This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2300857, IEEE Transactions on Software Engineering

An *Observe-Model-Exercise** Paradigm to Test Event-Driven Systems with Undetermined Input Spaces

Bao N. Nguyen *Student Member, IEEE,* Atif M. Memon *Member, IEEE*

Abstract—System testing of software applications with a graphical-user interface (GUI) front-end requires that sequences of GUI events, that sample the application's input space, be generated and executed as test cases on the GUI. However, the context-sensitive behavior of the GUI of most of today's non-trivial software applications makes it practically impossible to fully determine the software's input space. Consequently, GUI testers—both automated and manual—working with undetermined input spaces are, in some sense, blindly navigating the GUI, unknowingly missing allowable event sequences, and failing to realize that the GUI implementation may allow the execution of some disallowed sequences. In this paper, we develop a new paradigm for GUI testing, one that we call *Observe-Model-Exercise** (OME*) to tackle the challenges of testing context-sensitive GUIs with undetermined input spaces. Starting with an incomplete model of the GUI's input space, a set of coverage elements to test, and test cases, OME* iteratively *observes* the existence of new events during execution of the test cases, expands the *model* of the GUI's input space, computes new coverage elements, and obtains new test cases to *exercise* the new elements. Our experiment with 8 open-source software subjects, more than 500,000 test cases running for almost 1100 machine-days, shows that OME* is able to expand the test space on average by 464.11%; it detected 34 faults that had never been detected before.

1 Introduction

1.1 Motivation & Background

Consider how graphical-user interface (GUI)-based software applications are tested today [1], [2], [3], [4]. Most often, a tester is given a set of tasks with the job of verifying that these tasks can be performed using the software; and that the software does not "behave badly". Sometimes, the tester is also given a set of use cases with *high-level* descriptions of their steps (*e.g.*, *"save the file"*, *"load the document"*). The tester executes these high-level steps by using GUI widgets on which events can be performed. The tester does this either manually, or by writing test scripts using tools such as jfcUnit [5], Abbot Pounder [6], UI Automation classes in the .NET Framework[1], and Quality Test Pro (QTP)[2], or by using capture/replay (record/playback) tools such as Marathon[3] and Jacareto[4], or a combination thereof. The choice of which widgets/events[5] to execute is most often left to the tester. For example, the tester may perform *"save the file"* in one of three ways: (1) click on the *Save* icon in the toolbar, (2) use menu items *File→Save*, or (3) use alternative menu items *File→Save_As* followed by a file name in the text-box. During this process, the tester may "discover" new ways to combine certain events to perform a task.

The tester does not have a complete picture of the GUI's input space, i.e., the set of all possible sequences of user-interface events. The tester is never supplied a *blueprint* of the GUI or its set of allowable workflows. In principle, the tester has no idea of what event sequences were missed during the testing process. End users may execute untested event sequences and encounter failures. Moreover, the implemented GUI may allow event sequences that the designers never wanted to allow. But there is no way for a tester to determine which sequences are missed and which should not be allowed.

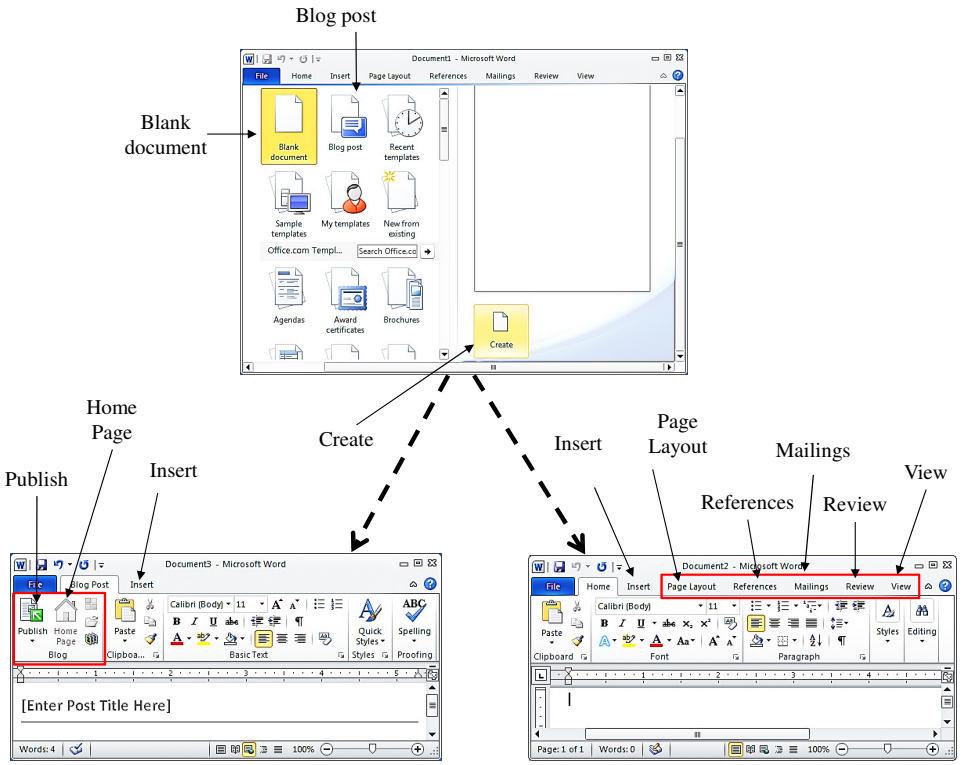
Human testers have the experience and domain knowledge to navigate through and verify the correctness of such systems with unknown or partially known input spaces; *i.e.*, systems that lack the aforementioned *blueprint* [1]. For example, a human tester who clicks on a button labeled *Close* expects something (most often the current window) in the GUI to close, and all its constituent events to become subsequently unavailable. Similarly, in MS Word 2010, Figure 1(a), a human tester *creating a new document* will expect the menu structure (and hence set of available events) to be different for a blog type of document versus a conventional blank document.

However, automated test harnesses/tools lack the experience and domain knowledge of humans. Without a representation of allowed and disallowed work-

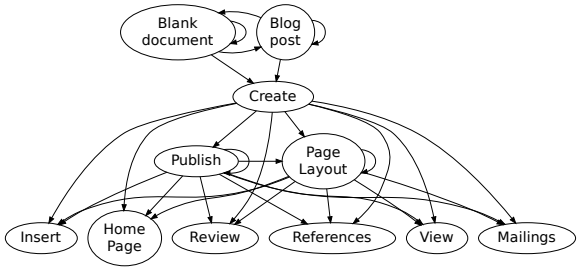
• *The authors are with the Department of Computer Science, University of Maryland, College Park, MD 20742. E-mail: {baonn, atif}@cs.umd.edu.*

1. http://msdn.microsoft.com/en-us/library/ms747327.aspx
2. http://en.wikipedia.org/wiki/HP_QuickTest_Professional
3. http://www.marathontesting.com/
4. http://jacareto.sourceforge.net
5. An event (*e.g., click-on-Cancel-button*, *select-Radio-button*) is the action that a user performs on a widget (*e.g., Cancel button*, *Radio button*). Whenever the context is clear, we use the terms "event" and "widget" interchangeably; *e.g.*, when we say "the user performs the *Cancel* event" we actually means that "the user performs the action *click-on* the *Cancel* button widget."

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2300857, IEEE Transactions on Software Engineering



(a) *Create* (top) is context-driven; document type, either *Blank document* or *Blog post* creates different events (bottom-left for Blog; bottom-right for document).



(b) The partial event-flow graph.

Fig. 1. MS Word 2010 motivating example.

flows, they are unable to reason their way out of an unexpected situation. This causes many GUI testing tools to (1) rely on a human tester: for example, capture/replay to recreate manually pre-recorded (or programatically coded) event sequences; (2) perform very limited automated testing tasks: for example, tools such as Android's Monkey[6] and Eclipse-based GUIdancer[7] perform simple random walks of the user interface, executing events as they encounter them; and detect crashes. These approaches are insufficient with the result that GUI quality is often compromised [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [2], [19], [20], [21].

6. http://developer.android.com/guide/developing/tools/monkey.html

7. http://www.guidancer.com

1.2 Our New Approach – OME*

We have been developing a third type of approach based on a *model* of the GUI. In earlier work, we developed a directed graph model, called event-flow graph (EFG) [22]. An EFG is essentially a formal model of the GUI – a type of blueprint. Such a blueprint contains all the workflows allowed, and in some sense, disallowed, by the GUI's structure and events. More specifically, it is a directed graph structure with nodes that represent events, and edges that represent the `follows` relationship; "event e_x `follows` event e_y" means that e_x may be executed *immediately* after e_y along some execution path in the GUI. This is represented as an edge from node n_y to n_x, where n_x represents e_x and n_y represents e_y. Part of the EFG for the GUI of Figure 1(a) is shown in Figure 1(b). We see edges from *Create* to *Publish*, *Home Page*, and *Insert*; and to *Page Layout*, *References*,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2300857, IEEE Transactions on Software Engineering

Mailings, *Review*, *Insert*, and *View*; each of these events may be executed immediately after *Create* in the GUI.

It is not easy to create a complete and accurate EFG for a GUI. In past work, we developed a technique called *GUI Ripping* to create an approximation of the EFG [23]. The *GUI Ripper* dynamically traverses the GUI, opening windows, performing events, keeping track of all windows seen, and using algorithms to construct an EFG. The goal of *Ripping* is not to test the GUI's events; rather, it attempts to open as many windows as possible, extracting events from each, and computing the `follows` relationship.

Our most successful Ripper to date uses a depth-first traversal (DFS) of the GUI, i.e., it starts with the *Main* window, extracts the set of all the widgets on which events can be performed, stores this set, and performs one of these events. If this event invokes a new window, then it is recursively processed; otherwise the next event is executed. If no new windows open, then the Ripper closes the current window and restarts, using the previously saved sets of events to explore alternate parts of the GUI.

In our example of Figure 1(a), starting with the top-most window, the *Ripper* would select *Blank document* (because it is the first icon in the list) and then perform *Create*. This opens a new window. After the *Ripper* has finished interacting with this window and all of its sub-windows, it returns to the top window and performs all events that it had missed earlier. Because it has already performed *Create*, it will not do so again. Hence, the bottom-right state of the window will never be reached; events *Page Layout*, *References*, *Mailings*, *Review*, and *View* will be missed. In general, because the *Ripper* performs a generalized, fully automatic traversal, it may miss application-specific parts of the GUI that are "guarded" with very specific inputs, such as a password; or behavior that requires very specific combinations of inputs, *e.g.*, context-sensitive menu items.

Having encountered the above situation, one could, in retrospect, redesign the *Ripper* to handle it. However, there are many such situations and it is impossible to devise a general algorithm that works for all. The only general algorithm that will work in all situations will need to use exhaustive search strategy, e.g., one that enumerates all non-cyclic paths between the 2 GUI parts, in which case the bottom-right state of the window in Figure 1(a) may be reached. However, although exhaustive search seems to be feasible in a relatively simple event-flow graph like Figure 1(b), is not applicable in real-world applications, as we will show in subsequent sections.

In this paper, we tackle the challenges presented by the context-sensitive behavior of GUIs. To provide focus, we only consider the behaviors directly caused by the order of GUI events. Other potential causes of "context-sensitivity" such as timing and multiple-user profiles are left for future work. Specifically, we develop a new paradigm for GUI testing, one that we call *Observe-Model-Exercise** (OME*). The key feature of OME* is its opportunistic use of test execution for model enhancement. More specifically, we now *observe* the existence of new events either during *Ripping* or test execution, create or enhance our EFG+ *model* – an extension of our EFG model, and *exercise* the newly observed GUI events in test cases using test adequacy criteria. As new test cases are generated and executed, their executions are simultaneously used to observe new events, which are added to the model and used to compute new test requirements, and subsequently obtain additional tests. The "*" is due to the iterative nature of the entire approach. The iteration ends when no new enhancements are made to the model.

