RELATED WORK

The application of static analysis is not only the use of sophisticated tools such as symbolic execution (King, 1976) using Java Path Finder (NASA, 2015) for instance, abstract interpretation (Cousot and Cousot, 1977) for instance using Julia (Spoto 2005), program slicing (Weiser, 1981) using JSlice (Wang et al., 2017) as an example or theorem proving using Extended Static Checker (ESC) (Flanagan et al., 2013) to analyze the code. Other types of code analysis, even without the use of sophisticated tools, can be considered to be static analysis. Arlt and colleagues (Arlt et al., 2012) applied static analysis of events relationships by checking the bytecode of a GUI application and its dependent libraries for GUI functional (black-box) system testing. This allowed them to infer dependencies between events. The relation is used to build an Event Dependency Graph (EDG), to select relevant event sequences among the event sequences generated from a black-box model.

Zhang et al., (2011) analyze the dependent libraries of a GUI application for test case generation.

Alkhalid, A. and Labiche, Y.
On Graphical User Interface Verification.
DOI: 10.5220/0006916603730380
In *Proceedings of the 13th International Conference on Software Technologies (ICSOFT 2018)*, pages 373-380
ISBN: 978-989-758-320-9
Copyright © 2018 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

Yuan and Memon (2007) 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 a GUI functional system testing process. Techniques similar to static analysis from the machine learning field, such as reinforced learning, have been used (Mariani et al., 2012) to discover the most relevant functionalities and to generate test cases that thoroughly sample these functionalities. This technique learns by itself 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 (Gross et al., 2012) to generate test cases at the GUI level.

3 PROPOSED SOLUTION

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. Subsection 3.1 describes the ECB design principle. Subsection 3.2 presents our definition of input-output relation for the GUI layer of a GUI-based software. Section 3.3 presents our fault model. Subsection 3.4 presents our solution. Subsection 3.5 presents our implementation of the solution.

3.1 Entity Control Boundary (ECB) Design Principle

Our work assumes that the design of the GUI-based software under verification follows the Entity-Control-Boundary (ECB) design principle which divides classes over three main categories (Bruegge and Dutoit, 2000; Bein, 2017; IBM, 2017; Pearce, 2017): 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 and determine how the software is presented to the outside world (Bruegge and Dutoit, 2000; Bein, 2017; IBM, 2017; Pearce, 2017). When the software interacts with humans, Boundary classes necessarily represent GUI classes and are implemented with well-known packages (e.g., Swing in Java). This design principle also assumes that when a Boundary class transmits requests and data back and forth between the application logic (Control classes) and actors, thereby converting it from/into a form that can be dealt with in the Control classes, it does so without changing the semantics of the information, though possibly changing the type of the information (e.g., from an `int` variable to an `Integer` object) (Nunes and Cunha, 2000), (Bruegge and Dutoit, 2000) (page 182). Our work relies on this important design assumption.

3.2 Input-Output Relation

We refer to *input variable* as any variable in the GUI code that receives a value from the user. We refer to *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: an input is received from a human actor as an input variable which, through some control flow in the UI code, reaches an argument variable. This is a way to model the flow followed by data when a Boundary class converts data received from a human actor into a form that can be dealt with in a Control class.

Our solution, which we discuss below, relies on the understanding of the multiplicities of this input-output relation. This understanding will also help during the validation of our solution. We distinguish between six different kinds of multiplicities. In the Many to One (N-1) multiplicity, several input variables to Boundary classes are used to form (i.e. compute) one argument variable to a function in a Control class. In a One to One (1-1) multiplicity, one input variable to a Boundary class becomes one argument variable to a Control method. In a Many to Many (N-M) multiplicity, many input variables to Boundary classes contribute to many argument variables. In a Many or One to zero (1-N..0) multiplicity, one or more variables do not contribute to any argument variable to the Control class. In a One to Many (1-M) multiplicity there is one input variable to the GUI that contributes to many arguments of methods in Control classes. In a Zero to one or Many (0-1..M) multiplicity, there is no input variable to the GUI but one or many arguments to Control methods.

