Closing Gaps between Capture and Replay: Model-based GUI Testing

Oliver Stadie and Peter M. Kruse

Berner & Mattner Systemtechnik GmbH, Berlin, Germany,
{oliver.stadie|peter.kruse}@berner-mattner.com

Abstract. Testing software as a black box can be time consuming and error-prone. Operating and monitoring the graphical user interface is a generic method to test such systems. This work deals with convenient and systematic testing of GUI software systems. It presents a new approach to model-based GUI testing by combining the strengths of four well-researched areas combined: (1) the intuitive capture&replay method, (2) widget trees for modeling the GUI, (3) state charts and (4) the classification tree method. The approach is implemented as a prototype and is currently under validation on a real GUI. The presented approach includes the whole test cycle, from scanning the GUI and model-based test specification to the automatic execution of tests.

Keywords: Automated GUI Testing, Systematic GUI Testing, Model-based Testing, Classification Tree Method, State Chart, Capture&Replay

1 Introduction

There are many approaches, how to test software. Today, many software systems provide a graphical user interface (GUI) to ease the users to access these systems. When testing these software systems, the GUI can be used to test the software from the user-perspective (black box testing) or to test the GUI itself. In current systems, the GUI takes up to 60 percent of the total source code [1]. Testing generally causes 50 percent of the total costs of software development [2]. By automating GUI tests, up to 80 percent of the costs could be saved compared to manual GUI testing [3].

Especially for regression testing, a major problem is that changes to the GUI should not require manual steps in order to adapt the test [4].

Semi and fully automatic methods have been published, in order to simplify the GUI testing process [3, 5]. Nevertheless, the approach most widely used is still capture&replay [6]. This gap between explored and used methods could be due to a lack of intuitiveness, learning and lack of tool support.

In this work we develop a method to support the GUI testing process by combining existing methods. The developed method is to be implemented in a tool that can be used for any type of GUI, regardless of the underlying technology. Specifically, the following methods and models are used: the *capture&replay method* [7], *widget trees* [8], *state charts* [9] (esp. UML state diagrams), and the *classification tree method* [10].

Table 1. Combined Methods and Models in This Work

Method	Advantages	Disadvantages
Capture&Replay [7]	- Intuitive Usability - Widespread - Quick and Easy Specification of Individual Sequences - Simple Means to Scan the GUI	- High Maintenance Costs - Low Stability against Changes
Widget Trees [8]	- Detailed Modeling of GUI States - Convenient Management and Overview by Hierarchies	- Stability to Changes Uncertain
State Charts [9]	- Modeling and Selection of System Behavior (Sequences) - Relatively Stable to GUI Changes - Easy to Learn	- Difficult Automated Construction
Classification Tree Method [10]	- Classification of the Input Data Space, Reduction of the Necessary Test Cases - Systematic Derivation of Test Cases - Established in Practice - Suitable for Functional Black Box Testing	- Can be Too Large for Complex Systems *(Splitting into Several Trees Circumvents the Problem)*

These methods are to be combined in the following sections, with the aim to obtain as many of the benefits as possible and to eliminate as many of the disadvantages as possible (Table 1).

2 Background

The widespread capture&replay tools work as follows: The tester records a manually executed sequence of actions on the GUI. This is the capture phase. Then the recorded sequence can automatically be executed on the GUI repeatedly. That's the replay phase. One advantage of capture&replay tools is that they can be easily learned and used. A draw-back is that they are not inherently systematic, so the quality of the recorded test depends of the skills of the tester [7].

Memon pays attention to both the importance and the lack of GUI test methods [7]. Methods used are often unsystematic, ad hoc or too expensive.

Surveying recent works related to model based GUI testing [11–13], reveals that there are two dominant methods for modeling GUIs. One common approach in model-based testing are state charts [5]. A second approach is the model of Memon et al. consisting of *GUI forest*, *event-flow graphs* and *integration tree* [14].