Our paper makes 5 intellectual contributions, corresponding to 5 challenges:

Challenge 1: It is challenging to generate particular event sequences to replicate context-sensitive behaviors. Because events and event sequences are context sensitive, they may have been observed due to the execution of particular prior event sequences. *Contribution 1:* We make use of a new context-aware mapping that maintains information about the event sequences that were used to reach model elements.

Challenge 2: It is challenging to devise new event sequences that reveal new parts of the input space and enhance the model. *Contribution 2:* We simultaneously extract new GUI model elements—events and `follows` relationships—during test execution.

Challenge 3: It is challenging to identify new events, *i.e.*, to determine whether an event has already been seen before. *Contribution 3:* We develop a unique signature for each widget and matching heuristics to help detect new widgets.

Challenge 4: It is challenging to incrementally make changes to the model to add new elements. *Contribution 4:* We develop new operations on the EFG+ to incrementally enhance it as new information becomes available.

Challenge 5: It is challenging to incrementally generate new test cases. *Contribution 5:* We develop an algorithm to compute new test requirements from recent model enhancements and generate test cases to satisfy the requirements.

Our final contribution is our experiment, involving 8 open-source applications on which we executed over 500,000 test cases that consumed almost 1100 machine days. We compared OME* with the current state-of-the-art. We saw significant improvements, both in terms of new areas of the input space that we explored (1044% for one subject application), and fault detection – we fully automatically detected 34 new faults in these applications that manifested as crashes.

In the next section, we present a step-by-step overview of OME* via an example. In Section 3, we present the new algorithms developed to realize

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2300857, IEEE Transactions on Software Engineering

OME*. We then discuss our experiment in Section 4, present related work in Section 5, and finally conclude with a discussion of future work in Section 6.

2 OVERVIEW

Because this work leverages several of our previously reported techniques [23], [24], [25], [26] we feel that it is appropriate to present an overview, with a running example, to demonstrate the prior work as well as the new OME* paradigm. Figure 2(a) presents the GUI of our running example, motivated by the MS Word example that we showed in the previous section. It consists of four events in the *New document* window. Events e_1, e_2, and e_3 are *non-structural events*—they do not open/close windows/menus—that manipulate radio buttons and checkbox states. Selecting the *Blog post* radio button enables e_3. Event e_4 opens a new *modal window*[8] entitled either *Blog post* or *Blank document* depending on the states of the radio buttons in the *New document* window. If the new window's title is *Blog post*, then it is shown with non-structural events e_5, e_6, and e_7. Otherwise, when its title is *Blank document*, it is shown with non-structural events e_5 and e_8. Note that event e_5 is available in both states. Finally, checking the *Already have a home page* check box enables e_7.

Our overall goal is to test this running example. We summarize our process using the following steps:

Step 1: *Running the Ripper:* We start by running our *Ripper* on the application—in its start state—to obtain its EFG. Events e_1, e_2, e_3, and e_4 are all available in the main window; their states, each represented as a set of triples of widget, property, and value, are shown in Figure 2(a). Because of their availability in the GUI's start state, these events form the *initial nodes* set, I. The *Ripper* incorporates these nodes into the EFG; they are shown as shaded ovals in Figure 2. The Ripper then starts executing the encountered events one by one: e_1 followed by e_2, then e_3, and e_4. After events e_1, e_2, and e_3, the Ripper determines that they are non-structural events because no window is opened or closed; the `follows` relationships are then computed according to the algorithms presented in earlier work [23] and added to the EFG. Event e_4 opens a new window; because of the selected state of the *Blog post* radio button and checked state of e_3, the new window is titled *Blog post* with three events e_5, e_6, and e_7, all enabled. They are all executed but no new window opens. Their `follows` relationships are then computed and added to the EFG. The final EFG after the *Ripping* phase is shown in Figure 2(c).

Step 2: *Generating and executing test cases:* In this example, we will assume that we want to cover all EFG edges as our test criterion; we have used this criterion in earlier work (*e.g.*, [22], [24], [25]). There are 24 edges in the EFG of Figure 2(c), yielding 24 test cases. The process for test case generation has been explained in earlier reported work [23]. Edges are selected one by one; for each edge (e_x, e_y), a path is computed—using a method called *prepend_context()*—from one of the *initial nodes* to (e_x, e_y), yielding a test case.

In previous work, because we lacked specific information about the events, our *prepend_context()* method could only rely on the EFG's topology to obtain a path from one of the nodes in the initial nodes set to the edge in question. For efficiency reasons, we used the shortest path. For example, if we select the edge (e_5, e_7), the shortest path to its first event is $\langle e_4 \rangle$, yielding a test case $\langle e_4, e_5, e_7 \rangle$. However, execution of this test case stops at e_5 because e_7 is disabled. This presents us with Challenge 1 mentioned in Section 1: *it is challenging to generate particular event sequences to replicate context-sensitive behavior of events.*

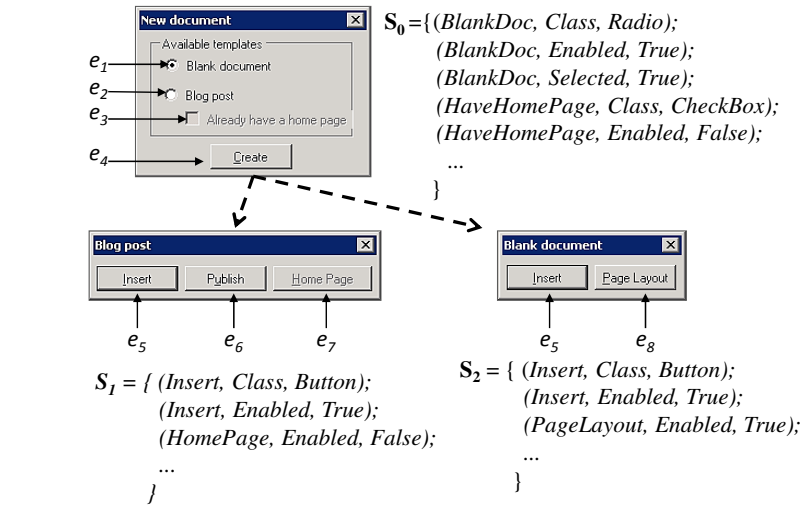
In our work presented in this paper, we now maintain a context-aware mapping between edges and paths to edges that have previously been seen to be executable. This mapping, together with our previous EFG model forms our new EFG+ model. Using the mapping, partly seen in Figure 2(b), the entry for edge (e_5, e_7) is $\langle e_1, e_2, e_3, e_4 \rangle$ because this was the executable path seen during *Ripping*. Hence, we will get $\langle e_1, e_2, e_3, e_4, e_5, e_7 \rangle$ as our test case. All 24 test cases are generated in this fashion, guaranteeing that all 24 EFG edges will be covered. These 24 test cases are then executed.

From our knowledge of the GUI, we know that we have yet to test event e_8. However, our *Ripper* does not even know of the existence of e_8. We need ways to drive the GUI into such a state that e_8 is exposed, tested, and added to our EFG model. To do so would, in principle, require that we traverse all possible paths in the GUI. This presents us with Challenge 2: *it is challenging to devise new event sequences that reveal new parts of the input space and enhance the model.*

In our work presented in this paper, our approach to handle this challenge is to *simultaneously* use test execution for model enhancement. For example, one of our 24 test cases is $\langle e_1, e_4 \rangle$, whose execution will open the *Blank document* window with events e_5 and e_8. If at this time, we can recognize e_8 as a new yet-to-cover event, we can devise ways to cover it. We have developed mechanisms to add newly discovered events during test execution to our EFG. This presents us with Challenge 3: *it is challenging to identify new events/widgets, i.e., to determine whether an event/ widget has already been seen.* For example, we know that e_5 is the *Insert* button that we have seen earlier. On the other hand, we have never seen e_8 before, the *Page Layout* button. Do we make a determination based solely on the "text labels" of these widgets? This would cause problems as many widgets in the GUI have the same text label (*e.g.*, *OK*, *Cancel*). We have

8. A modal window, once invoked, restricts the focus of the user to the events within the window, until explicitly closed.

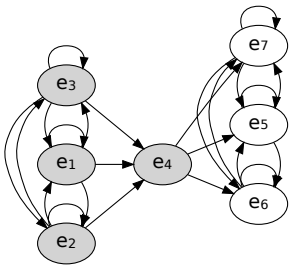
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2300857, IEEE Transactions on Software Engineering



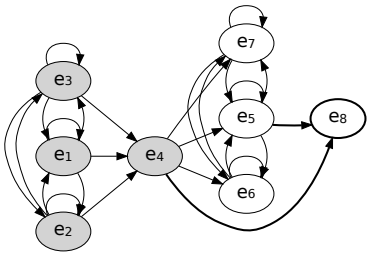
(a) GUI of running example

Edge	Path to edge
...	...
(e_1, e_2)	$NONE$
(e_2, e_3)	$\langle e_1 \rangle$
(e_3, e_4)	$\langle e_1, e_2 \rangle$
(e_4, e_5)	$\langle e_1, e_2, e_3 \rangle$
(e_5, e_7)	$\langle e_1, e_2, e_3, e_4 \rangle$
...	...
...	...

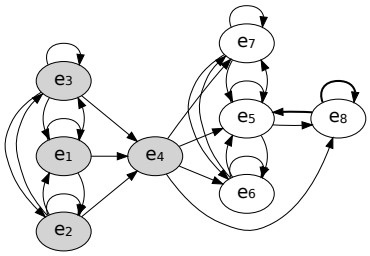
(b) Context-Aware Mapping



(c) EFG after *Ripping*



(d) EFG after e_8 is observed



(e) EFG after e_8 is executed

Fig. 2. Running example.

developed mechanisms to assign unique *signatures* to each widget; and heuristics to determine the uniqueness of the signatures.

Step 3: *Iteratively enhancing the EFG model, and generating and executing new test cases:* Having developed the ability to identify newly encountered widgets during test execution, we face Challenge 4: *it is challenging to incrementally make changes to the model to add new elements.* To date, we have developed algorithms to create the EFG in one pass. In our work presented in this paper, we develop techniques to incrementally enhance the EFG. The new EFG after the addition of e_8 is shown in Figure 2(d). Because we observed e_8 after the execution of e_4, we know that "e_8 `follows` e_4" which is why we have a new edge from e_4 to e_8. Moreover, because we know that e_5 is not a structural event, *i.e.*, it does not open a new window nor does it close the current window, e_8 could potentially `follow` e_5; hence, we also add the edge (e_5, e_8) to the EFG.

Now that we have two new not-yet-covered edges, (e_4, e_8) and (e_5, e_8), we need to generate test cases to cover them so that we can satisfy our test criteria. This presents us with Challenge 5: *it is challenging to incrementally generate new test cases*. In our work presented in this paper, we have developed an algorithm to compute new test requirements from changes to the EFG+ model and generate test cases to satisfy the requirements. Using that algorithm, assume that we get test cases $\langle e_1, e_4, e_8 \rangle$ and $\langle e_1, e_4, e_5, e_8 \rangle$, to cover (e_4, e_8) and (e_5, e_8), respectively. These test cases are executed; e_8 is determined to be a non-structural event; two new `follows` relationships are added; these are new EFG edges (e_8, e_8) and (e_8, e_5) (new EFG shown in Figure 2(e)). As before, we now need to cover these new edges via new test cases. No changes are made to the EFG model during the execution of these test cases, and so the test process is complete, having satisfied the test criterion of covering all edges.

