Automated GUI Testing Validation guided by Annotated Use Cases

Pedro Luis Mateo Navarro, Diego Sevilla Ruiz, Gregorio Martínez Pérez
{pedromateo,dsevilla,gregorio}@um.es
Departamento de Ingeniería de la Información y las Comunicaciones
Departamento de Ingeniería y Tecnología de Computadores
University of Murcia, 30.071 Murcia, Spain

Abstract: This paper presents a new approach to Automatic GUI Test Case Generation and Validation: a use case-guided technique to reduce the effort required in GUI modeling and test coverage analysis. The test case generation process is initially guided by use cases describing the GUI behavior, recorded as a series of interactions with the application widgets (e.g. widgets being clicked, data input, etc.) These use cases (modeled as a set of initial test cases) are annotated by the tester to signal interesting variations in widget values (ranges, valid or invalid values) and validation rules with expected results. Annotations and validation rules allow this approach to automatically generate new test cases and expected results, easily expanding the test coverage. Also, the process allows narrowing the GUI model testing to precisely the set of widgets, interactions, and values the tester is interested in.

Keywords: GUI Testing, Model Based Testing, Test Case Auto Generation, GUI Verification

1 Introduction

It is well known that testing the correctness of a GUI (*Graphical User Interfaces*) is difficult for several reasons [XM06]. One of those reasons is that the space of possible interactions with a GUI is enormous, which leads to a large number of GUI states (a related problem is to determine the coverage of a set of test cases); the large number of possible GUI states results in a large number of input permutations that have to be considered. Another one is that validating the GUI state is not straightforward, since it is difficult to define which objects and properties have to be verified.

This paper describes a new approach between Model-less and Model-Based Testing approaches which describes a GUI Test Case Auto-generation process based on a set of use cases (which describe the GUI behavior) and the annotation (values, validation rules, etc.) of the relevant GUI elements.

The rest of the paper is structured as follows. Related work is presented in section 2. In section 3 we describe the new testing approach. Annotation, auto-generation and execution/validation processes are described in sections 4, 5, and 6 respectively. Finally, section 7 provides conclusions and lines of future work.

2 Related work

Model Based GUI Testing approaches can be classified depending on the amount of GUI details that are included in the model. By GUI details we mean the elements which are chosen by the *Coverage Criteria* to faithfully represent the tested GUI (e.g. window properties, widget information and properties, GUI metadata, etc.)

Many approaches usually choose all window and widget properties in order to build a highly descriptive model of the GUI. For example, in [XM06] (Xie and Memon) and in [MBN03, MBHN03] (Memon, Banerjee, and Nagarajan) it is described a process based on GUI Ripping and the construction of Event-Flow Graphs (EFG) and Event Interaction Graphs (EIG) which include all the elements and events that can be found in a GUI. Once the model is built, the process generates automatically all the possible test cases.

As said in [YM07], the primary problem with these approaches is that as the number of GUI elements increases, the number of event sequences grows exponentially. Another problem is that the model has to be verified, fixed, and completed manually by the testers, being this a tedious and error-prone process itself, decreasing also the modification tolerance.

Other approaches use a more restrictive coverage criteria in order to focus the test case auto-generation efforts on only a section of the GUI which usually includes all the relevant elements to be tested. In [VLH$^+$06] Vieira, Leduc, Hasling, Subramanyan, and Kazmeier describe a method which uses enhanced UML Diagrams to describe the behavior of the GUI. Then, an automated process generates test cases from these UML diagrams. The diagrams are enriched in two ways: first, the UML Activity Diagrams are refined to improve the accuracy; second, these diagrams are annotated by using custom UML Stereotypes representing additional test requirements. In [PFV07] Paiva, Faria, and Vidal also describe a UML Diagrams based model. In this case, however, the model is translated to a formal specification.

The scalability of this approach is better than the previously mentioned because it focuses its efforts only on a section of the model. The diagram refinement also helps to reduce the number of generated test cases. On the other hand, some important limitations make this approach not so suitable for certain scenarios: the building, refining, and annotation processes require a considerable effort since they have to be performed manually, which does not suit some methodologies such as, for instance, Extreme Programming; these techniques also have a low tolerance to modifications; finally, testers need to have a knowledge of the design of the tested application (or have the UML Model), which makes impossible to test binary applications or applications with an unknown design.

3 Overview of the Annotated Use Case Guided Approach

In this paper, a new GUI Testing approach is introduced, based on a Test Case Auto-generation process that does not build a complete model of the GUI. Instead, it models two main elements:

- A set of Use Cases: these use cases describe the behavior of the GUI to be tested.
- A set of Annotated Elements: this set include the GUI elements whose values may vary and those with interesting properties to validate.

With these elements, the approach addresses the needs of GUI verification, since, as stated in [VLH$^+$06], the testing of a scenario can usually be accomplished in three steps: launching the GUI, performing several use cases in sequence, and exiting. The approach combines the benefits from both "Smoke Testing" [MBHN03, MX05] and "Sanity Testing" [ZKP$^+$08], as it is able to test the main functionality (in the first steps of the development process) and fine-tune checking (in the final steps of the development process) by an automated, script-based process.

The *test-case generation process* described in this paper takes as its starting point the set of use cases (a use case is a sequence of events performed on the GUI; in other words, a use case is a test case). From this set, it creates a new set of auto-generated test cases, taking into account the variation points (according to possible different values of widgets) and the validation rules included in the annotations. The resulting set includes all the new auto-generated test cases.

This process can be seen, in a test case level, as the construction of a tree (which initially represents a test case composed of a sequence of test items) to which a new branch is added for each new value defined in the annotations. The validation rules are incorporated later as validation points.

Therefore, in our approach, modeling the GUI and the application behavior does not involve building a model including all the GUI elements and generating a potentially large amount of test cases exploring all the possible event sequences. In the contrary, it works by defining a set of test cases and annotating the most important GUI elements to include both interesting values (range of valid values, out-of-range values) and a set of validation rules (expected results and validation functions) in order to guide the *test-case generation process*.

Once the new set of test cases is generated, and the validation rules are incorporated, the process ends with the *test-case execution process* (that includes the *validation process*). The result of the execution is a report including any relevant information to the tester. This test case set can be re-executed in order to perform a regression testing process.

4 Annotation Process

The *annotation process* is the process by which the tester indicates what GUI elements are important in terms of: first, which values can a GUI element hold, and thus should be tested; and second, what constraints should be met by a GUI element at a given time, and thus should be validated. The result of this process is a set of annotated GUI elements which will be helpful during the *test-case generation process* in order to identify the elements that represent a variation point, and the constraints that have to be met for a particular element or set of elements. From now on, this set will be called *Annotation Test Case*.

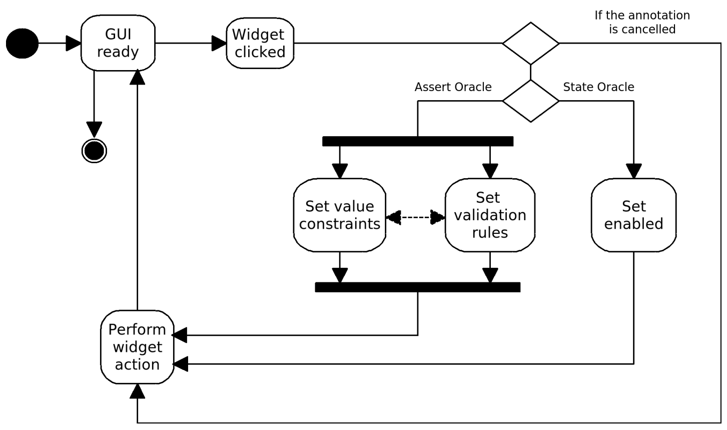

Figure 1: Widget Annotation Process.

This process could be implemented, for example, using a capture and replay (C&R) tool.[1] These tools provide the developers with access to the widgets information, so they could use this information along with the new values and the validation rules (provided by the tester in the *annotation process*) to build the *Annotation Test Case*.

As we can see in Figure 1, the *annotation process*, which starts with the tested application launched and its GUI ready for use, can be performed as follows:

1. For each widget the tester interacts with (e.g. clicks a widget or enters some data), he or she can choose between two options: annotate the widget (go to the next step) or continue as usual (go to step 3).

2. A widget can be annotated in two ways, depending on the chosen Test Oracle method: it might be an "Assert Oracle" (checks a set of validation rules related to the widget state), or a "State Oracle" (checks if the state of the widget during the *execution process* matches the state stored during the *annotation process*).

3. The annotations (if the tester has decided to annotate the widget) are recorded by the C&R tool as part of the *Annotation Test Case*. The GUI performs the actions triggered by the user interaction as usual.

4. The GUI is now ready to continue. The tester can continue clicking widgets or just finish the process.

[1]A *Capture and Replay Tool* captures events from the tested application and use them to generate test cases that replay the actions performed by the user. Authors of this paper have worked on the design and implementation of such tool as part of a previous research work, accessible on-line at `http://sourceforge.net/projects/openhmitester/` and at `http://www.um.es/catedraSAES/`.

The annotated widgets should be chosen carefully as too many annotated widgets in a test case may result in an explosion of test cases. Choosing an accurate value set also helps to get a reasonable test suite size.

The selection of the type of the test oracle depends on the needs of the tester:

- **Assert Oracles** are useful in two ways: first, if a new set of values is defined, new test cases will be generated to test these values; and second, if a set of validation rules is defined, these rules will be validated during the *execution and validation process*.
- **State Oracles** are useful when the tester has to check if a certain widget property or value remains constant during the *execution and validation process* (e.g. a widget that can not be disabled).

5 Test-Case Auto Generation Process

The *test-case auto generation process* is the process that automatically generates a new set of test cases from an initial set (those corresponding to the use cases that represent the behavior of the GUI) and an *Annotation Test Case*. As can be seen in Figure 2, the process generates a new *Annotated Test Suite* from an initial test suite and an *Annotation Test Case* (both together make up the initial *Annotated Test Suite*).

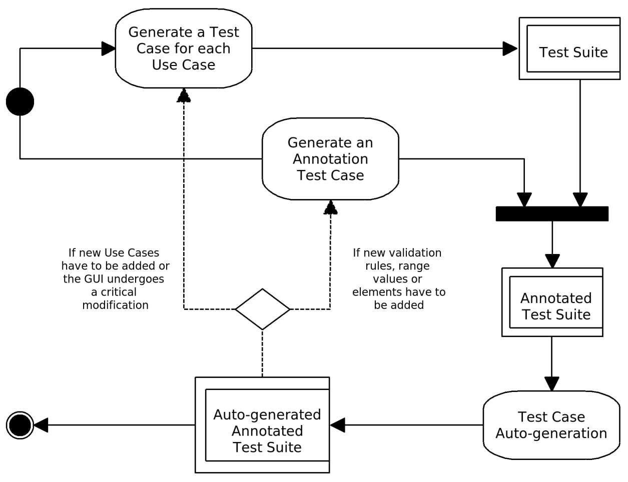
Generate a Test Case for each Use Case
Test Suite
Generate an Annotation Test Case
If new Use Cases have to be added or the GUI undergoes a critical modification
If new validation rules, range values or elements have to be added
Annotated Test Suite
Auto-generated Annotated Test Suite
Test Case Auto-generation

Figure 2: Test-Case Auto Generation Process.

The set of auto-generated test cases can be updated, for example, if the tester has to add or remove new use cases due to a critical modification in the GUI, or if new values or validation rules have to be added or removed. The tester will then update the initial test case set, the *Annotation Test Case*, or both, and will re-run the generation process.

The process will take as its starting point the initial set of test cases, from which it will generate new test cases taking into account the variation points (the new values) and the validation rules included in the annotations. The algorithm is shown in Figure 3.

```
initial_test_set ← ... // initial test case set
auto_gen_test_set ← {} // auto-generated test case set (empty)
annotated_elements ← ... // user-provided annotations

for all TestCase $tc \in$ initial_test_set do
  new_test_cases ← add_test_case (new_test_cases, tc)
  for all TestItem $ti \in tc$ do
    if $ti$.widget $\in$ annotated_elements then
      annotations ← annotations_for_widget($ti$.widget)
      new_test_cases ← create_new_test_cases (new_test_cases, annotations.values)
      new_test_cases ← add_validation_rules (new_test_cases, annotations.rules)
    end if
  end for
  auto_gen_test_set ← auto_gen_test_set $\cup$ new_test_cases
end for
return auto_gen_test_set
```

Figure 3: Test Case Auto-generation Algorithm.

Figure 4 is a graphical representation of the algorithm. The figure shows a test case which includes two annotated test items (an *Annotated Test Item* is a test item that includes a reference to an annotated widget). The first annotation specifies only two different values (15 and 25) and the second one specifies two new values (1 and 2) and two validation rules (one related to the *colour* property and another related to the *text* property). The result of the *test-case auto generation process* will be four new test cases, one for each possible path (15-1, 15-2, 25-1, and 25-2), and a validation point in the second annotated test item.

6 Execution and Validation Process

The *execution and validation process* is the process by which the test cases (auto-generated in the last step) are executed and the validation rules are asserted to check whether the constraints are met.

The *test-case execution process* executes all the test cases in order. For each test case, the GUI must be reset to its initial state. This constraint is very important to ensure that every use case is launched under the same conditions. This feature allows the tester to implement different test configurations, ranging from a set of a few test cases to an extensive battery of tests for a nightly or regression testing process [MBHN03]. As for the *validation process*, in this paper we describe a Test Oracle[2] based validation process, which uses test

[2]A *Test Oracle* is a mechanism that generates the expected output that a product should have for determining,

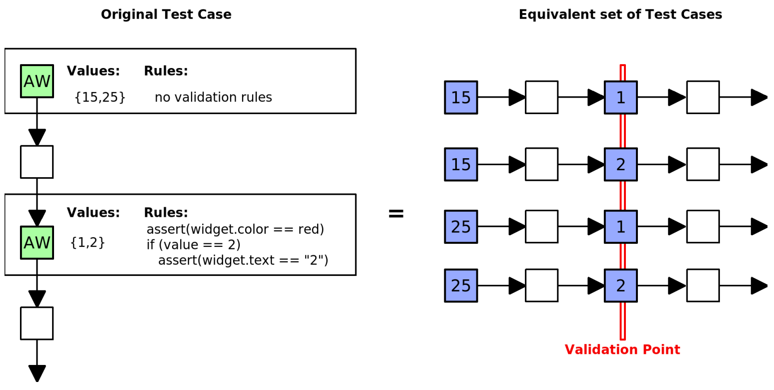

Figure 4: Test Case branching.

oracles [XM07, XM06] to perform widget-level validations. The features of the *validation process* vary depending on the oracle method selected during the *annotation process*.

For *Assert Oracles* (which check a set of validation rules related to the widget) the tester needs to somehow define a set of validation rules. Defining these rules is not straightforward. Expressive and flexible (e.g. constraint or script) languages are needed to allow the tester to define assert rules for the properties of the annotated widget, and, possibly, to other widgets. Another important pitfall is that if the GUI encounters an error, it may reach an unexpected or inconsistent state. Further executing the test case useless; therefore it is necessary some mechanism to detect these "bad states" and stop the test case execution.

For *State Oracles* (which check if the state of the widget during the *execution process* matches the state stored during the *annotation process*) the system needs to know how to extract the state from the widgets, represent it somehow, and be able to check it for validity. In our approach, it could be implemented using widget adapters.

The *validation process* may be additionally completed with *Crash Oracles*, which perform an application-level validation (as opposed to widget-level) as they can detect crashes during test case execution. These oracles are used to signal and identify serious problems in the software.

Finally, it is important to remember that there are two important limitations when using test oracles in GUI testing [XM07]: first, GUI events have to be deterministic, to be able to predict their outcome; second, since the software back-end is not modeled (e.g. data in a data base), the GUI may return a non-expected state which would be detected as an error.

after a comparison process, whether the product has passed or failed a test.

7 Conclusions and Future work

Automated GUI test case generation is an extremely resource intensive process as it is usually guided by a complex and fairly difficult to build GUI model. In this context, this paper presents a new approach for automatically generating GUI test cases based on both GUI use cases (required functionality), and annotations of possible and interesting variations of graphical elements (which generate families of test cases), as well as validation rules for their possible values. This reduces the effort required in test coverage and GUI modeling. Thus, this method would help reducing the time needed to develop a software product since the testing and validation processes spend less efforts.

As a statement of direction, we are currently working on an architecture and the details of an open-source implementation which allow us to implement these ideas and future challenges as, for example, extend the GUI testing process towards the application logic.

Acknowledgements

This paper has been partially funded by the "Cátedra SAES of the University of Murcia" initiative, a joint effort between SAES (Sociedad Anónima de Electrónica Submarina, `http://www.electronica-submarina.com`) and the University of Murcia to work on open-source software, and real-time and critical information systems.

References

[MBHN03] Atif Memon, Ishan Banerjee, Nada Hashmi, and Adithya Nagarajan. DART: A Framework for Regression Testing "Nightly/daily Builds" of GUI Applications. *IEEE Internacional Conference on Software Maintenance (ICSM'03)*, 2003.

[MBN03] Atif Memon, Ishan Banerjee, and Adithya Nagarajan. GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing. *IEEE 10th Working Conference on Reverse Engineering (WCRE'03)*, 2003.

[MX05] Atif Memon and Quing Xie. Studing the Fault-Detection Effectiveness of GUI Test Cases for Rapidly Envolving Software. *IEEE Computer Society*, 2005.

[PFV07] Ana C. R. Paiva, Joao C. P. Faria, and Raul F. A. M. Vidal. Towards the Integration of Visual and Formal Models for GUI Testing. *elsevier: Electronic Notes in Theoretical Computer Science 190*, pages 99–111, 2007.

[VLH$^+$06] Marlon Vieira, Johanne Leduc, Bill Hasling, Rajesh Subramanyan, and Juergen Kazmeier. Automation of GUI Testing Using a Model-driven Approach. *Siemens Corporate Research*, 2006.

[XM06] Qing Xie and Atif M. Memon. Model-Based Testing of Community-Driven Open-Source GUI Applications. *IEEE International Conference on Software Maintenance (ICSM'06)*, 2006.

[XM07] Q. Xie and A. M. Memon. Designing and comparing automated test oracles for GUI-based software applications. *ACM Trans. Softw. Eng. Methodol. 16, 1, 4*, 2007.

[YM07] Xun Yuan and Atif M Memon. Using GUI Run-Time State as Feedback to Generate Test Cases. *29th International Conference on Software Engineering (ICSE'07)*, 2007.

[ZKP$^+$08] R. S. Zybin, V. V. Kuliamin, A. V. Ponomarenko, V. V. Rubanov, and E. S. Chernov. Automation of Broad Sanity Test Generation. *ISSN 0361-7688, Programming and Computer Software, Vol. 34*, 2008.