3.3 Fault Model

A fault model assists practitioners during test case generation, and data or control flow analysis (Rajput, 2013), and allows one to qualitatively assess fault detection of a specific verification technique (Harris, 2003). We explain some faults that we expect to discover with our approach. They relate to the functional behaviour of the UI part of the software, specifically how the UI transmits data from the user to Control classes; non-functional properties of the UI are out of our scope.

The first type of faults in our fault model is an unexpected change of value of an input variable; the GUI changes the value entered by the user before passing it to a Control class when it should not; the input variable (integer, Boolean, String) is expected to equal the argument variable. The second type of faults is incorrect change in the syntax of the input. For example, the developer may split one string entered by the user (input variable) and pass it to two methods in the Control when it should be passed as two argument values to only one method. This type of faults is not a broader description of the first type because here we have a mathematical function (that is not an equality) that describes the output as a function of the input. 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 belonging to the second type.

3.4 Solution

Our solution is to study such input-output relations 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 suggest to use static analysis techniques to achieve this goal. In essence, our solution is to statically verify that the UI code does indeed implement the ECB principle correctly. And we argue that from a functional point of view, if the UI satisfies this condition and functional system-level tests applied to the application logic (i.e. Control classes) pass, then we have functionally verified the entire GUI-based software.

We have a similar handling for the different variable types. For each argument variable there must exist a mathematical function or expression that can express possible values of this variable as a function of one or more input variables. In the simplest case where an input variable is passed as is to a Control method, this mathematical function is an equality: the argument variable must equal the input variable. The next question that needs to be answered is: what is the mathematical expression, relating input variables and argument variables, and how this expression can be verified statically?

One solution is to ask the designer to specify such expressions and rely on some static analysis technology to verify that this expression is indeed true on all execution paths in the code. This however puts some burden on the designer. An alternative can be to use some advanced static analysis such as abstract interpretation and let the analyzer discover expressions that relate argument variables and input variables; it is up to the designer or tester to then validate whether this expression is correct. Again, the solution puts some burden on an engineer. Given we are working on case studies written in Java, we can investigate the use of the Java Modeling Language (JML) and the Extended Static Checker (ESC) (Flanagan et al., 2013) for implementing the first solution and the use of Julia for implementing the second solution. Regardless of the technology being used, we can expect to be subject to some technical limitations, even though the field of static analysis has made tremendous progress in the last decade. As discussed below, we opted for the first solution.

3.5 Implementation of Our Solution

Our solution is to rely on programmer-defined JML expressions that describe how argument variables ought to relate to input variables and then ask ESC to statically verify whether such expressions always hold. Because we are interested in argument variables, that is variables that are passed to methods of Control classes, these expressions are necessarily pre-conditions. Let us assume that the GUI is supposed to present a numerical input as is to a Control class. With a static checker, we can check if there is a difference in value, as specified in a JML precondition, between the input variable to the GUI and the argument passed to the Control class. 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 the static checker has revealed a fault. If the static checker does not report any violation (difference), then there is no fault. This way, the static checker is used for fault detection in the GUI. In order to specify such preconditions, a few additional questions must be answered: (1) Where to place the pre-condition? (2) how to make the input variable (input by the user) available/visible to that precondition so that the mathematical expression specifying the input-output

relation can be specified as a JML expression? Since the mathematical expression/function specifies a relation involving an argument variable, the precondition of course specifies the Control class method that receives that argument variable as input. This method can either be a constructor or a "regular" method of the Control class.