Even though we used a small example, we were able to show how OME* is used to discover new parts of the input space and exercise them. However, as we will see in our evaluation, Section 4, it is possible that we may not be able to automatically exercise all model elements that we observe; in such cases, manual intervention is needed.

3 Realizing the OME* Paradigm

We now discuss the new models, algorithms, and techniques that we developed to realize the new OME* paradigm. We structure our discussion around the contributions listed in Section 1.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2300857, IEEE Transactions on Software Engineering

3.1 Contribution 1: Context-Aware Mapping

In our past work, we relied on the "shortest-path algorithm" to obtain a sequence of events starting with a node in I, the initial nodes set, to the model element (*e.g.*, EFG nodes, edges) that we are trying to exercise. This is because of the limitation of the EFG model, which does not maintain context for each event, a deliberate design decision in order to keep the model compact and scalable. Such a design worked well in practice in our past work because of the non-context-sensitive nature of the applications we tested. However, as demonstrated by the example of edge (e_5, e_7) in the previous section, the shortest path does not always yield an executable event sequence, especially when GUI behavior is extremely context sensitive. To address this problem, we now maintain a new context-aware mapping between model elements and executable event sequences that have previously been successfully used to exercise these elements. Intuitively, during *Ripping* and test case execution, if we observe a certain model element is available after the execution of a particular event sequence, we create a new mapping to use later to reach the element.

Consider, for example, the execution of event sequence $\langle e_2, e_3, e_4 \rangle$ on the GUI of Figure 2(a). Recall that our coverage elements are EFG edges; hence our mapping will be between EFG edges and event sequences used to reach them. We start with the execution of e_2, after which the events e_1, e_2, e_3, and e_4 are available for execution. We execute e_3, which does not change the set of available events. We execute e_4, after which events e_5, e_6, and e_7 are available. The same information, put in terms of the model elements, EFG edges, can be thought of as: "*edges* (e_4, e_5), (e_4, e_6), *and* (e_4, e_7) are reachable via the event sequence $\langle e_2, e_3 \rangle$." If, in the future, we want to cover these edges, we can use this information. This is precisely what we record in our mapping. Hence we see entries for the edges (e_4, e_5), (e_4, e_6), and (e_4, e_7) in our partial mapping shown in Table 1; there are several more, *e.g.*, (e_3, e_4), which needs e_2. There are also several *NONE* entries, which means that the first element in the edge is in I, and the edge is enabled, making it trivial to reach it from the initial state.

TABLE 1
Partial Mapping.

Edge	Path to edge
(e_4, e_5)	$\langle e_2, e_3 \rangle$
(e_4, e_6)	$\langle e_2, e_3 \rangle$
(e_4, e_7)	$\langle e_2, e_3 \rangle$
(e_2, e_1)	*NONE*
(e_3, e_4)	$\langle e_2 \rangle$
(e_3, e_1)	$\langle e_2 \rangle$
(e_2, e_2)	*NONE*
(e_2, e_4)	*NONE*
...	...

We now describe the mapping formally and present an algorithm for its construction.

Definition: A **Context-aware Mapping** *CM* is a table of key-value pairs $\{me; \langle e_i, \ldots, e_j \rangle\}$; where *me* is a model element and $\langle e_i, \ldots, e_j \rangle$ is an event sequence after which *me* was previously observed to be available for execution, where event $e_i \in I$, the initial event set for the GUI. The entry is *NONE* if me is executable and the first event in *me* is in I, *i.e.*, no sequence is required to reach *me*.

As alluded to previously, the context-aware mapping is constructed from event sequence execution. During the execution of each sequence, we maintain an explicit structure to compute the context-aware mapping. Figure 3 shows the structure for the example discussed above. At the very top is the executing event sequence $\langle e_2, e_3, e_4 \rangle$. The set of enabled events after each executed event is enclosed in a dotted oval. The shaded nodes are events in I. Solid arrows show the sequence executed; a dashed arrow from event e_x to e_y shows that e_y was available and enabled after the execution of e_x. To obtain the context-aware mapping, one needs only to trace each edge back to the starting event. For example, the edges (e_4, e_5), (e_4, e_6), and (e_4, e_7) have a path $\langle e_2, e_3 \rangle$ from the left-most node.

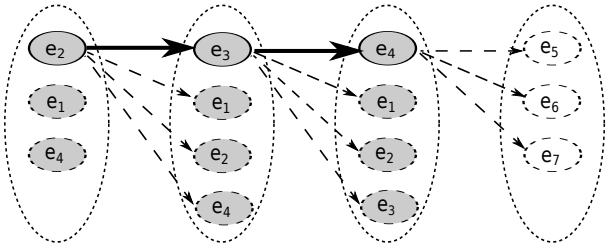


Fig. 3. Available Events Observed During Execution.

Algorithm 1 shows how this structure, T, is constructed and used to create/update the mapping, *CM*. The algorithm takes three inputs: (1) a sequence of executed events, each paired with a set of events available and enabled after its execution, (2) the context-aware mapping available thus far (from previously processed event sequences), and (3) the set of events enabled in the initial state I. Lines 1–6 create the structure T. Edges are added from each executed event e_i to all events e_j that are available and enabled after e_i. Lines 7–23 use the structure to create the mapping. First, all the model elements *ME* are obtained from T (Line 7); for our example, this is the set of edges. Then each element *me* is processed using one of two cases: (1) if the first event of *me* is enabled in the initial state, as is the case for edge (e_2, e_2), the mapping entry is set to *NONE* (Line 10), (2) otherwise a *searchPath()* function is used to find a sequence from the left-most element of T to me (Line 12). For example, for the edge (e_4, e_7), the path returned is $\langle e_2, e_3 \rangle$.

Because event sequences may be long, it is possible

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2300857, IEEE Transactions on Software Engineering

that the GUI is driven back to its initial state multiple times during execution. In such cases, the path may become unnecessarily long, which is why we use $truncate()$ to remove leading events (Line 14). Lines 15–22 update the mapping *CM*. If an entry for me does not already exist in *CM*, the key-value pair $\{em, contextSeq\}$ is simply added. Otherwise, if an entry exists, the lengths of the existing and new path are compared; the shorter is retained for efficiency reasons.

Algorithm 1 Construct Mapping

Input: $\langle(e_1, \alpha(S_1)) \ldots, (e_n, \alpha(S_n)\rangle$: executed sequence
Input: *CM*: Context-aware mapping
Input: $\alpha(I)$: Events enabled in initial state
1: $T = \emptyset$
2: **for** $i = 1 \rightarrow n$ **do**
3: **for all** $e_j \in \alpha(S_i)$ **do**
4: *T.addEdge*(e_i, e_j)
5: **end for**
6: **end for**
7: *ME* $\leftarrow$ *getModelElements*(T)
8: **for all** $me \in ME$ **do**
9: **if** *firstEvent*$(me) \in \alpha(I)$ **then**
10: *contextSeq* = *NONE*
11: **else**
12: *contextSeq* = *searchPath*(me, T)
13: **end if**
14: *truncate*(*contextSeq*)
15: **if** $me \notin$ *CM* **then**
16: *CM.addEntry*$(me,$ *contextSeq*$)$
17: **else**
18: $contextSeq_{old} \leftarrow lookUp(CM, me)$
19: **if** $|contextSeq_{old}| > |contextSeq|$ **then**
20: *CM.updateEntry*$(me,$ *contextSeq*$)$
21: **end if**
22: **end if**
23: **end for**
24: **return** *CM*: Updated context-aware mapping

3.2 Contribution 2: Simultaneously Extracting New Model Elements During Test Execution

We define a GUI test case as a pair $(S_0, \langle e_1; e_2; e_3; \ldots; e_n\rangle)$, where S_0 is a designated start state of the GUI for this test case; and each $e_i \in E$, the set of events in the GUI. Our test executor (or *Replayer*) starts executing the test case by launching the GUI under test in start state S_0, and executes each event one by one. It determines the correctness of the GUI by using a test oracle [27]. Consider the GUI of our running example shown in Figure 2(a). The start state is marked S_0. All test cases start in this state. During test execution, the GUI transitions through a sequence of states where each state is obtained after the execution of an event. In our work, we assume that the outcome of an event in a given state is deterministic. In our running example, once e_4 is executed, the GUI changes to state S_1 or S_2 based on the states of widgets corresponding to events e_1 and e_2.

We define a *GUI state* as the full set of all triples (w_i, p_j, v_k), where w_i is a widget currently extant in the GUI, p_j is a *property* of w_i, taken from a designated set of properties, and v_k is a *value* for p_j, taken from a set of possible values. We see some such triples in Figure 2(a) for our running example. The GUI states S_0, S_1, and S_2 would need to contain such triples for all widgets, all their properties, and values.

We augmented our test executor to collect the state of the GUI after the execution of each event. There are several ways to implement this functionality, including inserting hooks in the code to extract object state, invoking methods in the underlying code, and *reflection*[9]. In our implementation, we use reflection to obtain the object class for each widget as well as the set of methods associated with the class. If the method name starts with the *get*, (*e.g.*, *getLabel()*, *getX()*, *getY()*), we invoke it to *dynamically* obtain the value of the property. The part of the method name immediately following `get` becomes the name of the property. This approach is useful because it is impossible to predict the list of all properties of all possible widget types. For example, the `label` property is available for a *JButton* but not for a *JTextField*. Similarly, if the method name starts with the *is*, (*e.g.*, *isEnabled()*, *isVisible()*), we assume that it returns a boolean value that is also added to our properties. Figure 4 shows part of our Java code used to collect states for GUI widgets.

```
Method[] methods = widget.getClass().getMethods();
for (Method m : methods) {
  String methodName = m.getName();

  if (methodName.startsWith("get")) {
    property = methodName.substring(3);
    value = m.invoke(widget, new Object[0]);
  }

  if (methodName.startsWith("is")) {
    property = methodName.substring(2);
    value = m.invoke(widget, new Object[0]);
  }
  ...
}
```

Fig. 4. Code to collect GUI widget states.

We note that the above described process of collecting GUI widget states using *get* methods may not be entirely safe. It may accidentally invoke methods with side-effects and change the current object states. Although we did not encounter this situation in our work, there is no guarantee that we will not do so in the future. In the future, we will explore better ways of collecting states, e.g., by using a lightweight side-effect analysis [28] or by borrowing approaches

9. A process by which a computer program can access the object methods at runtime without knowing its implementation or type.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2300857, IEEE Transactions on Software Engineering

from specification mining [29] to first identify a set of side-effect-free methods, and use those methods as inspectors to obtain the current object states.

Once we have the sequence of states, one state after each event, we developed a post-processing step to pass it for addition to the EFG model, which we discuss in Section 3.4.

3.3 Contribution 3: Unique Widget Signatures

So far, we have conveniently referred to individual widgets by their *text labels*, *e.g.*, *Insert*. Although this is fine for informal discussion in this paper's text so long as the context is clear, use of a text label to identify a widget is insufficient for our tools such as the *Ripper* or *Replayer*. One cannot expect to perform an event on a widget, for example, using a method *invoke("Insert")*, and expect it to work correctly in all contexts; for instance, there might be two widgets, a button and a pull-down menu, in the current window with text label "*Insert*"; an automated tool does not know which one to execute. In such a situation, one might disambiguate by adding the "widget type" to the call, *e.g.*, *invoke("Insert", Button)*. But this too would not work if both widgets were buttons. One may specify additional widget attributes, *e.g.*, widget coordinates to the invocation to further disambiguate.

