An Extensible Heuristic-Based Framework for GUI Test Case Maintenance

Scott McMaster and Atif M.Memon
University of Maryland
College Park, MD, USA
{scottmcm,atif}@cs.umd.edu

***Abstract*—Graphical user interfaces (GUIs) make up a large portion of the code comprising many modern software applications. However, GUI testing differs significantly from testing of traditional software. One respect in which this is true is test case maintenance. Due to the way that GUI test cases are often implemented, relatively minor changes to the construction of the GUI can cause a large number of test case executions to malfunction, often because GUI elements referred to by the test cases have been renamed, moved, or otherwise altered. We posit that a general solution to the problem of GUI test case maintenance must be based on heuristics that attempt to match an application's GUI elements across versions. We demonstrate the use of some heuristics with framework support. Our tool support is general in that it may be used with other heuristics if needed in the future.**

***Index Terms*—Graphical user interfaces, Test suite management, Testing strategies, Testing tools**

I. Introduction

Graphical user interfaces (GUIs) make up a large portion of the code comprising many modern software applications. Testing of GUIs differs significantly from testing of traditional software. Consequently, a number of specialized techniques and test case representations for GUI test automation have been developed.

A. Capture/Replay

One common approach to GUI test automation is *capture/replay*, where end-user gestures such as mouse movements and keystrokes are recorded and played back. Capture/replay techniques have the advantage of making it simple for testers without significant programming skills to automate their test cases. However, these test cases often cause difficulty during software maintenance and regression testing, because relatively minor changes to the GUI can cause a test case to *break*, or, cease to be executable against an updated version of the software. When such a situation occurs, a large manual effort is often required to repair some subset of the cases in a test suite, or worse yet; regression testing is allowed to suffer.

B. Elements and Actions

An alternative approach models a GUI test case as a sequence of *actions* on *GUI elements* (sometimes called "widgets" or "controls"). Such a model is used in the event-based test cases of GUITAR, for example [3]. Consider the GUI dialog in Figure 1, a fairly typical "Find" dialog implemented using the Java Swing toolkit[1].

Figure 1: Initial Version of Find Dialog

An example test case for this dialog might perform the following actions:

1. Set the text in the textbox to be "GUI".
2. Check on the "Case-Sensitive" checkbox to select a case-sensitive search.
3. Click the "Find Next" button to execute the "find" operation.
4. Click the "Cancel" button to close the dialog.

If the GUI elements in the dialog have well-known unique identifiers, such as "FindTextBox" for the edit area where the text to find is to be entered, such a test case can be represented as the set of element-id/action pairs in Figure 2:

1. ["FindTextBox", "setText('GUI')"]
2. ["CaseSensitiveCheckBox", "click"]
3. ["FindButton", "click"]
4. ["CancelButton", "click"]

Figure 2: Example Test Case for Find Dialog

Consider what happens when the "Find" dialog of Figure 1 is enhanced to include "Replace" functionality as in Figure 3.

It is intuitively obvious that it is both desirable and possible to execute the sequence of actions in Figure 2 against the new version of the GUI. But assuming that we are not able to rely on each GUI element having a unique identifier (discussed in more detail below), this enhancement to the original "Find" dialog poses at least two potential problems for traditional test case automation tools when running this test case. First, the check box used in the second step has had its label changed

[1] http://java.sun.com/javase/6/docs/technotes/guides/swing/

Figure 3: Modified Find Dialog

from “Case-Sensitive” to “Match Case”. Second, the new elements added to the dialog (such as the new border around the checkboxes) will have changed the positions of most (if not all) of the previously existing elements.

In [1], one of us (Memon) takes on the problem of GUI test suite maintenance from the standpoint of *repairing transformations* on the *event flow graph* (EFG). This approach was shown to be quite successful in a family of empirical studies using multiple versions of four open-source applications. In order to correctly make repairing transformations, however, it must be possible to correctly identify GUI elements in program version *(n+1)* that correspond to events in program version *n*. In practice, certain modifications to the GUI and its underlying code make this difficult, particularly if each GUI element is not assigned a unique identifier.