Given that we assume we work on a GUI-based software built with the Java programming language, we must abide to the visibility rules of variables and attributes in Java when answering the second question. Since the input variable is defined in a Boundary class, and Boundary and Control classes are different, since this input variable may (likely) be a local variable in a Boundary class method, this variable is not visible to a precondition in a Control class method. We therefore add public static variables in the GUI classes to hold the value entered by the user (the input variables). Then, we must search for places in the Boundary classes where calls (methods or constructors) are made to Control classes, analyze which (local) variables in the calling methods are used as arguments in those calls, and identify which user inputs are used to set values of those arguments. This is a standard data flow problem that can be easily solved with a tool such as Atlas (Kothari, 2017). Once this is done, we set the value of the added static variable to the variable (user input) that has been identified.

3.6 A Simple Example

We use a simple, synthetic software to explain the solution described above. The software was 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.

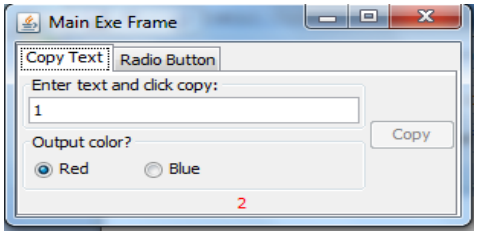


Figure 1: The main window of simple software after typing the input and clicking the copy button.

Figure 1 shows the main window of the software. When the user enters an input, using the text field, and presses the copy button, the software shows the result at the bottom of the window with the user-selected colour (radio buttons). The figure illustrates a fault: the software added 1 to the input value, which should have been 1, not 2.

Figure 2 shows a sample of the code from class `CopyTextComponent`, specifically an excerpt of the `actionPerformed()` method of the `CopyTextComponent` instance. This 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: this is the fault. Then it passes it to the Control class instance: call to the constructor of the `TextControl` class. Then, the call proceeds to the method `printCopyMessage()`.

```
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 adding 1.
3 TextControl t = new
    TextControl(Integer.toString(callArgumen
    t));
    // passing the value to a Control
4 t.printCopyMessage(evt.toString(),0);
```

Figure 2: Excerpt of the GUI code.

In this example, 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 value by one. The code converts the value (after the increment) to a string and passes the string to the Control class instance. Our procedure, discussed earlier, is to identify the call site to the Control class (here it is line 3 in Figure 2), and identify the input variable that leads to an argument used in the call site (here it is variable `userInput` that receives a value at line 1 in Figure 2). As mentioned earlier this can be facilitated by the use of a tool such as Atlas. Our procedure is then to add a public static variable to record the input value: here we record the value received by `userInput`. Figure 3 shows the modified `CopyTextComponent`. We define a public static variable in class `CopyTextComponent` called `verification_variable`. (Code showing this addition omitted.) The variable is used to store the value entered by the user: the same value as `userInput`. We also add a local variable

output_variable (line 5) to contain the data that is passed to the Control class. Adding this local variable will help ESC analyze the code. Our experiment shows that without such a local variable ESC does not evaluate this part of code.

```
1 Sting s = inputExpression.getText();
2 int userInput=Integer.parseInt(s);
    // get the user input from the text
    field
3 int callArgument = userInput + 1;
    // process the entered value by adding
    1.
4 verfication_variable = s;
5 output_variable=
    Integer.toString(callArgument);
6 TextControl t = new
    TextControl(output_variable);
    // passing the value to a Control
7 t.printCopyMessage(evt.toString(),0);
```

Figure 3: Modified Boundary code as per our solution.

The next step of our solution is to add a precondition before the constructor of TextControl since it is the constructor that is called in the Boundary class (CopyTextComponent): Figure 4.

```
1 /* process equality check on the entered
    value and value that reach the
    Control.
2 // @ requires s ==
    CopyTextComponent.verificaton_variable
3 */
4 TextControl(String s){ ...
```

Figure 4: The adding of a precondition in the source code.

```
ParseException{CopyTextComponent.java:109:
Warning: Precondition possibly not
established (Pre)
TextControl t = new
TextControl(output_variable);
^
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.
```

Figure 5: Excerpt of the file generated by ESC.