The above discussion is moot if each widget in the GUI had a unique identifier, perhaps assigned when programming the GUI, that remains unchanged across application runs. Such identifiers may be used by testers/tools to identify a widget, *e.g.*, during the *ripping* and test generation phases, and then again later during test execution. Several researchers and practitioners have advocated the need for such identifiers for good *testability* of GUI software [30], [31]. However, in practice, such identifiers are rarely used [32]. In all fairness, there are situations in which it becomes difficult to use identifiers for widgets. For example, widgets may be *dynamically generated* based on some underlying data, *e.g.*, one widget for each item available in an online store's database.

Whatever the reasons for not having widget identifiers in practice, the problem of not being able to uniquely identify widgets severely complicates our new work. Consider the *Insert* button in our running example. Our tools (*Ripper* and *Replayer*) may encounter it in two different contexts: first in the modal window entitled *Blog post* and second in the window entitled *Blank document*. These tools need to determine whether both these encounters were for the same widget or two different widgets; the determination will result in either one or two nodes in the EFG. Because we created this running example, we know that it is the same *Insert* in both instances, which is why we gave it the unique identifier e_5. In fact, we know that *Blog post* and *Blank document* are two instances of the same modal window. However, a tool has no way of automatically acquiring this knowledge (e.g., using object identifiers or source code locations) for a number of reasons. First, the OME* model is developed during multiple test runs – the execution environment will generate new object identifiers for GUI objects. Second, a GUI object may be created and destroyed multiple times during a test case's execution – a number of new object IDs may be given to the same GUI object. Finally, two GUI objects may be created by the same underlying source code.

Admittedly, it is impossible to devise a general unique widget identification scheme that works for all possible GUIs. Any solution will have to be application-specific. In this section, we describe a general mechanism that must be manually fine-tuned on a per-GUI basis. Our mechanism is based on using a combination of certain parts of the state of the widget and its container (*e.g.*, window). We cannot use the entire state for identification because it will contain some property values that change during the GUI's execution but do not play any role in identifying that widget. For example, the value of the *text* property for a *JTextField* object will change when the text changes; the *enabled* property changes when the object is enabled/disabled. Such properties cannot be used for our signature because any change to their values will indicate a new widget, which would be incorrect. We are, in some sense, defining *equivalent states* of widgets by using a subset of properties to uniquely identify widgets.

More formally, we define the *signature*, C_{sig}, for a container C as follows:

$$C_{state} \leftarrow \langle (p_1, v_1), (p_2, v_2), \ldots, (p_n, v_n) \rangle \quad (1)$$

$$\langle v_i, \ldots, v_k \rangle \leftarrow select(filter_p, C_{state}) \quad (2)$$

$$C_{sig} \leftarrow \Phi(\phi_i(v_i), \ldots, \phi_k(v_k)) \quad (3)$$

where the user defines, per GUI, $filter_p$, a specification of a subset of the container's properties and transformations $\phi_i \ldots \phi_k$ on the values of the properties. The function $select$ returns the values of the properties specified by $filter_p$ and function Φ is a hash function on the transformed values.

Along similar lines, we define the *signature*, w_{sig}, for a widget w in a container with signature C_{sig}, as follows:

$$w_{state} \leftarrow \langle (p_1, v_1), (p_2, v_2), \ldots, (p_n, v_n) \rangle \quad (4)$$

$$\langle v_i, \ldots, v_k \rangle \leftarrow select(filter_p, w_{state}) \quad (5)$$

$$w_{sig} \leftarrow \Gamma(C_{sig}, \gamma_i(v_i), \ldots, \gamma_k(v_k)) \quad (6)$$

where $filter_p$ and $\gamma_i \ldots \gamma_k$ are user-defined; and function Γ is a hash function on the transformed values and the container's signature.

In Section 4, we give examples of these user-defined functions and transformations, and empirically show, for our subject GUI applications, that they help to uniquely identify widgets.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2300857, IEEE Transactions on Software Engineering

3.4 Contribution 4: Incremental EFG+ Enhancements

Once a new widget/event is identified, it is used to enhance the EFG+ model. We have already discussed, in Section 3.1, how to incrementally update the context-aware mapping, which is an important part of the EFG+ model. We now discuss how to incrementally enhance the EFG.

We have already informally discussed EFG enhancement in Section 2 and illustrated it in Figures 2(d) and 2(e). These figures actually show the three important steps for incremental EFG enhancement: (1) add a node to represent the new event; (2) add edges *to* the new node and (3) add edges *from* the new node to other nodes.

To explain these steps, we revisit two important terms in GUIs: *modal* and *modeless* windows. At any time during GUI interaction, a user is allowed to execute events within a modal window and any modeless window that was opened from the modal window. At no time can the user jump between modal windows without explicitly terminating them. Moreover, the user cannot interleave events that belong to modeless windows associated with different modal windows. Again, the user must explicitly terminate the modal window that is associated with the modeless window, explicitly invoke the other modal window, open the modeless window, and invoke any of its constituent events. A part of MS Word's window hierarchy is shown in Figure 5. *Edit Picture* and *Edit Chart* are modal windows whereas *Format Picture*, *Help Picture*, *Manage Template*, and *Help Chart* are modeless. Consider events x, y, z, a, b, and c. A user may execute x, y, and z together because they are all contained in *Edit Picture*'s window group; similarly, events a, b, and c may be executed together. However, these two sets of events cannot interleave without their modal windows being explicitly invoked and terminated.

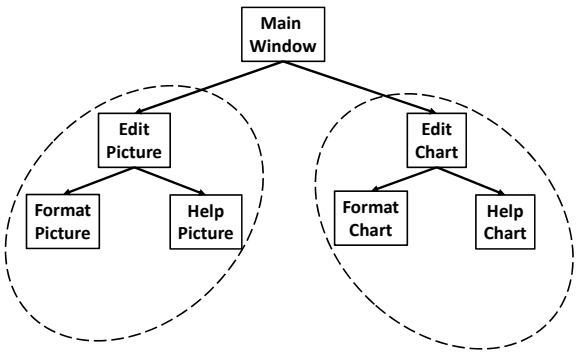


Fig. 5. A partial window hierarchy of MS Word.

The above behavior of GUI windows to restrict sets of events leads to the definition of a new term that we call the *scope* of an event. We define the *scope* of an event e as the set of events contained in the group of modal and modeless windows to which e belongs. We use scope in an algorithm to incrementally and efficiently enhance the EFG model.

More formally, we use Algorithm 2 to enhance our EFG. The algorithm is invoked after each event, e, is executed. It takes two parameters: (1) the EFG, and (2) the executed event. The set of all events available (enabled or disabled) is first obtained (Line 1). For each event, e_i, in this set, three steps are performed. First, if e_i has never been seen before (as was the case with e_8 in Figure 2(d)), then it is added to the set of nodes in the EFG (Line 4). Second, if the edge that was used to get to e_i was never seen before, then it is added as an edge (Line 7). This was the case for the edge (e_4, e_8) in Figure 2(d). Third, the set of events in e_i's scope are obtained (Line 9). Those that are not structural, *i.e.*, do not open/close modal windows, are used to add edges to the newly observed event e_i (Line 13). This is what we used for edge (e_5, e_8) in Figure 2(d).

Algorithm 2 Enhance EFG Model

Input: (N, E): EFG
Input: e: event executed
1: $AE \leftarrow$ *getAllEventsAfter*(e)
2: **for all** $e_i \in AE$ **do**
3: **if** $e_i \notin N$ **then**
4: N.*addNode*(e_i)
5: **end if**
6: **if** $(e, e_i) \notin E$ **then**
7: E.*addEdge*(e, e_i)
8: **end if**
9: $scope_i \leftarrow$ *getScope*(e_i)
10: **for all** $e_{ij} \in scope_i$ **do**
11: **if not** (structural(e_{ij})) **then**
12: **if** $(e_{ij}, e_i) \notin E$ **then**
13: E.*addEdge*(e_{ij}, e_i)
14: **end if**
15: **end if**
16: **end for**
17: **end for**
18: **return** (N, E): Updated EFG

The same algorithm is also used to add new edges *from* newly discovered events, as we saw in Figure 2(e) for e_8. However, this is done in a separate invocation of Algorithm 2, after the event is executed. Consider the invocation where the second parameter, e is the event e_8. The events available after e_8, Line 1, are $\{e_5, e_8\}$. Because there are no outgoing edges from e_8 in the EFG so far, Line 7 will add two new edges (e_8, e_5) and (e_8, e_8).

3.5 Contribution 5: Incremental Test-Case Generation

The new elements added to our model (*e.g.*, EFG) may create new test requirements. For example, if a new edge has been added to the EFG and our test

criterion is "cover all edges at least once," then we need to cover the new edge via a new test case. Hence, we need new ways to incrementally generate test cases to cover new model elements. Note that not all changes to the model will create a need for new test cases. For example, if the criterion is "cover all nodes," then newly added edges in the model may not require additional test cases. The need for additional test cases is dictated by the test criteria, not new model elements.

To incrementally generate test cases, we maintain a set of model elements that have already been covered. Another set of model elements (the complete set – covered and not covered) is obtained from the latest EFG. These two sets give us the set of model elements that still need to be covered. For each not-yet-covered model element, we generate a test case to attempt to cover it. We first try to get a path from the initial state to the element using the context-aware mapping; this test case is guaranteed to be executable. If there is no mapping entry, then a path is generated using the shortest-path algorithm.

4 Experiments

We now empirically determine whether the OME* paradigm improves the state-of-the-art, called the *baseline* (*BL*), in GUI testing. We will also compare the performance of OME* and BL against that of a random traversal (RND)—inspired by the RANDOOP [33] approach for test generation—of the GUI's input space. To this end, we will select several software subjects to test, and generate and execute test cases (for *BL* and OME*) that attempt to satisfy predetermined adequacy criteria. We will also execute the *RND* approach. We will then compare the outcomes of the test runs.

4.1 OME* Implementation

We implemented the OME* testing paradigm in an experimentation tool. The tool is written in Java, consisting of two main components: The *platform-specific component* consists of modules to interact with a specific GUI platform. These modules automatically send events to the GUI (*e.g.*, clicks, type-in-text), collect GUI states and store them in XML files. The *platform-independent component* implements the algorithms presented in Section 3, which, among other things, generate abstract test cases to supply to the platform specific component.

We highlight two aspects of our implementation. First, our tool implements a fully-automatic workflow for test case execution and analysis. An important part of this implementation is a distribution and consolidation algorithm to execute our test cases in parallel over a number of *slave* machines. The test suites are split into batches by a controller machine and sent to multiple remote slaves in a computing cluster. Upon completion of test case execution, the results are sent back to the controller machine where they are consolidated and analyzed (*e.g.*, the model is updated and new test cases are generated).

Second, our RND implementation works as follows. It starts in the *Main* window and obtains the state of all widgets, their properties, and values. It then gets the set of all enabled and visible widgets on which events can be performed. It randomly selects one event, executes it, and obtains the resulting state. If this state is different from all previously encountered states, it is recorded. This process is repeated until a failure is encountered. In our experiments, each event executed always led to a new state; we attribute this to the richness of GUI states.

4.2 Research Questions and Metrics

More specifically, we are interested in answering the following two research questions:

RQ1: How effective is OME* when compared with *BL* and *RND*? We will measure the *fault detection effectiveness* (FDE), *event coverage* (EC) [34], and *code coverage* (CC) of these three approaches.

RQ2: By how much does the context-aware mapping improve the OME* approach? We will implement OME* in two ways—one with the context-aware mapping and the other without—and compare their FDE, EC, and CC.