Widget Trees are another approach to modeling of GUI-states [8]. Widget trees focus on the modeling of all elements in the widget hierarchy, while state charts model the behavior and possible navigation paths through the system.

A systematic method for test specification is the classification tree method [10]. TESTONA is a test tool that implements the classification tree method [15, 16].

3 Approach

The developed method supports the tester in testing a model-based GUI. The method forms a cycle, which (1) starts and analyzes the GUI, it then (2) creates GUI models, from these (3) derives test sequences and ultimately (4) executes these sequences again on the GUI (Figure 1).

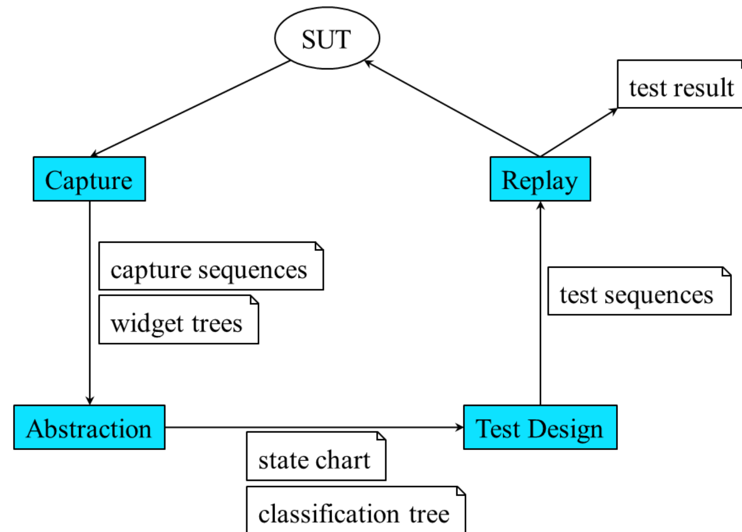

Fig. 1. Workflow

1. The tester initiates the capture process part. The tester performs to be recorded sequences, by using peripherals[1] (mouse and keyboard), on the system under test (SUT).
2. The inputs in the periphery and the output of the SUT can be observed and models of GUI are created and updated from it. After the tester completes a sequence, they can restart the capture process any number of times to record additional sequences.
3. The created GUI model is presented to the tester for test sequence generation.
4. Finally, the tester triggers the test sequence execution. The SUT is started automatically for each test sequence and manipulated automatically via generated, simulated peripheral inputs.

3.1 Capture

The first phase of the process ensures the creation of capture sequences. After starting the SUT, its GUI is scanned to determine its initial state. The tester makes any number

[1] Device used to put information into or get information out of the computer. Also called input/output device. [17]

of inputs on the keyboard and mouse during each recording. Each of these inputs affects the SUT. Thereafter, the method links the input to the last recorded GUI state and reads the new GUI. Each GUI scan is stored as a widget tree.

Stabilization: For all widgets in the widget tree, the details are reduced and only names and types are kept, resulting in a discarding of e.g. widget dimensions, pixel positions, colors, IDs (similar to [18]). This is done for increased robustness, esp. in regression testing. The interrelation of widgets is only maintained using their location in the widget tree.

Merging: Multiple similar user actions without consequences to the widget tree can be merged, e.g. typing several single letters into a textbox or moving around the mouse without actually clicking (assuming there are no interactive reactions cause by the typing and no hover-reactions of traversed GUI elements). This is done do limit state explosion.

Each capture sequence contains a (merged) chain of traversed states and transitions of the SUT. Each transition has an atomic action as its trigger. Each state consists of a stabilized widget tree. The model of capture sequences is a non-empty set of capture sequences, which in turn are modeled as traversed states and actions performed.

3.2 Abstraction of Models

After capturing has been completed, abstraction of GUI Models is performed. The capture sequences lack a relationship between the sequences and their branches. Therefore, we use the following heuristics to merge the capture sequences:

Treat equal what looks equal: Since operation is performed on stabilized widget trees, the algorithm merges sequences so that equal looking steps (containing widgets with same name and type) are merged into single states of the resulting state chart.