The precondition specifies that it is required, when entering the Control class constructor, that the value of argument s (the output variable) equals the value of the public static attribute value we added to record the input variable. Then, ESC, when analyzing the modified code, generates an output file: excerpt in Figure 5. ESC reports a violation of the precondition. (Note that line numbers reported by ESC in the figure do not match the lines numbers we have in the figures of this paper; this is because we re-numbered the lines in the paper to simplify the discussion.)

4 VALIDATION

We first discuss validation with synthetic examples in light of the input-output relations, and their multiplicities, that we discussed earlier, and then validation with a real, though simple, GUI-based software in terms of fault detection.

4.1 Applicability

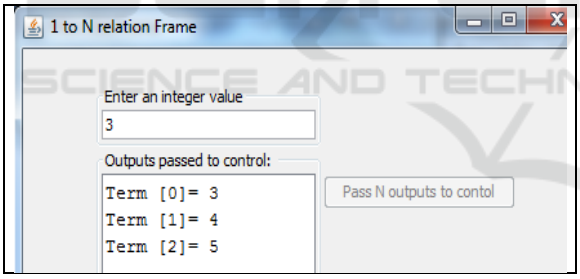
In this section, we validate the applicability of our solution on several types of multiplicity. Hence we argue that our approach will work on those scenarios. Consequently, we hope that such validation shows the capability to generalize our results. We use a synthetic case study to demonstrate how our verification technique is applicable for several multiplicities of the input-output relation. We use faulty preconditions instead of generating faults in the software. There are three buttons in this case study, 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 where the user can enter a text and clicks a button. 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. The GUI has a label that shows a text "String passed as is to control" at the bottom of the window. To verify this 1-1 relation, we proceeded as described earlier (added static variable, added precondition): the precondition is an equality between two variables (the static variable of the Boundary class, the argument of the Control method). With this precondition, ESC does not complain. However, if we change the precondition (e.g., inequality instead of equality), thereby simulating a fault in the Boundary class, then ESC warns about a violation of the precondition.

Figure 6 shows the UI of an example of 1-N relation and the corresponding JML precondition. In this example the UI receives an integer value as an input; 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. We use another example to

simulate a N-1 relation. In this synthetic example, the software receives several values as inputs. Then, it outputs the sum of those values. Similarly to the previous cases, we added static variables (for recording the values of input variables) and a JML precondition: the argument of the Control method should be the sum of inputs to the GUI. Details are not shown because of space constraints. We also simulated faults in the GUI with alternative JML preconditions, i.e., other than the correct one; each time ESC complained as it detected violations of the alternative preconditions whereas it was able to confirm the correct precondition was always satisfied.

These examples show that our approach to verify the input-output relation between the UI (Boundary classes) and the application logic (Control classes) applies to several kinds of multiplicities. We think that the three multiplicities discussed above (1-1, 1-N and N-1) are the most common types of multiplicities. Hence, we believe that our solution will work on all other types of multiplicities discussed in section 3.2, though more studies to confirm this assertion are warranted. We think that this would help us to generalize our solution.

4.2 Fault Detection




```
15 /*@ public normal_behavior
16    @ requires a + b + c == 3 + (3*
OneToManyComponent.verification_user_input);
17 @*/
18 public OneToManyControl (int a, int b,
int c) …
```

Figure 6: JML code to verify the 1-N relation.

We first discuss the software under test and then the experiment. Results are discussed last. Our software under test performs a number of computations on Boolean expressions provided by the user through a GUI. The Software Under Test (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 testing strategy for a Boolean expression (Weyuker et al., 1994). Our case study has one main window with three buttons. Clicking on any of the buttons generates a child window to handle the corresponding functionality. The truth table window accepts a string representing a Boolean expression (text field) and generates its truth table upon request (`Compute Truth Table` button). The DNF Expression window 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. In the third window 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 text fields, one term per text field.