Metrics: For FDE, we count the number of faults. Our test oracle was based on the software crashing (terminating abnormally or throwing an uncaught exception). For EC, we measure EFG *node coverage* (E1) and EFG *edge coverage* (E2). For CC, we measure *statement (stmt.)*, *branch*, *method*, and *class* coverage.

4.3 Selecting & Setting Up Software Subjects

We select eight subject applications from two popular open-source communities Tigris.org[10] and SourceForge[11].

1) **ArgoUML:** A CASE tool for UML diagram design, code generation and reverse engineering;
2) **Buddi:** A financial tool for personal budget management;
3) **CrosswordSage:** A tool for creating and solving crosswords;
4) **DrJava:** An advanced integrated development environment (IDE) for Java programs;
5) **JabRef:** A database management tool for bibliographies management;
6) **OmegaT:** A language tool for automated translation;
7) **PdfSam:** An office utility for advanced pdf files manipulation;

10. http://www.tigris.org
11. http://sourceforge.net

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2300857, IEEE Transactions on Software Engineering

8) **Rachota:** A time management tool for project time tracking;

They are all implemented in Java and rely on the GUI for user input. Table 2 summarizes their characteristics. The applications span a variety of domains, ranging from games to office utilities and software development tools. We selected the most recent released versions at the time the study was conducted. All of them are widely used, demonstrated by the high numbers of downloads, and have broad user communities, demonstrated by the multiple numbers of languages available. They are all mature applications, in that they have been around for at least 5 years. They also have non-trivial code sizes in terms of the numbers of non-comment statements (S), branches (B), methods (M), and classes (C). Over the years, a large number of bugs have been reported by their respective communities and fixed by the developers in response.

Having identified the study subjects, we now prepare our tools to use them.

4.3.1 Defining Functions for Unique Signatures

Our first preparation step is to ensure that we correctly identify each window and widget in the applications. As described in Section 3.3, we develop functions for windows (our containers) and widgets for this purpose. We start with windows, for which we need to develop $filter_p$, to select a subset of window properties, and $\phi()$ and $\Phi()$ to generate a unique window signature. It turns out that our study subjects only require the use of one window property, namely "*window title*." The value of this property is the title of the window, which, for the most part, are already unique. The exceptional cases are handled by mapping a few titles to regular expressions. For example, the title of the window "*Save file*" in ArgoUML can change dynamically during its execution; it always starts with the string *Save* followed by the full path to the file's location. The title changes when the user saves the file in another location. Hence, if we rely solely on window title, there is a danger that we consider each instance with a different title as a new window. To ensure that our tools recognize that all instances of this *Save file* window, with different titles, are in fact the same window, we map the title string, via the $\phi()$ function, to the regular expression '*Save (/.*)**'. We did this for a few windows. This process took less than 5 minutes per application; the vast majority of this time was spent recognizing that titles of certain windows change.

The case for widgets is more complex. The *title* of widgets in our study subjects is repeated many times in the application. For example, many buttons share the title `OK`. For this reason, we use three properties to identify widgets, namely *title, icon*, and *class*, representing the main title/label of the widget, the file-name of the icon labeling the widget if it exists, and the object class used to implement the widget, respectively. The values of these three properties are used in the Java *HashCodeBuilder* utility to generate a hash code for each widget. The manual effort required for this step was less than a few minutes.

We verified that our signatures are indeed unique. We manually examined each application and counted all its widgets and windows; we show the numbers in Table 3 under "*Manual Oracle*". We then used our tools to do the same. Our tools first extracted the *Unique Title Strings* and used our $\phi()$ function to map some of them to regular expressions. Similarly, our tools extracted the *Text Titles* for widgets, determined the image used for the *Icons*, and the Java *Classes* used to implement the widgets. These were then mapped to *Widget Hash codes*. Combined with their associated window hashes, we obtained unique *Mapped Widget Signatures*. As Table 3 shows, the resulting numbers matched our manual oracle exactly.

4.3.2 Sandbox and Text Parameters

Our second preparation step for testing the applications included setting up a *sandbox*. The key role of this sandbox was to ensure that each run of the application was independent of all prior runs. We defined a default configuration for each application, e.g., when possible, we bypassed the splash screen, or we specified a delay time to allow the splash screen to disappear. Before a test case is executed, the application is reset to its default configuration.

Finally, our third preparation step involved setting up data for parameterized events. A general parameter populating strategy was used. For events requiring a text input (*e.g.*, text field, text area), a database that contains one instance for each of the text types in the set {*negative number, real number, zero, long random string, empty string, string with special characters*} was used. All instances in the text type set were tried in succession for each test case.

4.4 Running the Experiment

Having prepared the study subjects, we are now ready to run the experiment, first establishing the baseline (*BL*) followed by the 2 instances of OME* (with and without the context-aware mapping), and finally the random (*RND*) traversal. For each, we record coverage (*E1*, *E2*, *Stmt.*, *Branch*, *Method*, *Class*), number of *Nodes* and *Edges* in the EFGs, and number of test cases that were *Generated*, and—for *BL* and OME*—those that executed to completion (*Feasible*), and number of faults found. The code coverage is obtained by a code coverage measurement tool called Cobertura[12]. Other data is obtained by our tool as discussed in Section 4.1.

More specifically, for each study subject, we perform the following steps:

12. http://cobertura.sourceforge.net

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2300857, IEEE Transactions on Software Engineering

TABLE 2
Subject applications

Name	Abbreviation	Version	Downloads	Languages Translated	Year	Bug Reports		Size**			
						Fixed	Total	S	B	M	C
ArgoUML	AU	0.33.1	N/A*	11	1999	N/A*	N/A*	69,954	32,084	16,091	1,891
Buddi	BD	3.4.0.8	897,520	13	2006	279	304	9,588	3,711	2,318	384
CrosswordSage	CS	0.3.5	4,623	1	2005	1	8	1,826	456	336	34
DrJava	DJ	r5004	1,227,393	1	2002	966	1091	64,994	17,485	15,229	2,394
JabRef	JR	2.7b	1,173,313	4	2003	564	768	44,522	18,176	7,502	1,267
OmegaT	OT	2.1.3	254,559	29	2002	462	503	19,756	6,772	4,519	714
PDFSam	PS	2.2.1	2,548,362	21	2006	71	87	6,097	2,043	1,504	194
Rachota	RC	2.3	74,107	11	2003	124	174	11,183	2,837	1,898	320

* The all-time statistics for ArgoUML are not publicly available. However, its popularity and maturity are partially demonstrated by the current more than 19,000 registered users and over 150 active developers (http://www.isr.uci.edu/tech-transition.html).
** S = Statements; B = Branches; M = Methods; C = Classes

TABLE 3
Automated Widget/Window Identification vs. Manual Oracle.

(a) Window Identification.

SUT	Manual Oracle	Automated	
	Numbers of Windows	Unique Title Strings	Mapped Title Strings
AU	28	38	28
BD	20	24	20
CS	9	9	9
DJ	38	44	38
JR	52	56	52
OT	30	31	30
PS	6	6	6
RC	12	18	12

(b) Widget Identification.

SUT	Manual Oracle	Automated				
	Numbers of Widgets	Text Titles	Icons	Classes	Widget Hash-codes	Mapped Widget Signatures
AU	1040	710	20	113	1023	1040
BD	728	410	0	119	697	728
CS	137	111	0	40	130	137
DJ	1144	742	16	114	1132	1144
JR	1600	1127	30	134	1511	1600
OT	824	539	6	96	801	824
PS	323	272	18	72	315	323
RC	462	361	6	65	453	462

1) Create EFG using the Ripper.
2) Generate tests from EFG, execute them, and record all metrics. This forms our *BL*.
3) As discussed in Section 3.2, new model elements are extracted during Step 2 above.
4) As per Section 3.4, enhance the EFG model.
5) Generate test suite from new EFG and execute them. Record all metrics for this suite. This forms our first implementation of OME* that does not use the context-aware mapping. We call this *noMap*.
6) As per Section 3.1, the context-aware mapping is created. Together with the EFG from Step 4, this forms the EFG+ model.
7) Generate test suite from new EFG+ and execute them. This forms one *iteration* of our *withMap* approach. Record all metrics for this suite and extract new model elements.
8) Enhance both EFG and mapping. Repeat Step 7 until the EFG+ model does not change.

The *RND* process was run completely independently of the above process. The total number of test cases and their execution times (in hours) are shown in Table 4. Note that we executed over 500,000 test cases in almost 1100 machine days. We used 120 2.8 Ghz P4 Linux nodes running in parallel.

4.5 Threats to Validity

As is the case with all empirical studies, our experiments suffer from several threats to validity. *Threats to external validity* are factors that may impact our ability to generalize our results to other situations. We have used eight open-source Java applications. Although carefully selected, they do not reflect the spectrum of all possible GUIs that are available today. Moreover, the applications are extremely GUI-intensive, *i.e.*, most of the code is written for the GUI. We expect that results may be different for applications that (1) have complex underlying business logic and a fairly simple GUI, (2) are developed using other programming paradigms, and (3) are tested in-house for commercial applications. Finally, we initialized various values for text-fields manually and stored them in a database. We may see different results for different values.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2300857, IEEE Transactions on Software Engineering

TABLE 4
Test Cases Generated and Execution Time

SUT	BL		withMap		noMap		RND	
	Test cases	Exec time (h)	Test cases	Exec time (h)	Test cases	Exec time (h)	Test cases	Exec time (h)
AU	4,468	192.9	22,086	1,726.7	9,782	248.7	3,191	57.6
BD	2,890	32.5	32,712	399.7	8,204	63.0	6,274	125.8
CS	266	1.1	333	1.3	333	1.9	29	0.4
DJ	5,842	206.4	34,702	1,514.1	18,141	48.5	4,250	126.4
JR	39,555	2,372.7	170,809	7,639.2	54,516	3,028.8	22,888	588.8
OT	3,605	60.9	18,275	282.4	9,581	110.6	2,205	58.0
PS	6,856	48.9	9,135	85.9	9,135	71.9	444	5.0
RC	1,456	3.5	8,504	122.3	4,784	40.6	966	57.5
Total	**64,938**	**2,918.8**	**296,556**	**11,771.6**	**114,476**	**3,614.1**	**40,247**	**1,019.5**

Threats to internal validity are possible alternative causes for experimental results. Because we wanted to achieve full automation, we developed functions to identify widgets/windows uniquely. The instruments used for run-time state collection of GUI widgets were based on Java Swing API. These widgets may have additional properties that are not exposed by the API. Hence the states captured may be incomplete, causing us to map different windows/widgets into the same unique element. Moreover, because GUI execution requires frequent painting/repainting of windows, the captured state will be inaccurate if captured too early in the painting process; we set long artificial delays to allow the GUI to finish repainting. For RND, we tried to reduce the effects of randomness by running the algorithm 5 times and averaging the results.

Threats to construct validity are discrepancies between the concepts intended to be measured and the actual measures used. We used the number of crashes as our fault detection effectiveness metric; event and code as coverage metrics; these might not be useful metrics in all situations.

4.6 Results

We summarize our results in terms of the metrics that we collected for all 4 suites, *i.e.*, *BL*, *noMap*, *withMap*, and *RND*, in Table 5. We also break up the results of *withMap* by iteration so as to see the effect of OME*. From this raw data, we want to bring several points to the reader's attention.

Number of Iterations: Technique *noMap* has a single iteration as opposed to several for *withMap*. This is because even though new model elements were discovered during test execution in *BL* and used in Iteration 1 of *noMap*, very few of them were in fact reachable because of the absence of the mapping. This led to a large number of "infeasible" test cases that did not execute to completion. We revisit this point in more detail later. Note that there are no infeasible sequences for RND because of the nature of running the test cases directly on the GUI; if a widget is enabled, only then it is executed.