Hierarchy: To create state chart hierarchies, a state on the first level of the state chart is created for each modal window. Each window state has a set of sub states for different application modes. All widget trees from all capture sequences with the same type structure (discarding all other properties, such as name) are merged into single application modes (all on the second level of the state chart). Each application mode state contains a set of sub states, derived from the text of the widgets in the widget tree. So we use the similarity in widget tree structures for creation of states on second level and the differences in widget tree properties (especially widget text) for creation of states on third level of the state chart.

Concurrency: Orthogonality is created using a set of predefined widgets, such as the main menu and pop-up menus. Once a menu is used in any sequence recorded, it is considers always accessible, independent of actual access in recorded sequences. The behavior of each such menu is modeled in its own orthogonal region in the state chart.

The first and the third heuristic here increase the possible number of variations and permutations in later test design. In contrast to the plain playback of linear sequences in conventional capture&replay. This generalization might however lead to non-executable sequences.

Each GUI-model consists of a set of widget trees, a state chart and a classification tree created from the state chart (as described in [19]). Each widget tree is assigned to exactly one state in the state machine.

3.3 Specification of Test Sequences

In the third phase, test sequences are specified. First, the classification tree part of the GUI model is used to identify test sequences and—as in the ordinary classification tree method [19]—described in a test matrix. Each sequence also represents a path through the state chart. The state machine is used to constrain possible test sequences. The sequences determined meet coverage criteria such as state coverage or path coverage.

Each test sequence defines a sequence of states to reach. Since the state chart has a higher abstraction level than the captured sequences, non-caputured sequences may occur here. The handling of infeasible paths in sequences is not yet automated and therefore left to the tester. The required actions to traverse the states are defined in the state chart. As such, events carry those input values (e.g. for text fields) that were recorded in the initial capture phase. The tester can however adopt these as part of test specification.

3.4 Test Execution

In the fourth phase, the previously specified test sequences are executed automatically. At this stage, all given test sequences are treated sequentially. At the beginning of each test sequence, the SUT will be started automatically. Similar to [18], it is required that the SUT always starts into the same initial state. The tester needs to take care of this (e.g. by resetting the SUT preferences prior test execution).

Each test step is then processed from each test sequence. First, the GUI is scanned to identify its widget tree. The to-use widget is searched in the widget tree. The action to be carried out by operation of the peripherals is then simulated. The tester can define a delay time between execution steps. Otherwise all events are fired as fast as possible, potentially leading to the SUT not receiving all events.

After all steps of a test sequence are performed, the SUT is terminated. After completing all the test sequences, this phase ends.

Simple test results can be produced here by comparing the actual with the expected widget trees after each step. This also includes reporting whether each action could be performed.

4 Evaluation

The developed solution has been implemented as a plug-in for TESTONA[2] and currently works for all GUIs in Windows operating systems (Figure 2). To this end, several existing works—as frameworks and libraries—have been reused.

The approach used here conforms to a general structure for GUI tests [20]: A testing framework consisting of a technology-independent GUI model and the general test sequence specification were already implemented as XML specifications.

The implementation of the developed method is currently under evaluation quantitatively and qualitatively for several GUI systems. The results of evaluation are intended

[2] http://www.testona.net

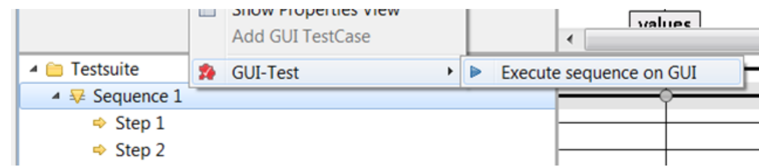

Fig. 2. Execution of Tests

to provide indications of the practicality of the developed test design process and about the quality of the prototype.