Unique identifiers for GUI elements are widely known to enhance testability of GUI applications [4], but we have observed that in practice, developers often fail to properly maintain them nonetheless. As a result, test automation frameworks must sometimes fall back on other properties to heuristically identify GUI elements for playback, such as textual labels or relative positions in the hierarchy of components on the dialog. Test cases that rely on these properties can be substantially less stable. Relying on textual labels can cause significant implementation difficulties when the application is localized into several languages, and even in a single language, labels are quite apt to change. And positions in the control hierarchy can change drastically as enhancements are made. Clearly, arbitrary modifications to a GUI can render all such heuristics useless, but in practice, we find that GUI changes in incremental software versions are minor enough that application of certain heuristics can allow test cases to move forward with the software with little-to-no manual maintenance.

In order to increase the robustness of automated GUI test execution, and to enable techniques such as repairing transformations, test case playback tools such as the one described in [2] sometimes use multiple heuristics to select GUI elements for manipulation. Indeed, we posit that a general solution to the problem of GUI test case maintenance must be based on heuristics that attempt to match an application’s GUI elements across versions. However, the effectiveness of these heuristics (and sets thereof) under different scenarios of GUI software evolution has not been formally studied.

In the remainder of this paper, we present a model for studying the efficacy of heuristics for GUI element identification. We discuss preliminary tooling support and close with a research agenda to evaluate the effectiveness of sets of GUI element identification heuristics for use in test case maintenance. If successful, this work will yield new insights into how GUI software evolves in practice, as well as provide GUI test automation tool developers with new insights into how to make their tools’ replay capabilities more robust.

II. THE GUI ELEMENT IDENTIFICATION PROBLEM

In this section, we introduce a model of the problem of identifying functionally equivalent GUI elements from version *n* of an application in version *(n+1)*.

A. Example

As a motivational example, consider the dialog from Figure 1. The GUI elements from that dialog can be viewed hierarchically Swing Explorer[2] as Figure 4:

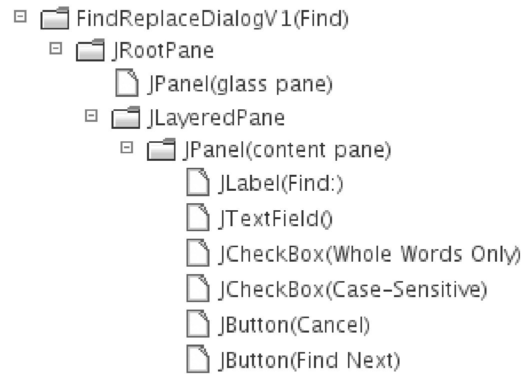

Figure 4: Find Dialog GUI Elements

When the dialog is enhanced as shown in Figure 3, the hierarchy changes to that in Figure 5.

Informally, for each node in the tree in Figure 4 (version *n*), we seek to heuristically identify the functionally equivalent node in Figure 5 (version *(n+1)*), where the goal is to correctly identify enough nodes to permit successful playback in the later version of a significant number of test cases defined against the earlier version.

Consider the test case in Figure 2. To execute this test case, we need to correctly map four GUI elements between Figures 4 and 5. Assuming that we have no unique identifiers consistent between versions, we can first surmise that the“FindTextBox” is the JTextField which follows the JLabel with the text “Find:”. The “FindNext” and “Cancel” buttons (with consistent labels) are straightforward. The check box indicating the case-sensitivity of the search is more difficult to automatically identify, because its text has changed from “Case-Sensitive” to “Match Case”. So we presume that it is the JCheckBox *following* the JCheckbox labeled “Whole Words Only”, which we can identify because *its* label is unchanged. In all, the heuristics in Table 1 have been applied.

[2] https://swingexplorer.dev.java.net/

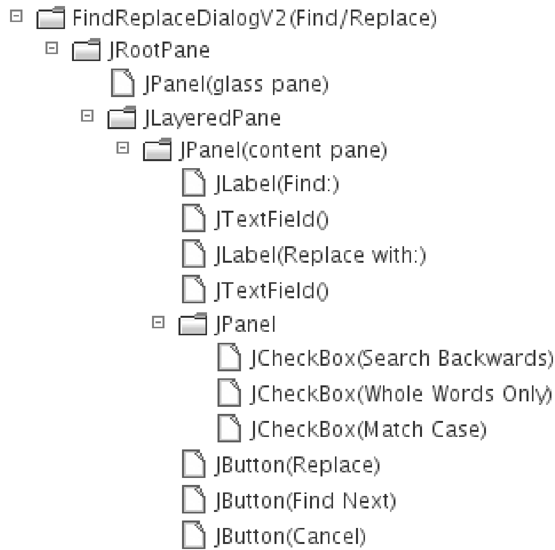

Figure 5: Find/Replace Dialog GUI Elements

Heuristic Name	*Description*
SameLabel	Element has the same label.
SamePreviousSibling	Element has the same previous sibling element.
SameNextSibling	Element has the same next sibling element.
SameType	Element is of the same type (button, checkbox, etc.).

Table 1: Sample Heuristics

B. *Definitions*

We define *actionable* GUI elements as those elements which can have actions performed on them (clicks or accepting keystrokes, for example) which are meaningful in the context of the application. Typically, in most GUI toolkits these are buttons, menus, text boxes, etc. Actionable GUI elements are paired with actions to form a GUI test case.

Non-actionable GUI elements such as panels and textual labels exist for layout, adornment, or other "passive" purposes. Because non-actionable elements do not appear as inputs to GUI test cases, it is not critical that we identify changes to non-actionable elements in order to achieve our goal of making a broken test case executable. However, when applying certain heuristics, identifying non-actionable elements can aid in identifying the actionable ones. Additionally, it may be important to identify non-actionable elements to properly verify GUI test case outputs, which will be a subject of future work.

Let W be the unordered set of actionable GUI elements $<e_1..e_m>$ that exist in a specific version of a single window in a GUI application. Let W' be a subsequent version of the same window with actionable elements $<e'_1..e'_n>$. (Note that *m* may not equal *n* if GUI elements were added and/or removed, changing the total count, between versions.) Then, the *GUI element identification problem* can be simply stated as:

For each actionable GUI element e_i in W, find a corresponding (possibly modified) element e'_j in W' whose actions implement the same functionality.

More formally, we seek to accurately assign each element from W and W' to exactly one of the following data structures:

- *Deleted* – Set of elements from $<e_1..e_m>$ with no corresponding element in $<e'_1..e'_n>$. These elements have been removed from the GUI window in the new version.
- *Created* – Set of elements from $<e'_1..e'_n>$ with no corresponding element in $<e_1..e_m>$. These elements have been added to the GUI window in the new version.
- *Maintained* – Set of mappings from $<e_i \text{ -> } e'_j>$. These are elements from the original GUI window which have been preserved (but possibly modified) in the new version.

Test cases that use elements in the *Deleted* set are not immediately executable in the new version, but may be made executable under certain circumstances using repairing transformations [1]. By definition, elements in the *Created* set may not appear in previously existing test cases. Therefore, the GUI element identification problem is primarily interested in accurately calculating the *Maintained* set. As discussed in the previous section, this problem generally requires heuristic approaches in the absence of a model where each element has a programmer-defined unique identifier. In practice, the widest array of GUI modifications can be accommodated by using a process that applies multiple heuristics. Such an approach is described in the next section.

III. AN INITIAL APPROACH

To explore the effectiveness of various heuristics in GUI element identification and test case maintenance, we need a framework that allows us to easily define heuristics and compose them into sets. This framework must have tool support to facilitate experimentation. In this section, we present an overview of such a tool, called GUIAnalyzer. GUIAnalyzer is implemented in Java and works against applications built with the Swing toolkit.

GUIAnalyzer uses the Jemmy library[3] to create a model of a GUI, which can be analyzed in memory and/or persisted as XML. Two such models (presumably different versions of the same application) can then be reconciled using heuristics.

The key concept in GUIAnalyzer is our AbstractHeuristic, which serves as a base class for specific heuristic implementations and is responsible for determining whether two components from different GUI models match. Seven heuristics are currently implemented, including those listed in Table 1. Heuristics can be configured as prerequisites for other heuristics. Thus, it is possible to state, for example, that two elements from different GUI versions with the same previous element only match if the elements are of the same type (see heuristic #5 in Figure 6, below).

[3] https://jemmy.dev.java.net/

Heuristics are grouped into a precedence-ordered list called a *heuristic set*. Multiple passes over the heuristic set are made until no additional components from the new version are identified. If multiple heuristics are capable of identifying an element during a pass over the heuristic set, the identification made by the highest-priority heuristic is applied to the element. An example heuristic set (with prerequisite heuristics indented) appears in Figure 6. In this example, the highest-priority heuristic is the SameNameHeuristic, which matches elements whose names are consistent across versions.

```
1: SameNameHeuristic
2: SameTextHeuristic
      --SameTypeHeuristic
3: SameTooltipHeuristic
      --SameTypeHeuristic
4: SameNextAndPreviousSiblingHeuristic
5: SamePreviousSiblingHeuristic
      --SameTypeHeuristic
6: SameNextSiblingHeuristic
      --SameTypeHeuristic
```

Figure 6: Sample Heuristic Set

Applying this heuristic set to reconcile models of Figures 1 and 2 yields the (abbreviated) output of Figure 7 and calculates the *Created*, *Deleted*, and *Maintained* sets of GUI elements. From the original Find dialog, all six actionable controls are correctly identified in the updated version. Thus, model-based test cases for the original dialog can be expected to execute successfully against the modified dialog.

IV. RESEARCH AGENDA

We plan to incorporate and evaluate the effectiveness of additional heuristics, heuristic sets, and heuristic set priorities in the GUI element identification problem. This will require us to improve the capabilities of the prototype GUIAnalyzer. Increasing the level of detail in the GUI model will enable us to implement additional, more advanced heuristics

The heuristic-based approach to the GUI element identification problem and to GUI regression test execution will be evaluated in a family of empirical studies. Multiple versions of individual GUI windows and dialogs used in the four open-source GUI applications from [1] will be used as subjects for these studies. Several heuristic sets will be defined and executed against each subject, and their relative effectiveness will be compared. To evaluate the effectiveness of different sets of heuristics, we will define and evaluate two quantities related to the *Maintained* mappings – percentages of false positives and false negatives.

False positives are the elements that a set of heuristics places in *Maintained* which actually either belong in *Deleted* or are mapped to the wrong $e_i \in W$. *False negatives* are elements from W that have a corresponding element in W' but do not appear in *Maintained* with the correct mapping. False positives and false negatives can both cause a maintained test case to fail to execute against a modified version of a GUI and therefore must be minimized.

```
Applying heuristics, pass 1
javax.swing.JLabel:Find: identified by
SameTextHeuristic as javax.swing.JLabel:Find:
javax.swing.JCheckBox:Whole Words Only
identified by SameTextHeuristic as
javax.swing.JCheckBox:Whole Words Only
javax.swing.JButton:Find Next identified by
SameTextHeuristic as javax.swing.JButton:Find
Next
javax.swing.JButton:Cancel identified by
SameTextHeuristic as javax.swing.JButton:Cancel
javax.swing.JTextField:null identified by
SamePreviousSiblingHeuristic as
javax.swing.JTextField:null
javax.swing.JCheckBox:Match Case identified by
SamePreviousSiblingHeuristic as
javax.swing.JCheckBox:Case-Sensitive
Applying heuristics, pass 2
Done
```

Figure 7: GUIAnalyzer Reconciliation Output

Large GUI modifications (where relatively many elements are added, removed, and renamed) will clearly have an adverse effect on the accuracy of any heuristic-based approach to element identification. We will attempt to quantify this impact by simulating different rates of change in the GUI.

An analysis of individual unidentified and misidentified GUI elements will be performed, with the goal of improving the heuristics and heuristic sets. An additional consequence of this analysis should be an improved knowledge of what types of GUI modifications cause difficulties for GUI test case execution. Ideally this will lead to a set of recommendations for developers on how to make GUI changes in ways that ease test case maintenance.

We will also extend the GUIAnalyzer with new approaches. One possible technique would be to apply multiple heuristic sets simultaneously, compare the results, and resolve any inconsistencies to produce a more accurate final output. Another approach would be to do an online evaluation of the executability of test cases against the identified GUI elements. This could improve accuracy by catching cases where, for example, a button was misidentified as a label. We hope to integrate the technologies developed in the GUIAnalzyer to improve the GUITAR suite of GUI regression testing tools [2].

REFERENCES

[1] A. Memon. Automatically repairing event sequence-based GUI test suites for regression testing. ACM Transactions on Software Engineering and Methodology. Vol. 18, no. 2, Article 4, October 2008.

[2] A. Memon and Q. Xie. Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software. IEEE Transactions on Software Engineering, vol. 31, no. 10, pp. 884-896, October, 2005.

[3] A. Memon. An event-flow model of GUI-based applications for testing. Software Testing, Verification and Reliability, vol. 17, no. 3, 2007, pp. 137-157, John Wiley and Sons Ltd.

[4] A. Ruiz, and Y.W. Price. GUI testing made easy. Practice and Research Techniques, 2008. TAIC PART '08. Testing: Academic & Industrial Conference, pp 99-103, Windsor, UK., August, 2008.