EFG: The EFG+ model improves—gets bigger in size—with each iteration of *withMap*. In most cases, the number of EFG edges is significantly larger (*e.g.*, 1044% for Buddi) compared to *BL*, showing that we were able to observe, model, and exercise a larger number of GUI events, hence test more functionality. The number of entries in the context-aware mapping also grows steadily with each iteration, indicating that we are able to reach and execute new coverage elements. This is all directly reflected in improved fault-detection effectiveness and increased event and code coverage.

We pictorially examine and explain the growth in the EFG model via an example. Figure 6 shows a bird's eye view of the EFGs of our subject application *Buddi* for *BL* and 5 iterations of *withMap*[13]. Our goal is not to show details of the EFGs; rather, we want to show very high level pictures of the EFGs so that the reader can visually appreciate the changes from one EFG to the next. The EFGs have been drawn in such a way that the (x, y) location of each node in the EFG is fixed across iterations. For example, the *OK* event labeled in Figure 6(a) is in the same location, relative to all other nodes in Figures 6(b) through 6(f). We add labels to highlight specific parts that we discuss in the text.

At a high level, there are stark differences between the EFGs of Figure 6(a) (*BL* technique) and Figure 6(f) (final iteration of *withMap*). For example, the *Language Items*, *New Account*, *Edit Account Types* clusters and a large number of edges do not even appear in Figure 6(a); all these are observed only during the OME* process. Hence, *BL* has no way to cover these events/edges.

Buddi has a context sensitive GUI, which changes the set of available events based on the end-user's

13. Additional visualizations of the EFGs, detailed code coverage reports, and actual fault reports are available at https://www.cs.umd.edu/users/atif/tse2012

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2300857, IEEE Transactions on Software Engineering

TABLE 5
Data for RQ1 and RQ2.

ArgoUML		BL	withMap					noMap	RND
			1	2	3	4	5		
EFG	# Nodes	328	372	385	418	473	-	372	398.4
	# Edges	4,468	9,062	11,731	17,147	20,485	-	9,062	5,069.4
Mapping	# Entries		8,485	11,014	15,600	18,481	-	-	-
Test Cases	# Gen.	4,468	5,314	3,911	6,621	4,772	-	5,314	3,191
	# Feas.	3,763	4,771	2,743	5,376	3,223	-	556	-
Event Cov.	% E1	66.81	84.78	86.47	88.16	89.01	-	69.40	83.59
	% E2	18.37	41.66	55.05	81.29	97.03	-	19.21	36.64
Code Cov.	% Stmt	22.45	24.87	24.89	24.91	24.91	-	24.15	27.71
	% Brnch	10.31	11.83	11.86	11.88	11.88	-	11.29	14.75
	% Mthd	26.21	28.22	28.23	28.23	28.23	-	27.59	30.48
	% Class	52.09	54.63	54.63	54.63	54.63	-	53.83	56.51
# New Faults		-	2	2	0	0	-	0	1.2
# Total Faults		4	6	8	8	8	-	4	3.4

Buddi		BL	withMap					noMap	RND
			1	2	3	4	5		
EFG	# Nodes	249	297	344	412	457	500	297	221.8
	# Edges	2,890	7,129	11,011	17,019	24,396	30,196	7,129	4,003.2
Mapping	# Entries		5,486	8,130	13,993	20,661	25,369	-	-
Test Cases	# Gen.	2,890	4,289	4,271	6,757	7,943	6,562	4,289	6,274.4
	# Feas.	2,324	3,721	3,434	5,966	7,094	5,577	2,471	-
Event Cov.	% E1	40.00	48.20	56.60	65.40	68.60	72.00	31.01	47.92
	% E2	7.70	20.02	31.39	51.15	74.64	93.11	15.88	10.67
Code Cov.	% Stmt	38.54	47.61	48.90	49.04	49.19	49.34	38.67	44.01
	% Brnch	17.06	21.77	22.42	22.50	22.90	22.96	17.08	18.41
	% Mthd	36.45	42.23	42.92	43.18	43.27	43.49	36.71	38.87
	% Class	64.06	75.00	75.78	75.78	76.30	76.30	64.32	74.22
# New Faults		-	4	3	0	0	0	0	0.4
# Total Faults		1	4	7	7	7	7	1	0.4

CrosswordSage		BL	withMap					noMap	RND
			1	2	3	4	5		
EFG	# Nodes	40	40	-	-	-	-	40	31.6
	# Edges	266	330	-	-	-	-	330	174.6
Mapping	# Entries		283	-	-	-	-	-	-
Test Cases	# Gen.	266	67	-	-	-	-	67	29
	# Feas.	230	64	-	-	-	-	7	-
Event Cov.	% E1	97.50	97.50	-	-	-	-	97.50	65.50
	% E2	69.70	89.09	-	-	-	-	71.82	35.03
Code Cov.	% Stmt	25.19	26.62	-	-	-	-	25.19	26.28
	% Brnch	8.55	8.77	-	-	-	-	8.55	7.19
	% Mthd	25.60	28.27	-	-	-	-	25.60	27.68
	% Class	41.18	41.18	-	-	-	-	41.18	44.12
# New Faults		-	3	-	-	-	-	0	0.8
# Total Faults		5	8	-	-	-	-	5	4.6

DrJava		BL	withMap					noMap	RND
			1	2	3	4	5		
EFG	# Nodes	275	399	462	523	-	-	399	380.0
	# Edges	5,842	12,299	19,876	30,613	-	-	12,299	20,322.8
Mapping	# Entries		9,614	17,260	28,272	-	-	-	-
Test Cases	# Gen.	5,842	9,120	8,189	11,551	-	-	9,120	4,250
	# Feas.	4,237	6,108	7,256	11,359	-	-	2,094	-
Event Cov.	% E1	46.85	69.41	78.78	82.22	-	-	33.25	59.43
	% E2	13.84	33.79	57.50	94.60	-	-	20.68	18.33
Code Cov.	% Stmt	26.09	27.70	28.95	29.63	-	-	26.30	26.44
	% Brnch	17.74	19.90	21.11	22.08	-	-	18.17	10.30
	% Mthd	28.28	29.58	30.61	31.26	-	-	28.43	27.31
	% Class	51.92	52.80	54.51	55.18	-	-	51.96	49.49
# New Faults		-	1	0	0	-	-	0	0.0
# Total Faults		3	4	4	4	-	-	3	2.2

JabRef		BL	withMap					noMap	RND
			1	2	3	4	5		
EFG	# Nodes	483	603	830	1,058	1,177	-	603	525.0
	# Edges	39,555	54,272	68,063	123,757	168,658	-	54,272	55,066.0
Mapping	# Entries		45,622	62,960	109,672	145,840	-	-	-
Test Cases	# Gen.	39,555	14,961	14,686	56,199	45,408	-	14,961	22,888
	# Feas.	30,850	21,164	14,302	53,607	44,601	-	12,248	0
Event Cov.	% E1	39.25	50.89	70.18	88.62	98.81	-	28.00	23.87
	% E2	18.29	30.84	39.32	71.10	97.55	-	25.55	11.77
Code Cov.	% Stmt	29.12	33.70	37.16	38.36	38.63	-	29.15	26.41
	% Brnch	12.04	15.12	17.53	18.84	19.16	-	12.17	10.27
	% Mthd	29.95	34.95	38.10	39.24	39.43	-	29.95	27.29
	% Class	51.62	57.62	62.83	63.38	63.54	-	51.62	49.46
# New Faults		-	6	3	0	0	-	1	1.0
# Total Faults		4	10	13	13	13	-	5	3.6

OmegaT		BL	withMap					noMap	RND
			1	2	3	4	5		
EFG	# Nodes	309	333	337	347	-	-	333	301.0
	# Edges	3,605	7,705	10,988	15,961	-	-	7,705	4,471.8
Mapping	# Entries		4,716	8,065	13,652	-	-	-	-
Test Cases	# Gen.	3,605	5,976	3,517	5,177	-	-	5,976	2,205
	# Feas.	2,712	4,308	3,356	5,001	-	-	2,730	0
Event Cov.	% E1	81.56	93.08	94.81	96.54	-	-	86.05	44.73
	% E2	16.99	43.98	65.01	96.34	-	-	49.53	12.57
Code Cov.	% Stmt	40.97	45.96	46.44	48.31	-	-	41.45	42.67
	% Brnch	23.36	29.43	29.71	32.21	-	-	24.08	27.16
	% Mthd	38.22	41.93	42.38	43.46	-	-	38.44	38.62
	% Class	62.61	65.97	66.67	67.23	-	-	62.75	63.89
# New Faults		-	1	0	0	-	-	0	0.0
# Total Faults		3	4	4	4	-	-	3	1.0

PdfSam		BL	withMap					noMap	RND
			1	2	3	4	5		
EFG	# Nodes	111	118	-	-	-	-	118	105.2
	# Edges	6,856	8,273	-	-	-	-	8,273	6,842.6
Mapping	# Entries		7,557	-	-	-	-	-	-
Test Cases	# Gen.	6,856	2,279	-	-	-	-	2,279	444
	# Feas.	6,086	1,675	-	-	-	-	445	-
Event Cov.	% E1	94.07	100.00	-	-	-	-	100.00	49.32
	% E2	73.56	93.81	-	-	-	-	91.03	7.95
Code Cov.	% Stmt	41.74	42.43	-	-	-	-	41.74	37.52
	% Brnch	15.91	16.84	-	-	-	-	15.91	13.25
	% Mthd	36.97	37.50	-	-	-	-	36.97	32.19
	% Class	68.04	68.56	-	-	-	-	68.04	65.98
# New Faults		-	5	-	-	-	-	0	0.4
# Total Faults		2	7	-	-	-	-	2	1.2