We have obtained some first results. In our approach widgets are described in terms of their name and type. This choice is sufficient for inferring a good abstraction of the model in the Windows applications tested, as for example most of the time button names were similar to their caption. This might be due to the Windows API or simply be good design of tested applications (Windows Calculator, TESTONA tool itself). With a growing body of applications under test, we might need to consider different stabilization rules.

Currently, keyboard and mouse inputs by the tester are the only

Table 1 lists the four methods used in our approach with their individual strengths and weaknesses. We will now evaluate, whether the combination of methods helps to overcome weaknesses without sacrificing the strengths.

4.1 Capture&Replay

A general problem with capture&replay are the high maintenance costs due to low stability of captured sequences against changes. By using widget trees and by scanning the GUI during test execution, our approach reduces the maintenance costs with increased stability against changes of the GUI. Details on cost reduction and on how stable the approach actually is, have not yet been provided due missing evaluation in detail.

The advantages of capture&replay are all preserved. Our combined approach also relies on the intuitive usability. The recorded individual sequences are, however, used to abstract a general GUI model, which allows more variation in later test specification.

4.2 Widget Trees

The stability of widget trees against changes in the application has not yet been assessed. Without a large-scale evaluation, we cannot yet overcome this problem.

The advantage of widget trees, both detailed modeling of GUIs and the introduction of hierarchies for better overview, are both kept.

4.3 State Charts

The construction of state charts is a challenging task. By introduction of heuristics provided, this weakness is completely resolved.

The strengths of state charts are all maintained. The influence of GUI changes to the stability of state charts have not yet been verified.

4.4 Classification Tree Method

By only including relevant parts of the state chart to the classification tree, the size of trees is kept small.

The mentioned advantages are all preserved. The automated generation of test sequences is considered helpful.

5 Related Work

Bauersfeld and Vos also implement a tool for testing GUI system: GUITest [18]. Their tool provides the following features, also present in our implementation: *a)* Works on all native GUIs, which are recognized by the Windows API. *b)* SUT must not be instrumented. *c)* Allows the user to define their own actions. *d)* Generated Test sequences can be stored and played back.

In contrast to GUITest our tool offers the following features: *a)* GUITest specializes in robustness tests. That is, it searches automatically for random test sequences through the GUI, without necessarily representing realistic or target-oriented user behavior. The point is to find errors. Our implementation is especially useful for functional testing. The aim here is to test if specific requirements are met and the SUT fulfills its intended purpose. *b)* Compared to GUITest our implementation displays the model of GUI and test sequences.

Memon et al. also offer an implementation—similar to this work—for model-based GUI testing, with prototypical capture, semi-automatic modeling and automated execution [14]. While Memon et al. model the SUT with GUI forests, event-flow graphs and integration trees, this work uses state chart, widget trees and classification trees.

We assume, that state charts are more suitable, because they are more compact due to hierarchies and orthogonality. Test models intended for end-user (Tester) presentation should be understandable. In this case, state charts might be better, esp. when dealing with self-transitions, which can occur in GUIs. However, state charts are not considered better *per se*.

Nguyen et al. combine state charts with the classification tree method [21]. They choose paths on the state chart, representing abstract test sequences. For each of these paths, they construct a classification tree. With these trees several specific test sequences are specified for each path. Nguyen et al. see the strength of the state charts in the specification and selection of sequences (consecutive events) and the strength of the classification tree method in the selection of specific input parameters and a meaningful reduction of the input parameters.

In the context of web applications, there are similar approaches [22, 20]. While this also is a challenging field, our work focuses on native applications outside the browser.

6 Conclusion

The developed tool enables the user to comfortably create GUI models by capturing. GUI models are then used for systematical test design in terms of the classification tree

method. Resulting test scenarios can be automatically executed on the SUT. Such scenarios allow to test the GUI itself or to use the GUI for black-box testing the underlying system.