First we formulated three preconditions for our case study because each functionality involves only one call to a Control class method. For each of these preconditions, we ran ESC twice: one time using a valid precondition and another time using a modified (incorrect) precondition. Hence, we simulate that ESC reports on issues when there are some (modified precondition) and is silent when there is no issue (original precondition). In our case study, there is one argument variable to one truth table Control class method, one argument variable to one DNF Control class method, and three or more argument variables (in fact it is an array) to one variable negation Control class method.

For the truth table window, there is only one text input that is the Boolean expression. In this example, there is no intermediate variable between the GUI and the Control. In other words, the value is taken from the GUI and passed directly to the Control. In fact, this Control class is called by all three functionalities / user interfaces. Its precondition must reflect that. For the DNF component functionality, the situation is similar as we need to handle only one input in the GUI of the DNF. We proceeded similarly to the previous case: addition of a local variable, addition of public static variable, JML precondition checking the equality between the static variable and the argument of the constructor of the control class. As for the variable negation component, the situation is different as there are three inputs by default from the GUI and the user can add more inputs. The input-output relation in this GUI is an example of a N-1 input-output relation. The user inputs are used to create an array that is passed to the Control; each element in the array is a string coming from a text field. We need to define some verification (public static) variables. We should define those static variables at specific points of the code. We had to

make a few more code modifications for this third functionality compared to the previous cases. We defined two arrays; adding arrays is simply due to the fact that the data we need to record (input variables) is an array; in previous cases, the data was a simpler type. The first array is to store the verification variables. We pass the second array to the Control class. The original code passes an iterator of values to the Control class. In the precondition of the Control class, we have to inspect every item inside the iterator as a verification procedure. Since dealing with an iterator is not as easy as dealing with arrays in JML, we instead look for a place in the application logic code (Control) where the elements of the array/iterator can be checked. This does not happen in the constructor but in a method immediately called by the constructor. To perform the verification, we added an "abstract" method (empty body) and specified the required verification in JML as its precondition.

For fault detection, similarly to the previous case, we asked ESC to verify correct and incorrect preconditions. 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. This is much less than the time consumed by GUITAR. GUITAR tests the underlying functionality only if we update the test cases with valid inputs and takes approximately 20 minutes to replay a test suite on the case study we used here regardless if it is updated or not (Alkhalid and Labiche, 2017). We therefore argue that 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 a much more effort-consuming task than adding verification variables and preconditions to the code.

5 CONCLUSION

Recognizing that GUI testing in the sense of executing system level functional tests through the GUI is expensive and likely leads to redundant testing efforts, we investigate the use of static analysis to verify the GUI part of the GUI-based software. We describe the notion of input-output relation at the GUI level that is data flow relationships that exist between data provided by the user to the UI and data provided by UI to the application logic code. We use this notion of input-output, describing various multiplicities of the relation, to structure the discussion of our solution (including a fault model). We then present an approach to statically verify how the GUI implements those relations, without testing. We implement the approach using the Extended Static Checker (ESC) that relies on Java Modeling Language preconditions that specify the input-output relations to be verified. In essence our approach assumes the UI is designed by following the Entity-Control-Boundary (ECB) design principle: class responsibilities lead to Entity classes (hold the data, the state of the software), Boundary classes (interacting with actors), and Control classes (realizing use cases); Boundary classes may change the syntax of data they collect from actors and pass to Control but not their semantics. The solution is a verification that the ECB principle holds: the JML preconditions being checked by ESC specify what it means for the UI to not change the semantics of the data.

Our solution overcomes other solutions of GUI functional system testing, at least in principle (we have not made any experimental comparisons). We made steps towards the validation of our solution. In the evaluation of our work, instead of a faulty software, we decided to use faulty preconditions. Hence, we run the ESC twice, once on a correct precondition and another time on an incorrect one. In doing so, we argue we simulate a fault in the code. Our results, though limited by the size of the GUI-based software we used, as well as some limitations of ESC, looks encouraging.