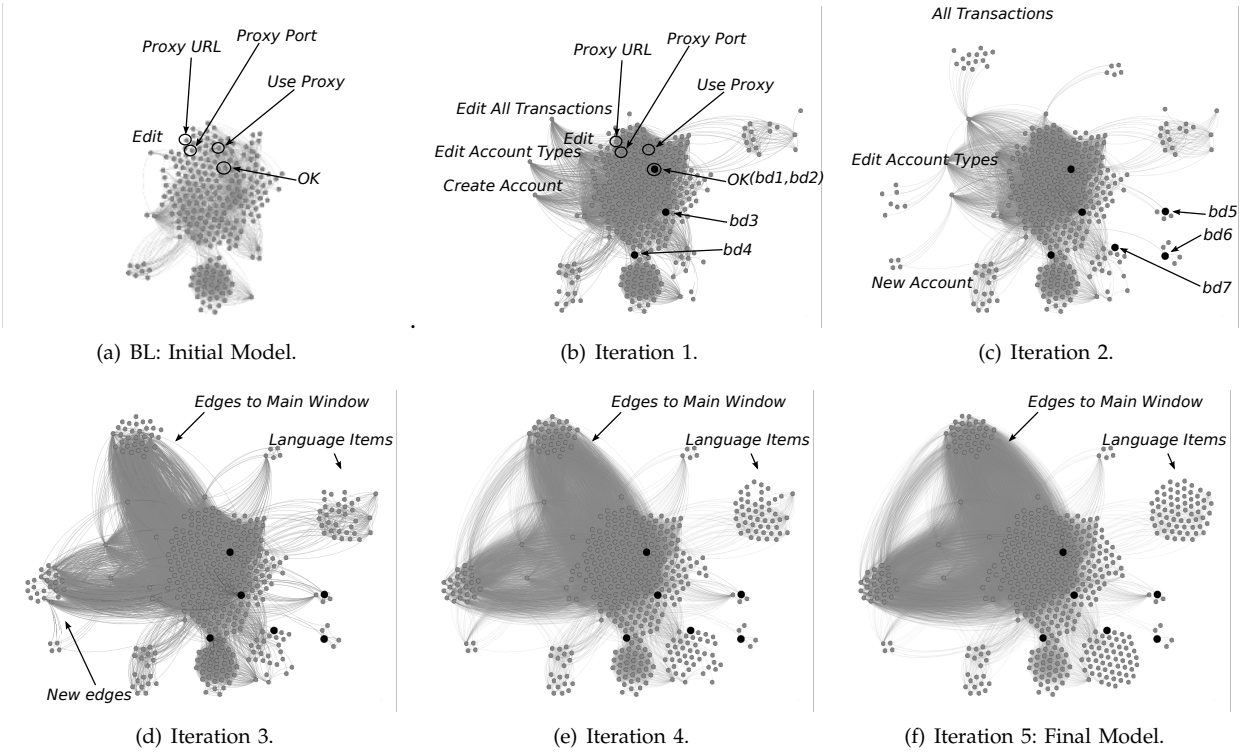
Rachota		BL	withMap					noMap	RND
			1	2	3	4	5		
EFG	# Nodes	151	167	171	185	185	-	167	135.4
	# Edges	1,456	4,726	6,136	6,849	8,324	-	4,726	1,277.0
Mapping	# Entries		3,444	5,684	6,188	7,490	-	-	-
Test Cases	# Gen.	1,456	3,328	1,474	739	1,507	-	3,328	966
	# Feas.	1,107	3,288	1,451	718	1,476	-	1,237	-
Event Cov.	% E1	73.51	84.32	87.03	96.22	96.22	-	83.24	54.59
	% E2	13.30	52.80	70.23	78.86	96.59	-	38.10	18.78
Code Cov.	% Stmt	61.19	64.37	65.62	66.37	66.40	-	61.19	53.55
	% Brnch	33.45	37.12	38.14	38.74	38.88	-	33.45	25.45
	% Mthd	46.21	47.89	48.89	50.53	50.53	-	46.21	37.70
	% Class	81.88	85.63	86.25	86.25	86.25	-	81.88	77.88
# New Faults		-	1	2	1	0	-	0	0.0
# Total Faults		2	3	5	6	6	-	2	0.2

current "working perspective," *i.e.*, at any time during its execution, events related to specific perspective are displayed. For example, during the execution of the *Ripper*, the *Edit* menu is only exercised only once in the *Report creation* perspective. Hence, only the *report* related sub-menu items are observed by the *Ripper*, and hence the *BL* technique.

In contrast, *withMap* is able to exercise *Edit* several times in many other perspectives. As a result, new sub-menu items are observed. As marked in Figure 6(b), for example, three new events *Edit All Transaction*, *Edit account types*, and *Create Account* are added to the original EFG when *Edit* is executed in the *Account Management* perspective. These events, when performed in subsequent iterations, will in turn, open new windows to extend the EFG even further (marked by the ovals in Figure 6(c)).

Our example illustrations also show that changes

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2300857, IEEE Transactions on Software Engineering



(a) BL: Initial Model. (b) Iteration 1. (c) Iteration 2.

(d) Iteration 3. (e) Iteration 4. (f) Iteration 5: Final Model.

Fig. 6. OME* sees more of the EFG with each iteration.

to the EFGs across iterations may not necessarily be changes in events, i.e., new edges may be added between previously observed events. For example, events in both *Edit Account Types* and *New Account* windows have been observed during Iteration 2 (via menu items in the main window). However, the edges linking these two windows are only revealed during Iteration 3, when the *New Account* event in *Edit Account Types* is exercised (shown using a "New edges" label at the bottom-left of Figure 6(d)). These new edges provide a new way to exercise events in the *New Account* window.

We also note that even a very small number of newly observed events may lead to a significant change in model size. Because *Buddi* allows users to work in multiple perspectives simultaneously (*e.g.*, adding a new account when generating a report), the windows are mostly implemented as modeless. For that reason, as soon as a new event is observed and added to the model, it gets connected to all previously known events, making the graph very dense. As shown in Figure 6(d) through Figure 6(f), the edge cluster from the *All Transactions* window's events back to all the events in the main window (the center of the EFG) grows significantly across iterations.

Mapping and Test Case Feasibility: Our context-aware mapping size also grows across iterations of *withMap*. This mapping plays a big role in ensuring that a larger number (compared to *noMap*) of test cases remain feasible. The 4 most important data points to illustrate this are the # *Generated* and # *Feasible* entries under Iteration 1 of *withMap*, and under *noMap*. For example, in *ArgoUML*, only 556 of 5314 test cases were feasible, *i.e.*, executed to completion, with *noMap*. In contrast, 4771 of 5314 test cases were feasible with *withMap*. This shows that by retaining when events were observed to be executable and using this information when exercising these events again proved to be very successful at making test cases executable.

The *JabRef* data of # *Generated* and # *Feasible* entries under Iteration 1 of *withMap* highlights another important aspect of our mapping. Even though we generated 14,961 new test cases during this iteration, a total of 21,164 test cases were feasible and executed. This is because some of the test cases from the previous run (in this case from *BL*) that remained unexecutable earlier, were now executable due to new entries in the mapping.

Event and Code Coverage: We see that we gradually increase the amount of code and events that we cover across iterations. However, we never achieve 100% coverage of our criterion, *i.e.*, cover all EFG edges. This means that we observe edges during some iterations but never get to reach them in subsequent iterations. This is due to the highly context-sensitive nature of our GUIs, where a context-aware mapping entry is no longer valid for a subsequent iteration. Addressing these cases is a subject for future work.

Missing event coverage causes us to lose code coverage (our code coverage varies from 24.91% to

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2300857, IEEE Transactions on Software Engineering

66.40%). In addition, our GUI subjects have numerous callbacks that never get invoked, either because they are dead code or get invoked in specific environments.

Faults: Table 6 shows the list of faults detected by OME*. Many of these faults were reported and fixed. Because of lack of space, we will discuss the 7 new faults that were detected in *Buddi*. All 7 were detected by the *withMap* technique. Of these 7, one fault was also detected by *noMap*. These faults are also indicated in Figure 6. The faulty events (*i.e.*, the last events in the failed test cases) are solid dark nodes with their IDs ($bd1$–$bd7$) pointed by an arrow. We now discuss these faults and the test cases that detected them.

Fault *bd1* results in an *IllegalArgumentException* when setting an out-of-range proxy port for network configuration. It is detected by a test case consisting of 6 events: $\langle e_1$: *Expand Edit menu*; e_2: *Open Preferences window*; e_3: *Select Network tab*; e_4: *Enable Use Proxy option*; e_5: *Enter a large number for proxy port*; e_6: *Click OK*$\rangle$. This fault did not occur earlier in the *BL* iteration because, by default, the *Proxy Port* text box (*i.e.*, e_5) is disabled. It is unable to change the proxy port unless the *Enable Use Proxy* check box is checked (*i.e.*, performing e_4). However, this information is not available at *BL*. During the *BL* process, event e_5 was observed after the execution of event e_4. Hence, a new context aware mapping entry was created to reach e_5. In the next iteration, test cases were generated to cover e_5. One of them led to the failure. Fault $bd2$ is similar to $bd1$ except that it throws an *UnknownHostException* when using a non-existing proxy URL (with a valid port).

This is a particularly interesting test case because event e_6, *i.e.*, *Click OK*, the one that revealed the failure had been executed several times in *BL*. However, the failure was manifested only when e_6 executed in the context of e_5. Moreover, e_6 revealed a different failure, $bd2$, when executed after setting a non-existing proxy URL. All events in the fault-revealing test cases were available in the initial model (as marked in Figure 6(a)). However, the faults were revealed only when the events were tested in specific combinations.

The remaining faults in *Buddi* were detected due to the *discovery of new events*. For example, fault *bd3* causes an *InvalidValueException* when saving a transaction with an empty name. In *Buddi*, a transaction can be saved only after a document change. However, during ripping, the events related to the save functionality (*e.g.*, *Save*, *Save All* buttons) were all executed before any document changing event. Hence, the *Save Transaction* dialog, which is opened by these events, did not show up. In subsequent iterations, however, it was opened and tested in new executing contexts. As a result, the fault was detected.

Faults *bd5*, *bd6*, *bd7* were detected during Iteration 2 when new events were observed and exercised. Detecting faults in later iterations is not uncommon, as can be seen in Table 5. The reason for this is that exercising fault-revealing events requires going through multiple other events, performed in a sequence. By iteratively identifying the enabling/opening relationships between events, the OME* test case generator is able to compose test cases that reveal faults. Consider, for instance, Fault $rc7$ in *Rachota*; this was detected in Iteration 3. It is an uncaught *NullPointerException*, thrown when simultaneously adjusting the current date, and setting the starting time of an active task to empty (recall that *Rachota* is an application for tracking project time). The event sequence leading to this fault is: $\langle e_{11}$: *Select a task*; e_{12}: *Start selected task*; e_{13}: *Expand Tool menu*; e_{14}: *Open Adjust start time window*; e_{15}: *Change to a previous date*; e_{16}: *Click OK to confirm the start time adjustment*$\rangle$. This sequence brings the GUI through a series of states where events are reached in a chain: first, e_{11} enables e_{12} in *BL*, and then e_{12} in turn enables e_{14} in Iteration 1. When performed in Iteration 2, e_{14} opens a new *Adjust starting time* window (e_{13} is known earlier to expand the *Tool* menu to reach e_{14}). This window contains the events allowing the user to adjust the starting time of the active task (*e.g.* e_{16}). Finally, the pair (e_{15}, e_{16}) is exercised in Iteration 3, leading to the exception.

We conclude our discussion of these faults by noting that due to the complexity of the GUI, the *state-based relationships between events* that led to failures are difficult to predict manually. Similarly, because GUI event handlers are often spread across multiple independent modules/classes [35], such faults cannot be detected by code analysis techniques such as static analysis.

Comparison with *RND*: The data for *RND* is the average over 5 runs. In all cases, we saw *RND* perform worse than OME*. We attribute this weakness of *RND* to its non-systematic GUI exploration strategy. In contrast, because OME* used the underlying criterion of covering all EFG edges, its test-case generator was forced to generate test cases that covered all parts of the GUI, including context-sensitive parts. *RND* did not have this advantage of an underlying event-based criterion.

Going back to our research questions, we were able to show that OME* is more effective than *BL*, when using FDE, EC, and CC as our metrics, thereby addressing **RQ1**. Further, we implemented OME* in two ways, with and without the mapping, and showed that because *noMap* does not maintain the context-aware mapping to reach coverage elements, it yields many infeasible test cases. The mapping is key to the success of OME*. This result answers **RQ2**.

