An Automated Testing Framework for Statistical Testing of GUI Applications

Lan Lin, Jia He, Yufeng Xue
Ball State University
Department of Computer Science
Muncie, IN 47396, USA
{llin4, jhe, yxue2}@bsu.edu

Abstract

It is known to be inherently more difficult and labor-intensive to functionally test software applications that employ a graphical user interface front-end, due to the vast GUI input space. We propose an automated testing framework for functional and statistical testing of GUI-driven applications, using a combination of two rigorous software specification and testing methods and integrating them with an automated testing tool suitable for testing GUI applications. With this framework we are able to achieve fully automated statistical testing and software certification. We report an elaborate case study that demonstrates a pathway towards lowered cost of testing and improved product quality for this type of applications.

1 Introduction

Software applications that employ a graphical user interface (GUI) front-end are ubiquitous nowadays, yet they present additional challenges to software testing. It is inherently more difficult and labor-intensive to functionally test a GUI-driven application than a traditional application with a command line interface, due to the vast GUI input space and the prohibitively large number of possible sequences of user input events (each event sequence being a potential test case) [13, 10, 11, 20]. Testing therefore needs to be automated in order to run a large sample of test cases to verify correct functionality.

In this paper we propose an automated testing framework for functional and statistical testing of GUI-driven applications, using two rigorous software specification and testing methods in combination, namely sequence-based software specification [12, 18, 19, 17] and Markov chain usage-based statistical testing [14, 16, 22, 21], and integrating them with an automated testing tool suitable for testing GUI applications, that provides fully automated statistical testing and software certification as a means to achieve high product quality. Both methods and the supporting tools [1, 2, 3] were developed by the University of Tennessee Software Quality Research Laboratory (UTK SQRL). Although work has been done in the past to combine these methods together [8, 7, 9], it remains application and problem specific to work out a seamless integration from original requirements to fully automated statistical testing and software certification. We present in this paper our efforts and experiences along this path in solving a real world problem.

Sequence-based specification is a method for systematically deriving a system model from informal requirements through a sequence enumeration process [12, 18, 19, 17]. Under this process stimulus (input) sequences are considered in a breadth-first manner (length-lexicographically), with the expected system response to each input sequence given. Not all sequences of stimuli are considered since a sequence need not be extended if either it is illegal (it cannot be applied in practice) or it can be reduced to another sequence previously considered (the sequences take the system to the same state). Sequence enumeration leads to a model that can be used as the basis for both implementation and testing [8, 7, 9].

Markov chain usage-based statistical testing [14, 16, 22, 21] is statistical testing based on a Markov chain usage model. It is a comprehensive application of statistical science to the testing of software, with the population of all uses of the software (all use cases) modeled as a Markov chain. States of the Markov chain usage model represent states of system use. Arcs between states represent possible transitions between states of use. Each arc has an associated probability of making that particular transition based on a usage profile. The outgoing arcs from each state have probabilities that sum to one. The directed graph structure, together with the probability distributions over the exit arcs of each state, represents the expected use of the software in its intended operational environment. There are both informal and formal methods of building the usage model structure (sequence-based specification can be used as a formal method). The transition probabilities among states come
from historical or projected usage data for the application.

The paper is structured as follows. The next section introduces our case study. Sections 3 and 4 illustrate how we constructed a usage model for statistical testing and present the model analysis results. Section 5 presents an automated testing framework we have developed for fully automated statistical testing of GUI applications, using the case study as a running example. Section 6 discusses our test and certification plan. Section 7 illustrates how testing was performed and presents results following a test case analysis. Section 8 concludes the paper.

2 The case study: The BlackBoard Quiz Editor (BBQE)

Our chosen case study is a Java GUI application, the BlackBoard Quiz Editor (BBQE, see Figure 1), that can author quizzes and save them in a format that can be imported in BlackBoard [4] - a learning management system used by Ball State University and many other institutions across the country for course delivery and management. The application was delivered in 2013 as a completed Ball State Computer Science major capstone project, considered of good quality by the client, and is being used by many faculty at Ball State. We were interested in automated statistical testing of this application to find bugs and to get a quantitative measure of its projected reliability.