Our future work includes the evaluation of our solution on other case studies, as well as the use of more advanced static checkers than ESC that would overcome ESC's limitations. Theoretical analysis and experimental evaluation should also validate whether mutating preconditions actually simulates faults in the functional behaviour of the UI code in our context.

ACKNOWLEDGEMENTS

This research has been funded by the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

Alkhalid, A. and Y. Labiche (2017). How does GUI testing exercise application logic functionality? *2017 IEEE 41st COMPSAC*. Torino, IEEE. 2: 90-95.

Arlt, S., A. Podelski, et al. (2012). Lightweight static analysis for GUI testing. *2012 IEEE 23rd International Symposium on Software Reliability Engineering*, IEEE.

Bein, A. (2017). "Simplicity by Design." Retrieved 2017, from http://www.oracle.com/technetwork/issue-archive/2011/11-jan/o11java-195110.html.

Bruegge and Dutoit (2000). "Object-Oriented Software Engineering: Using UML, Patterns and Java."

Cousot, P. and R. Cousot (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, ACM.

Flanagan, C., K. R. M. Leino, et al. (2013). "PLDI 2002: Extended static checking for Java." A*CM Sigplan Notices 48(4S)*: 22-33.

Gross, F., G. Fraser, et al. (2012). Search-based system testing: high coverage, no false alarms. *Proc. of the 2012 International Symposium on Software Testing and Analysis*, ACM.

Harris, I. G. (2003). "Fault models and test generation for hardware-software covalidation." *IEEE Design & Test of Computers* 20(4): 40-47.

IBM. (2017). "Guideline: Entity-Control-Boundary Pattern." Retrieved 2017, from http://epf.eclipse.org/wikis/openup/core.tech.common.extend_supp/guidances/guidelines/entity_control_boundary_pattern_C4047897.html.

King, J. C. (1976). "Symbolic Execution and Program Testing." *Communications of the ACM*

Kothari, S. (2017). "Atlas." Retrieved 2017, from http://www.ensoftcorp.com/atlas/.

Mariani, L., M. Pezze, et al. (2012). Autoblacktest: Automatic black-box testing of interactive applications. *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, IEEE.

NASA. (2015). "Java path finder." Retrieved 2015, from http://babelfish.arc.nasa.gov/trac/jpf/wiki.

Nguyen, B. N., B. Robbins, et al. (2014). "GUITAR: an innovative tool for automated testing of gui-driven software." *Automated Software Engineering* 21(1): 65-105.

Nunes, N. J. and J. O. F. O. Cunha (2000). Towards a UML profile for interaction design: the Wisdom approach. *International Conference on the Unified Modeling Language*, Springer.

Pearce, J. (2017). "The Entity-Control-Boundary Pattern." Retrieved 2017, from http://www.cs.sjsu.edu/~pearce/modules/patterns/enterprise/ecb/ecb.htm.

Rajput, D. (2013). "Fault Models and Test Generation for Covalidation Techniques in Hardware & Software" *Advance in Electronic & Electric Engin.* 3(7): 817-826.

Spoto, F. (2005). Julia: A generic static analyser for the java bytecode. *The 7th Workshop on Formal Techniques for Java-like Programs, FTfJP'2005*, Glasgow, Scotland, July 2005. Available at www.sci.univr.it/~spoto/papers.html.

Wang, T., A. Roychoudhury, et al. (2017). "JSlice - a Java Dynamic Slicing Tool."

Weiser, M. (1981). Program slicing. *Proc. of the 5th International Conference on Software Engineering*, IEEE Press.

Weyuker, E., T. Goradia, et al. (1994). Automatically generating test data from a Boolean specification. *IEEE Transactions on Software Engineering*. 20: 353-363.

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

Zhang, S., D. Saff, et al. (2011). Combined static and dynamic automated test generation. *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ACM.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS