THE GUI TESTING METHOD BASED ON TESTING META-MODEL

Andrej Usaniov, Sarunas Packevicius, Kestutis Motiejunas

Kaunas University of Technology, Department of Software Engineering, Studentu 50, Kaunas, Lithuania, andrej.usaniov@soften.ktu.lt, packsaru@soften.ktu.lt, kestas@soften.ktu.lt

Abstract. Software testing consumes more than 50 % of all software development resources. Today the majority of software has a graphical user interface. The most popular way to test software functions is to test them through a user interface. Testing can be easily dropped out and increasing likelihood of producing a faulty software by many times [1]. To reduce testing costs a testing automation is employed. The GUI testing method based on testing meta-model, experiments execution and results evaluation are presented in this article.

Keywords: GUI testing method, tests meta-model.

1 Introduction

Today the majority of software has graphic interface. Users could access software functions through graphical user interface. Of course the most popular way to test software is to test it through its user interface [2]. This is usually references as GUI (Graphical User interface) testing. During GUI testing software is verified through its interface. A tester enters some input data into software windows and checks if produced result is correct. This process is manual and very labour intensive.

In practice testing process is often associated with hard time and budget constraints, lightly documented requirements, misunderstandings of testing objectives, and inaccurate evaluation of testing scope.

The user interface tests are usually documented in some form as steps in some semi formal text documents. Later texts documents are read by testers and described test cases are executed manually. The test outcome is evaluated manually by tester as well.

In order to reduce testing costs tests automation is employed [3-7]. The way to reduce testing costs and to allow performing more extensive software testing is to automate software testing process [8]. The automation includes automatic test case preparation, tests execution and result verification. This allows testing software more extensively, thus finding more bugs and increasing its quality while reducing testing costs.

Ryser and Glinzhas proposed a GUI testing method by formally describing software usage scenarios and providing instructions on how to manually perform testing using defined scenarios [9]. The drawback of this approach is that all testing has to be performed manually, the method only provides best practices how to develop testing scenarios and perform tests execution.

Jesus and Luis presented method of developing graphical user interfaces from two UML models: use case and activity diagrams. They defined some rules of transformation of specifications into the user interface. Firstly user interface components are represented into an UML class diagram. Then class diagram is used for generating code fragments which can be considered as GUI prototypes [10].

Memon proposed GUI testing method is based on system’s GUI behaviour graph representation [11], where nodes are the states of GUI and transitions are events. To specify graph during manual GUI examination the basic steps of the method are: Identifying Components and Events, Creating Event-flow Graphs, Computing Event-sequences. Thus this approach is prone to graph incompleteness. The drawback of this method is that it is a labour-intensive.

Currently one of the most popular GUI testing techniques is a record-playback automation method [9]. During first run testing is executed manually. Another drawback of record-playback tools is high dependency on a platform. For instance often tools support multiple platforms, but its scripts recorded on the one platform are invalid for execution on another one. Also a script recorded with a one tool is incompatible for another tool [2].

Some authors are proposing an easier ways to create automated user interface tests then using record playback tools [2, 12] or automatically generating user interface tests [13, 14].

The majority of existing GUI testing methods orientates on GUI graphical representation preparation methodology. Often only providing instructions how to prepare tests manually, but doesn’t provide means on how to get test cases. Also test cases should be executed and results evaluated manually. Some methods define how to automate only a part of test process. Other testing methods describe how to execute manually created tests automatically.

To overcome the manual tests creation problem the tests generation approach based on tests meta-model creation is proposed in this paper.

- 233 -
2 The GUI testing method

The GUI testing method is based on tests meta-model. The structure of proposed method consists of those parts:

1. Tests meta-model creation;
2. Generation/selection of testing scenarios sets using tester’s selected graph traversing algorithm: all paths, main paths, all nodes;
3. Generation of executable testing scripts from testing scenarios;
4. Execution of testing scripts on SUT and code coverage measurement;
5. Selection of testing scripts from generated testing scripts sets based on code coverage measured during execution;

The structure of the GUI testing method is presented below (Figure 1). Tests meta-model is the graphical representation of testing process from tester’s perspective. The UML 2.0 activity diagram is used. Each activity diagram is the meta-model of some SUT function the tester intents to test. The way how the tester’s actions are depicted corresponds to tester’s understanding of system behaviour. The activities in the diagram correspond to the steps of the test. Also activities are mapped to certain graphical components of SUT and describe events on them. Data pins are used for input data passing and output data receiving. Finally all activity diagrams are used to create master diagram by using calls to functions diagrams – the tests meta-model of SUT. Test meta-model creation is the only manually performed step of proposed GUI testing method.

![Figure 1. The structure of the GUI testing method](image)

Depending on testing goals and project constraints different graph traversing algorithms might be used for generation of initial testing scenarios sets. The testing scenarios set generated using all paths traversal algorithms, hereafter “all paths set”, stands for all independent paths within system under test (SUT). Thus execution of all generated scenarios requires the biggest amount of time. The testing scenarios set generated using all nodes traversal algorithm, hereafter “all nodes set”, ensures that all defined tester’s action within testing scenarios will be reached at least once. This is a subset of all paths set with smaller number of testing scenarios and requires less execution time. The testing scenarios set generated using main path traversal algorithm, hereafter “main paths set”, consists only of minimal number of testing scenarios which covers only the main aspects of software functionality.

Generated testing scenarios sets are converted into executable testing scripts. They may be expressed in different programming languages. Prepared testing scripts are being automatically executed on SUT providing code coverage feedback for subsequent testing scenarios sets selection step.

3 Example

To demonstrate the GUI testing method the automated teller machine (ATM) is used. Below is given an example of ATM pin code entering functionality tests meta-model (Figure 2).
The diagram consists of eight (8) nodes and ten (10) edges with five (5) paths. The numbers of paths or testing scenarios for pin code entering functionality generated using different traversal algorithms from testing meta-model are presented in the table below, when cycles within diagram are limited to one (Table 1).

### Table 1 The numbers of testing scenarios for pin code entering functionality

<table>
<thead>
<tr>
<th></th>
<th>All paths</th>
<th>All nodes</th>
<th>Main paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of testing scenarios</td>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Tests meta-model enables generation of testing scenarios sets. Tests meta-model is directed graph which is being parsed using different traverse algorithms: all nodes, main paths and all paths. Example of generated testing scenario is given below (Listing 1).

#### Listing 1. Testing scenario for Pin code entering functionality

```latex
EnterPinCodeActivityTest1(
  InitialNode
  DecisionNode
  EnterPinCode
  DecisionNode
  Ok
  ActivityFinalNode
)
```

By supplementing testing scenarios with testing data executable testing scripts are being generated. Example of generated executable testing script is given below (Listing 2).

#### Listing 2. Executable testing script for Pin code entering functionality

```latex
function EnterPinCodeActivityTest1(pinCode, result)
  PinCode=pinCode;
  Sys.Process("ATM").ATM.EnterPin.enterPinTextBox.Text = PinCode;
  Sys.Process("ATM").ATM.EnterPin.enterPinOkButton.ClickButton();
  result[0]="OK";
}
```

The number of testing scripts depends on the size of given testing data applied for selected testing scenarios. Numbers of generated testing scenarios and testing scripts for ATM tests meta-model are given below (Table 2).

### Table 2. The numbers of testing scenarios for ATM

<table>
<thead>
<tr>
<th></th>
<th>All paths</th>
<th>All nodes</th>
<th>Main paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testing scenarios set size</td>
<td>650</td>
<td>176</td>
<td>19</td>
</tr>
<tr>
<td>Testing scripts set size</td>
<td>2375</td>
<td>638</td>
<td>61</td>
</tr>
<tr>
<td>Testing scripts execution duration</td>
<td>38'12&quot;</td>
<td>7'56&quot;</td>
<td>0'53&quot;</td>
</tr>
</tbody>
</table>
During execution of testing scripts the code coverage of the ATM is measured. This allows evaluating effectiveness of created tests meta-model. The code coverage values reached by testing scenarios sets generated using different traversal algorithms are presented below (Figure 3).

![Figure 3. The code coverage using different traverse algorithm](image)

None of generated scripts sets was able to reach 100% code coverage. The reason for that is inability to measure code coverage within classes’ destructors on program exit.

Main path set requires about one (1) minute of execution and ensures symbol code coverage at 91.49% level. Execution time increases almost forty (40) times then all path set is used. This allows increasing symbol code coverage up to 97.44%.

Using code coverage measurement generated testing scripts sets could be reduced by subsequently selecting only those scripts which have influence on the code coverage increase. Finally subset, hereafter selected set, having significantly reduced number of scripts provides the same code coverage as generated set (Table 3).

Table 3. Reduction of testing scripts set

<table>
<thead>
<tr>
<th>Testing scripts sets size</th>
<th>Times decreased</th>
<th>Selected set size, %</th>
</tr>
</thead>
<tbody>
<tr>
<td>generated</td>
<td>selected</td>
<td></td>
</tr>
<tr>
<td>Main path</td>
<td>61</td>
<td>30</td>
</tr>
<tr>
<td>All nodes</td>
<td>638</td>
<td>118</td>
</tr>
<tr>
<td>All paths</td>
<td>2375</td>
<td>329</td>
</tr>
</tbody>
</table>

The main path selected set having 30 scripts provides symbol code coverage of 91.49%. The all paths selected set having 329 scripts is eleven (11) times bigger and gives only 5.95% coverage increase.

The main paths’ set is best suited for projects with tight time constraints. With relatively low execution duration of about 1 minute it provides acceptable, higher than 90%, code coverage. The all paths’ set gives only a slight code coverage increase, up to 6%. The all nodes set reaches code coverage level close to all paths’ set results, but execution duration is fivefold (5) lower.

A generated and selected testing scenarios sets where evaluated using mutation testing. The ATM application was modified to create mutated versions so called mutants. The modifications are called mutation operators or mutations. Typical mutation operators alike mimic programming errors such as using wrong operator or variable name: replace each operand by every other syntactically legal operand, or modify expressions by replacing operators and inserting new operators, or delete entire statements. The size of the mutation operators set is determined by the language of the program being tested and the mutation system used[15].

Below are given result for mutation testing (Table 4).

Table 4 Mutation testing results

<table>
<thead>
<tr>
<th>Mutants killed, %</th>
<th>All paths</th>
<th>All nodes</th>
<th>Main paths</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>100</td>
<td>92.59</td>
<td>82.75</td>
</tr>
</tbody>
</table>
Depending on testing goals tester can choose main path, all nodes, or all paths approaches:

- The main path set kills 82.75% of mutants with relatively low execution duration of about 1 minute;
- The all nodes set gives 9.84% increase of killed mutant count and execution duration takes almost 8 minutes;
- The all paths’ set kills all mutants but requires relatively high execution duration up to 40 minutes;

The example shows that proposed method allows choosing suitable testing strategy: the trade-off between testing duration and test quality.

4 Conclusions

The GUI testing method based on tests’ meta-model was presented in this article. The proposed method allows to decrease the required manual effort for test creation, simplifies tests maintenance when changes take place, helps to clarify testing scope and objectives, allows to generate tests automatically, and enables regression testing.

The proposed GUI testing method allows making the trade-off between testing duration and test quality. 100% of mutants were killed using all paths’ set. This is 20% greater than main path set’s result, but execution duration was 40 times longer. While code coverage of main paths’ set is 91.49% which is lower by 5.95% than all paths’ set.

References