The BBQE interface contains three main areas: a main toolbar, a quiz panel, and a question editor panel. It supports eleven question types, including the essay question type, the fill in the blank question type, the matching question type, the true or false question type, the short answer question type, etc. Quizzes created in BBQE can be saved as a text file and easily imported into BBQE and BlackBoard.

To limit the testing problem to a manageable size, we defined our testing scope such that the System Under Test (SUT) contains only one question type: the essay question type, but includes all GUI features. Other models can be constructed similarly addressing other question types.

3 Usage modeling

In order to develop a usage model for statistical testing, we adopted a formal approach and first developed a rigorous specification (that encodes a formal system model) based on the requirements. As detailed requirements for the BBQE could not be found, we re-engineered requirements based on the user manual and the delivered application. We applied sequence-based specification [12, 18, 19, 17] and the supporting tool, the REAL [2], to developing a rigorous specification of the SUT.

The resulting specification contains 14 stimuli (inputs), 14 responses (outputs), 48 distinct states, 673 enumerated stimulus sequences, 9 original requirements, and 13 derived requirements. Figure 2 shows all the original (the top 9) and derived requirements for the SUT. Table 1 and Table 2 list all the stimuli and responses, respectively, across the system boundary. Each stimulus (input) and response (output) is given a short name (see the first columns) to facilitate sequence enumeration, tied to an interface in the system boundary, and traced to the requirements. Excerpts of an enumeration for the SUT are shown in Table 3. Stimulus sequences are enumerated in length-lexicographical order (based on the alphabetical order of stimuli as shown in Table 1) following the enumeration rules. Stimulus sequences are represented by concatenating stimuli to string prefixes with periods in the Sequence and the Equivalence columns. Each row shows for an enumerated stimulus sequence what should be the software’s response, if the sequence could be reduced to a prior sequence (based on whether they take the system to the same state), and traces to the requirements that justify these decisions.

From the completed sequence enumeration we obtained a state machine for the SUT. We further added a source state together with an arc from the source to the state represented by the empty sequence (λ), and a sink state together with an

<table>
<thead>
<tr>
<th>Stimulus</th>
<th>Long Name</th>
<th>Description</th>
<th>Interface</th>
<th>Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Add button</td>
<td>Click the create question button</td>
<td>Main window</td>
<td>req1</td>
</tr>
<tr>
<td>C</td>
<td>Create button</td>
<td>Click the create question button</td>
<td>Main window</td>
<td>req1</td>
</tr>
<tr>
<td>CO</td>
<td>Copy question</td>
<td>Copy the question</td>
<td>Main window</td>
<td>req1</td>
</tr>
<tr>
<td>EQ</td>
<td>Essay question</td>
<td>Select the essay question type</td>
<td>Question creation window</td>
<td>req2</td>
</tr>
<tr>
<td>F</td>
<td>Favorite button</td>
<td>Click the favorite question button</td>
<td>Main window</td>
<td>req1</td>
</tr>
<tr>
<td>FS</td>
<td>Favorite checkbox</td>
<td>Click (Check/Uncheck) the favorite checkbox</td>
<td>Question creation window</td>
<td>req2</td>
</tr>
<tr>
<td>H</td>
<td>Help button</td>
<td>Click the help button</td>
<td>Main window</td>
<td>req1</td>
</tr>
<tr>
<td>HC</td>
<td>Help-cancel button</td>
<td>Click the help cancel button</td>
<td>Help window</td>
<td>req1</td>
</tr>
<tr>
<td>P</td>
<td>Paste button</td>
<td>Paste the question</td>
<td>Mouse</td>
<td>req4</td>
</tr>
<tr>
<td>QF</td>
<td>Question fill</td>
<td>Fill in the question edit box</td>
<td>Question editor panel</td>
<td>req6</td>
</tr>
<tr>
<td>U</td>
<td>Up button</td>
<td>Click the question up button</td>
<td>Main window</td>
<td>req1</td>
</tr>
</tbody>
</table>

Figure 1. The BlackBoard Quiz Editor
For the lack of compelling information to the contrary regarding the usage profile, we took the mathematically neutral position and assigned uniform probabilities to transitions in the usage model. The constructed usage model is diagrammed in Figure 3 using a graph editor (with 50 nodes and 619 arcs the visualization becomes very cluttered). Although not readable unless one zooms in, it illustrates the size of our testing problem.

4 Model analysis

We performed a model analysis using the JUMBL [3]. Table 4 shows the model statistics, including the number of nodes, arcs, and stimuli in the usage model, the expected test case length (the mean value, i.e., the average number of steps in a randomly generated test case) and variance.

The following statistics are computed for every node, every arc, and every stimulus of the usage model:

- **Occupancy.** The amount of time in the long run that one will spend testing a node/arc/stimulus.

- **Probability of Occurrence.** The probability of a node/arc/stimulus appearing in a random test case.

- **Mean Occurrence.** The average number of times a node/arc/stimulus will appear in a random test case.

- **Mean First Passage.** The number of random test cases one will need to run on average before testing a node/arc/stimulus for the first time.

Note: The values in the table are rounded to two decimal places for simplicity.
Figure 3. A state machine for the BBQE under test (with the source and the sink marked in green and red respectively)

Table 4. Model statistics

| Node Count | 50 nodes |
| Arc Count  | 619 arcs |
| Stimulus Count | 18 stimuli |
| Expected Test Case Length | 11.573 events |
| Test Case Length Variance | 47.776 events |
| Transition Matrix Density (Nonzeros) | 0.1056 (264 nonzeros) |
| Undirected Graph Cyclomatic Number | 215 |

These statistics are validated against what is known or believed about the application domain and the environment of use.

5 An automated testing framework

Following the model analysis we developed an automated testing framework for fully automated, statistical testing of BBQE. This required (1) finding an automated testing tool suitable for our chosen application, and (2) integrating it with our statistical testing tool, JUMBL, for automated test case generation, automated test case execution, and automated test case evaluation.

After some research we chose HP’s Quick Test Professional (QTP) [5] as an automated testing tool for BBQE because it is HP’s successor to its WinRunner and X-Runner software supporting functional and automated GUI testing and it also works for 32-bit machines. We created an object repository in QTP that registers all the static GUI objects of the SUT, and used QTP’s Test Scripting Language (TSL) (a subset of VBScript) to write test cases that can run automatically in QTP.

Our constructed usage model was written in The Modeling Language (TML) [6] (Figure 4 shows a tiny piece). State names (representing usage states) are enclosed in square brackets and arc names (representing usage events/expected software’s responses) are enclosed in quotation marks. For each state the list of event/expected response - next state pairs is given following the state name. For instance in Figure 4 there is an arc from state [A] triggered by usage event “A” with expected response “null” going back to state [A].

Using labels the TML model can include test automation information, which can be extracted when JUMBL automatically generates test cases of all types from the usage model using the GenTest command. The challenge was how we should annotate the states and arcs of the usage model with test scripts that could be understood by QTP such that when these test scripts are extracted and concatenated into a generated test case they literally become a program written in TSL that QTP could automatically execute. To achieve this we accomplished the following steps:

- **Stimulus generation with TSL.** We wrote TSL scripts that issue each possible stimulus (input) to the SUT. For instance, to issue the stimulus “A” (for clicking the add questions button in the main window), the following function is called:

  ```tcl
  Function stim_A()
  JavaWindow("Blackboard Quiz Editor") _
  .JavaButton("add").Click
  End Function
  ```

- **Response checking with TSL.** We wrote TSL scripts
that check each possible response (output) (including the null response) is observed as expected. For instance, the following function is called to verify the response “EQC” (for having created an essay question in the question panel):

```pascal
Function check_EQC()
    qnum = qnum + 1
    Set props = JavaWindow("Blackboard Quiz Editor") .JavaStaticText("ESS(st)") .GetTOProperties
    Set ChildObjects = JavaWindow("Blackboard Quiz Editor") .JavaObject("JPanel_2") .ChildObjects(props)
    ess = ChildObjects.Count
    If qnum <> ess Then
        returnValue = False
    Else
        returnValue = True
    End If
    qnum = ess
    check_EQC = returnValue
End Function
```

- **State verification with TSL.** We wrote TSL scripts that verify if the SUT is in any specific state as described by the usage model, probing values from the system’s state variables. For instance, the following function is called to identify if the system is in the state represented by the stimulus sequence $\lambda$ (each such sequence is a canonical sequence in the sequence enumeration):

```pascal
Function verify_lambda()
    If (check_var_EQ() = False) And (check_var_EQS() = False) And (check_var_HW() = 0) And (check_var_QW() = 0) Then
        returnValue = True
    Else
        returnValue = False
    End If
    verify_lambda = returnValue
End Function
```

- **Usage model annotation.** We wrote Python code that automates the usage model annotation with TSL. Each state is annotated with a call to a state verification function. Each arc is annotated with a call to issue the stimulus to the SUT followed by one or more calls to check the observed responses. The rigorous specification serves as the test oracle. Each state/arc after annotation is associated with testing commands that are understood by QTP. When test cases are automatically generated and exported using the JUMBL, each test case literally becomes a TSL script that can automatically execute in QTP.

- **Result recording with TSL.** We wrote Python code that embeds TSL scripts in the annotated usage model recording test results. A test case is considered successful only if all its constituting steps are successful. For any failed test case the test case number and all failure step numbers are written to a text file, together with information indicating whether each failure step is a continue failure (the rest of the steps can still run to completion after this failure step) or a stop failure (the following test steps included in this test case cannot be executed).

- **Automated test case execution and evaluation.** We wrote a shell script that runs from command line, automatically executes a large sample of generated test cases with our developed JUMBL-QTP interfaces, and records test results in a text file.

- **Reading failure data and recording it back in JUMBL.** We wrote a script that runs from command line, reads the failure data after testing is completed and records it back into the JUMBL for statistical analysis.

Figure 5 shows an excerpt of the usage model after annotation. Test automation information is included with a label (the text following a |$ up to and including the end of line) in TML and an associated key (“a” in Figure 5 followed with a colon (:)) before the label). Test automation scripts can be attached to a model (e.g., the lines following declaring “model bbq.mml” up to the next empty line; here we do any needed test initialization and declare all stimulus generation/response checking/state verification functions), a state (e.g., the lines following declaring state “[A]” up to the next empty line; we verify if the SUT is in this state and if not record the last step/event as a stop failure and exit the test), or an arc (e.g., the lines following arc “A/null” up to declaring the to-state “[A]”; we issue the stimulus and check the response and if the observed and expected responses differ record the current step as a failure step).

Figures 6 and 7 show excerpts of an automatically generated test case from the usage model. Before model annotation the exported test case is a sequence of events/steps traversing the usage model starting from the source and ending with the sink (see Figure 6). After annotation all the test scripts associated with the states and arcs of the particular path are extracted and concatenated into a TSL script that is understood by QTP and automatically executable (see Figure 7).

6 A test and certification plan

We developed the following test and certification plan for the SUT:

- Run 48 minimum coverage test cases that cover every arc and every state of the usage model.
Figure 5. An Excerpt of the usage model for the BBQE under test after annotation

```
 model bbqml
  a: If JavaWindow("Blackboard Quiz Editor").Exist(2)=true Then
    $SystemUtil.Run "D:\bbq.exe", "open"
  $Function stim_A()
    $JavaWindow("Blackboard Quiz Editor").addAction("add").click
  $End Function

  [A]
  a: If (verify_A() = False) Then
    $currStamp = CStr(testCaseNo) & "," & CStr(stepNo)
    $if (StrComp(currStamp, stamp) <> 0) Then
      $errorFile.writeFile(stepNo)
      $errorFile.writeFile(""")
    $End If
    $errorFile.writeFile(""")
    $ExitTest
  $End If

  "A(null)"
  a: $stepNo = stepNo + 1
  $record(0)
  $stim_A()
  $record(2)
  If (check_null() = False) Then
    $stamp = CStr(testCaseNo) & "," & "$"
    $errorFile.writeFile($"
    $errorFile.writeFile(""")
  $End If

  [A]

  "Extra arc to sink state"
  a: $stepNo = stepNo + 1
  $JavaWindow("Blackboard Quiz Editor").Close()
  $Exit

end // of model bbqml
```

Figure 6. An example test case that is automatically generated from the usage model before model annotation

```
 # ==-----------------------------------------------
 # Trajectory: 0
 # Model: bbqml
 # Key: random
 # Events: 13
 # Including failure information.
 # ==-----------------------------------------------
 # Step: 1, Trajectory: 0
 # (int): From init to lambda state for initialization purposes
 # Step: 2, Trajectory: 0
 # [Lambda]: "p/null"
 # Step: 3, Trajectory: 0
 # [Lambda]: "u/null"
 # Step: 4, Trajectory: 0
 # [Lambda]: "u/null"
 # Step: 5, Trajectory: 0
 # [Lambda]: "u/null"
 # Step: 6, Trajectory: 0
 # [Lambda]: "A/qe"
 # Step: 7, Trajectory: 0
 # [Lambda]: "A/qe"
 # Step: 8, Trajectory: 0
 # [Lambda]: "D/null"
 # Step: 9, Trajectory: 0
 # [Lambda]: "D/null"
 # Step: 10, Trajectory: 0
 # [Lambda]: "D/null"
 # Step: 11, Trajectory: 0
 # [Lambda]: "H/mr"
 # Step: 12, Trajectory: 0
 # [Lambda]: "H/mr"
 # Step: 13, Trajectory: 0
 # [Lambda]: "Extra arc to sink state"
```

Figure 7. An excerpt of an example test case that is automatically generated from the usage model after model annotation

```
 # ==-----------------------------------------------
 # Trajectory: 0
 # Model: bbqml
 # Key: random
 # Events: 13
 # Including failure information.
 # ==-----------------------------------------------
 # Step: 2, Trajectory: 0
 # (int): From init to lambda state for initialization purposes
 # Step: 3, Trajectory: 0
 # [Lambda]: "p/null"
 # Step: 4, Trajectory: 0
 # [Lambda]: "u/null"
 # Step: 5, Trajectory: 0
 # [Lambda]: "u/null"
 # Step: 6, Trajectory: 0
 # [Lambda]: "A/qe"
 # Step: 7, Trajectory: 0
 # [Lambda]: "A/qe"
 # Step: 8, Trajectory: 0
 # [Lambda]: "D/null"
 # Step: 9, Trajectory: 0
 # [Lambda]: "D/null"
 # Step: 10, Trajectory: 0
 # [Lambda]: "D/null"
 # Step: 11, Trajectory: 0
 # [Lambda]: "H/mr"
 # Step: 12, Trajectory: 0
 # [Lambda]: "H/mr"
 # Step: 13, Trajectory: 0
 # [Lambda]: "Extra arc to sink state"
```
- Run 200 weighted test cases that represent the 200 most probable paths of the usage model.

- Run 2,000 random test cases that are generated from the usage model based on the arc probabilities.

- Total testing consists of 25,577 transitions for the above 2,248 test cases.

- If all tests run successfully, this will demonstrate empirical evidence to support a claim of reliability > 0.90 given the defined protocol (of our usage model for the SUT, our selection of test cases, the actual result of testing, and the reliability model implemented in the JUMBL).

7 Automated statistical testing and test case analysis

Using the automated testing framework we had developed and the JUMBL, we were able to automatically generate, automatically execute, and automatically evaluate the sample of 2,248 test cases. Our testing was done on a laptop with Intel Core™ i-7-3630QM CPU with 4 cores, 2.40 GHz clock speed, and 8 GB memory. It took 2 days, 15 hours, 8 minutes and 4 seconds to run the 2,248 test cases, of which 710 were successful and 1,538 were failed.

We did a test case analysis using the JUMBL based on our testing experience. Excerpts of the test case analysis are shown in Table 5. Some important statistics include:

- Nodes/Arcs/Stimuli Generated. The number of states/arcs/stimuli covered in the generated test cases.

- Nodes/Arcs/Stimuli Executed. The number of states/arcs/stimuli covered in the executed test cases.

- Arc/Stimulus Reliability. The estimated probability of executing an arc / a stimulus in a test case successfully.

- Single Event Reliability. The estimated probability that a randomly selected arc can be executed successfully in a test case.

- Single Use Reliability. The estimated probability of executing a randomly selected test case successfully.

- Optimum Reliability. The estimated reliability if all generated test cases were executed successfully.

- Relative Kullback Discriminant. A measure of how close the performed testing matches the software use as described by the usage model.

Of which the most important statistic, the single use reliability, estimates “the probability of the software executing a randomly selected use without a failure relative to a specification of correct behavior.” [15] The low single use reliability observed in this example (0.270545355) was due to the high number of failed test cases (1,538 out of 2,248).

Tracing through some failed test cases we identified 12 discrepancies between the specification and the code (see Figure 8). This record of specification-implementation discrepancies will be helpful in locating and fixing bugs in the released code.

Figure 8. BBQE specification-code discrepancies

<table>
<thead>
<tr>
<th>Table 5. Excerpts of the test case analysis: Reliabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Event Reliability</td>
</tr>
<tr>
<td>Single Event Variance</td>
</tr>
<tr>
<td>Single Event Optimum Reliability</td>
</tr>
<tr>
<td>Single Event Optimum Variance</td>
</tr>
<tr>
<td>Single Use Reliability</td>
</tr>
<tr>
<td>Single Use Variance</td>
</tr>
<tr>
<td>Single Use Optimum Reliability</td>
</tr>
<tr>
<td>Single Use Optimum Variance</td>
</tr>
<tr>
<td>Arc Source Entropy</td>
</tr>
<tr>
<td>Kullback Discrimination</td>
</tr>
<tr>
<td>Relative Kullback Discrimination</td>
</tr>
<tr>
<td>Optimum Kullback Discrimination</td>
</tr>
<tr>
<td>Optimum Relative Kullback Discrimination</td>
</tr>
</tbody>
</table>
8 Conclusion

In this paper we demonstrated an automated testing framework for fully automated statistical testing of GUI applications. We applied two rigorous software specification and testing methods and the supporting tools, and integrated them with an automated testing tool suitable for GUI applications, and reported an elaborate case study. As the readers might find out, working on any non-trivial real world problem requires considerable efforts be made to work out all the details needed for fully automated testing with no human intervention, however, by the end of the process we have the ability of running large numbers of tests, as well as an automated testing facility for low-cost, quick-turnaround testing and re-testing. All the artifacts we have produced in this process, including the usage model, test oracle, JUMBL-QTP interfaces, testing records, test plans, test scripts, test cases, product measures and evaluation criteria, all become reusable testing assets. Our experiences demonstrated a pathway towards lowered cost of testing and improved product quality for this type of applications.

Acknowledgements

This work was generously funded by Lockheed Martin Corporation and Northrop Grumman Corporation through the NSF Security and Software Engineering Research Center (S²ERC).

References