Despite the problems worked out the combination of the four methods *capture&replay*, *widget trees*, *state charts*, and *classification trees* seem to be much-promising and suitable for the testing of GUI systems. Weaknesses of single methods were overcome without accepting many sacrifice of their strengths. Many of the problems identified will be addressed in future work.

Future work will also concentrate on a large scale evaluation and comparison with capture&replay tools in terms of efficiency and effectiveness considering both, initial creation and maintenance efforts.

References

1. Brad A Myers. User interface software tools. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 2(1):64–103, 1995.
2. Frederick P Brooks. *The mythical man-month*, volume 1995. Addison-Wesley Reading, MA, 1975.
3. Michael Turpin. Survey of gui testing processes. 2008.
4. Atif M Memon and Mary Lou Soffa. Regression testing of GUIs. *ACM SIGSOFT Software Engineering Notes*, 28(5):118–127, 2003.
5. Imran Ali Qureshi and Aamer Nadeem. Gui testing techniques: A survey. *International Journal of Future Computer and Communication*, 2(2), 2013.
6. Stephan Arlt, Cristiano Bertolini, Simon Pahl, and Martin Schäf. Trends in model-based gui testing. *Advances in Computers*, 86:183–222, 2012.
7. Atif M Memon. Gui testing: Pitfalls and process. *Computer*, 35(8):87–88, 2002.
8. Sebastian Bauersfeld, Stefan Wappler, and Joachim Wegener. A metaheuristic approach to test sequence generation for applications with a gui. In *Search Based Software Engineering*, pages 173–187. Springer, 2011.
9. David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.
10. Matthias Grochtmann and Klaus Grimm. Classification trees for partition testing. *Softw. Test., Verif. Reliab.*, 3(2):63–82, 1993.
11. Valéria Lelli, Arnaud Blouin, Benoit Baudry, and Fabien Coulon. On model-based testing advanced GUIs. In *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*, pages 1–10. IEEE, 2015.
12. Pekka Aho, Matias Suarez, Atif Memon, and Teemu Kanstrén. Making GUI testing practical: Bridging the gaps. In *Information Technology-New Generations (ITNG), 2015 12th International Conference on*, pages 439–444. IEEE, 2015.
13. Tanja EJ Vos, Peter M Kruse, Nelly Condori-Fernández, Sebastian Bauersfeld, and Joachim Wegener. TESTAR: Tool support for test automation at the user interface level. *International Journal of Information System Modeling and Design (IJISMD)*, 6(3):46–83, 2015.
14. Atif M Memon, Ishan Banerjee, and Adithya Nagarajan. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 260–260. IEEE Computer Society, 2003.
15. Peter M Kruse and Magdalena Luniak. Automated test case generation using classification trees. *Software Quality Professional*, 13(1):4–12, 2010.

16. Eckard Lehmann and Joachim Wegener. Test case design by means of the CTE XL. *Proceedings of the 8th European International Conference on Software Testing, Analysis and Review (EuroSTAR 2000), Kopenhagen, Denmark, December*, 2000.
17. Philip A Laplante. *Dictionary of Computer Science, Engineering and Technology*. CRC Press, 2000.
18. Sebastian Bauersfeld and Tanja EJ Vos. Guitest: a java library for fully automated gui robustness testing. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 330–333. ACM, 2012.
19. Peter M Kruse and Joachim Wegener. Test sequence generation from classification trees. In *Proceedings of ICST 2012 Workshops (ICSTW 2012)*, Montreal, Canada, 2012.
20. Peter M Kruse, Jirka Nasarek, and Nelly Condori Fernandez. Systematic testing of web applications with the classification tree method. In *Proceedings of the XVII Iberoamerican Conference on Software Engineering (CIbSE 2014)*, 2014.
21. Cu D Nguyen, Alessandro Marchetto, and Paolo Tonella. Combining model-based and combinatorial testing for effective test case generation. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 100–110. ACM, 2012.
22. Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana. Rich internet application testing using execution trace data. In *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, pages 274–283. IEEE, 2010.