5 Related Work

This work belongs to the family of *model-based* techniques for testing GUI-based applications. A model is developed to represent the GUIs behaviors and the test cases are generated by traversing the model.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2014.2300857, IEEE Transactions on Software Engineering

TABLE 6
List of new faults detected by OME*

ID	Iteration	Confirmed	Fixed	Description
AU4	1	✓	✓	ProfileException when using '/crash/crash' as name to save a user profile
AU5	2	✓	✓	NullPointerException when changing layout of an empty Activity diagram
AU6	1	✓	✓	NullPointerException when trying to 'Revert to Saved' an unsaved document
AU7	2	✓	✓	InvalidObjectException when keeping the 'Open Project' window open and saving another project
BD1	1	✓	✓	IllegalArgumentException with an out-of-range Proxy Port
BD2	1	✓	✓	IllegalArgumentException when updating with an non-existing proxy address
BD3	1	✓	✓	InvalidValueException when creating a transaction with an empty name
BD4	1	✓	✓	FileNotFoundException when saving with a non-encrypted file with name containing special characters
BD5	2	✓	✓	FileNotFoundException when saving with an encrypted file with name containing special characters
BD6	2	✓	✓	ZipException when using a plugin name containing special characters
BD7	2	✓	✓	FileNotFoundException when configuring with a non-existing language file
CS1	1	×	×	FileNotFoundException when providing an invalid file name to save
CS2	1	×	×	NullPointerException when generating Write Clue with an empty crossword
CS3	1	×	×	NullPointerException when generating Suggest Word with an empty crossword
DJ1	1	✓	✓	IOException when saving file with a file name containing special characters
JR1	1	×	×	StringIndexOutOfBoundsException when using "!#" as a 'Preview text' in Preferences
JR2	1	✓	✓	ArrayIndexOutOfBoundsException when allowing to generate keyword for a disabled bibtex entry
JR3	1	✓	✓	ArrayIndexOutOfBoundsException when generating keyword for an already closed bibtex file
JR4	2	✓	✓	Pattern Exception when searching with an regular expression containing the special characters ' []'
JR5	2	×	×	IOException when setting default owner name containing a '{' character
JR6	1	✓	✓	ClassNotFoundException when providing a non-existing class to setup the 'Look and Feel'
JR7	1	✓	✓	ServerSocketException with an out-of-range Proxy port
JR12	1	✓	✓	ArrayIndexOutOfBoundsException when changing properties of an already closed bibtex file
JR13	2	×	×	StringIndexOutOfBoundsException when import with a non-exist ImportFormat plugin
OT1	1	✓	✓	NullPointerException when checking spell with a blank spell-checking plugin name
PS1	1	×	×	ParseException when providing a non-numeric value for the split after these pages text filed
PS2	1	×	×	ParseException when providing a non-numeric value for the split every "n" pages text field
PS3	1	×	×	ParseException when providing a non-numeric value for the split at this size text file
PS4	1	✓	✓	NullPointerException when providing an invalid split by bookmarks level
PS7	1	×	×	FileNotFoundException when saving with an invalid environment file name
RC1	2	✓	✓	NullPointerException when generating report with a non-existing file name
RC2	2	×	×	NullPointerException when adjusting time after switching to a previous day
RC3	1	✓	✓	NumberFormatException when entering a non-numberic value to the Inactivity time text field
RC7	3	×	×	NullPointerException when simultaneously adjusting starting time and correcting the scheduled time

✓ = Yes; × = No

The most popular models used are *state machines*. Examples include variable FSM [15], Complete Interaction Sequences [16], *hierarchical state-machine models* represented as UML state diagrams [2], off-normal FSM [17], multiple FSMs called *Label Transition Systems* [18], and semantic models [14], [19], [20]. *AI planning* has also been used for test case generation [4]. A specification of the GUI is manually created by a tester in form of *planning operators*. Test cases are automatically generated by invoking a planner which searches for a path from the initial to the goal state. Memon *et al.* [36], provide a more detailed survey on model-based testing techniques for GUI-based applications. In some proposed techniques, the event context is partially captured in their models (*e.g.*, in the *states* or *planning operators*). However since the models are *manually* created, those techniques suffer from the scalability issues.

Our first reported work on *GUI Ripping*, already discussed in Section 1, in the year 2003 [23] set the stage for using the executing GUI software itself to *automatically* model its own input space. In summary, the *Ripper* starts at the *main* window of a software application under test, automatically detects all 'clickable' GUI widgets and exercises the application by systematically executing these elements. The GUI structure obtained is then converted to an EFG for test case generation. Since then, we developed techniques to augment the EFG model using annotations. For example, Yuan *et al.* [25] annotated the EFG with

semantic information derived from the runtime state of the GUI. With the augmented model, we are able to prioritize in the input space and generate longer test cases. Marchetto *et al.* [37] proposed a similar approach but applied to Web testing domain. However, none of the previous approaches was able to discover unexplored parts of the input space. This is something we do in the current paper.

In the context of random testing, Pacheco *et al.* [33] propose an object-oriented unit testing approach called Randoop. Similar to our approach, Randoop starts with a seed test suite consisting of short method-call sequences. The longer test cases are then incrementally obtained by randomly combining the shorter ones. The feedback from previous test executions are used to avoid generating redundant and illegal combinations. The key difference between Randoop and our approach is that in unit testing, it is trivial to obtain the input space, *i.e.*, the list of available methods. Execution feedback is primarily used to effectively explore the input space. However, this assumption does not hold true for the GUI testing domain where GUI events are often dynamically produced. Therefore, in our approach, the feedback is not only used to effectively explore the input space but also to *discover* it (*i.e.*, adding new GUI events).

A technique called *exploratory testing* [1], [38] is most related to our current approach. In exploratory testing, human testers explore the system under test without fully knowing the input space of the system under test. As the system is being tested, they learn the system's behaviors and manually decide what to do next. There is no predetermined test script or test input. This technique takes advantage of the testers' experiences and provides rapid feedback to the developers. However, because it heavily relies on human skills, the results are often *subjective*, *hard to replicate* [1], and *do not scale* to large systems [3].

Extending our work on GUI ripping, Mesbah *et al.* [28], [39] leverage a *crawl-based* technique to reverse engineer the structure of the website under test. A tool called CRAWLJAX automatically detects all 'clickable' web elements and crawls the website by exercising these elements. The website structure is then analyzed to construct an intermediate abstract state machine model, which is used as a skeleton to systematically generate test cases. Saxena *et al.* [40] extend this technique by adding a string constraint solver to the crawler to better explore the event space.

The remainder of the papers in this section share a common theme. During test execution, they keep track of all new event handlers, object states, web services, observed during execution and try to generate additional test cases to exercise them.

In *web application testing*, Artzi *et al.* [41] use execution states to generate additional test inputs. Due to the nature of the web applications, the event handlers can be dynamically registered to and removed from the client at runtime. An execution unit dynamically monitors the set of event registered at a particular time and attempts to exercise them.

In *object oriented unit testing*, Dallmeier *et al.* [42] dynamically synthesize a state-machine model by monitoring the object states in different executions. As test cases are executed, new object states are observed and incrementally incorporated into the original model. The extended model is then used to generate additional test cases until some stopping criterion is met. To further enhance the model, a source code scanning technique is used to extract all available method calls and invoke them from all obtained states, in a trial-error process, to reveal the possibly unobserved class behaviors. Zhang *et al.* [43] present a similar approach but use some advanced static analysis techniques to infer the constraints between method calls and their arguments. The constraints help to avoid generating illegal test cases as well as to direct test case generation toward the unexplored program behaviors to achieve a higher code coverage.

In *service oriented application (SOA)* testing, Bartolini *et al.* [44] introduce an approach to "whiten" the external binary services in support for testing the service consumers. An intermediate agent is instrumented to the services to expose their coverage level as the test cases are executed. Based on the data collected, the test case generator can infer the internal behaviors of the services and then decides how to generate test cases for the service consumers.

Our work differs from the above in several ways. First, our target domain is that of GUIs, which have enormous, even infinite, input spaces. GUI applications increasingly integrate multiple source code languages and object code formats, along with virtual function calls, reflection, multi-threading, and event-handler callbacks. These features severely impair the applicability of techniques that rely on static analysis or the availability of language-specific and format-specific instrumentation tools. Second, we have a fully automated technique. Our underlying model is an EFG, not a state machine, which, for reasons discussed in prior work [22], [25] are more appropriate for this domain. Third, our model enhancement is based on new events, not states, *i.e.*, we are extending the input alphabet of the model. Finally, most of the above approaches rely on code instrumentation. Our approach, in contrast, does not require code.

6 Conclusions

As software systems have grown increasingly complex, our testers are tasked with verifying that these systems function correctly; but the testers do not fully understand these systems' input spaces. This problem is severely compounded in GUIs that have immense, even infinite, input spaces. GUI testers routinely miss allowable event sequences, any of which may cause

failures once the software is fielded. And the tester may fail to discover that the softwares implementation allows the execution of some disallowed sequences. All this is because the tester has no knowledge of softwares overall input space, *i.e.*, the set of all possible event sequences that may be input to the software.

To address some of these challenges, we presented a new paradigm for GUI testing that we call *Observe-Model-Exercise** (OME*). We described the key features of OME* and the algorithms used to realize it. An experiment on 8 open-source applications showed that OME* did much better compared to the current state-of-the-art. In some cases, we observed more than 200% improvement in the set of events that we executed. We also discovered 34 new faults that have not been detected before.

Our work has implications on the overall GUI testing lifecycle, which may include the development of test cases via model-based techniques, capture/replay tools, and hand-coding. Armed with our OME* model—essentially a blueprint of the GUI—the tester now has a more complete picture of allowed/disallowed sequences of events. We envision that in a practical testing scenario, a tester will first run the OME* approach, obtaining the complete EFG, and use the EFG to further create test cases.

This work presents numerous opportunities for future research. In the immediate future, we will extend our subject application pool. In particular, we want to use non-Java, non-desktop (*e.g.*, web, mobile) as well as industrial applications to reduce the external threats to validity in our empirical studies. Furthermore, we used natural faults (*i.e.*, crashes) to measure fault detection effectiveness. This approach, on one hand, provides evidence that our technique can detect actual faults. On the other hand, we are limited in the analysis that we can perform. For example, we cannot examine faults that were missed. For this reason, we will seed artificial faults in future work. We will also develop metrics to evaluate the completeness (or more generally, the quality) of a constructed EFG. This is necessary because even though OME* substantially expands the test space (measured by the size of EFG) than other approaches, it is unclear how much the constructed EFG can cover a perfectly completed EFG. Finally, for input supplied to text fields, we intend to use domain specific data instead of our current general purpose database.

In the medium term, we will apply our paradigm to other test case generation techniques (*e.g.*, capture/replay, AI planning [4], feedback-driven [25]). Finally, in the long term, we will apply our techniques to test web applications and object-oriented software as they also use test cases that are sequences of events (*e.g.*, web user actions and method calls).

ACKNOWLEDGMENTS

This research was partially funded by the National Science Foundation (NSF#1205501).

REFERENCES

[1] J. A. Whittaker, *Exploratory Software Testing: Tips, Tricks, Tours, and Techniques to Guide Test Design*, 1st ed. Addison-Wesley Professional, 2009.

[2] José L. Silva, José Creissac Campos, and Ana C. R. Paiva, "Model-based User Interface Testing With Spec Explorer and ConcurTaskTrees," *Electron. Notes Theor. Comput. Sci.*, vol. 208, pp. 77–93, 2008.

[3] J. Itkonen, M. V. Mantyla, and C. Lassenius, "How do testers do it? An exploratory study on manual testing practices," in *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 494–497.

[4] Atif M. Memon, Martha E. Pollack, and Mary Lou Soffa, "Hierarchical GUI Test Case Generation Using Automated Planning," *IEEE Trans. Softw. Eng.*, vol. 27, no. 2, pp. 144–155, 2001.

[5] "Introduction to jfcUnit," 2009.

[6] "Pounder Java GUI Testing Utility," 2009.

[7] Matthew B. Dwyer, Vicki Carr, and Laura Hines, "Model checking graphical user interfaces using abstractions," in *ESEC '97/FSE-5: Proceedings of the 6th European SOFTWARE ENGINEERING conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering*. New York, NY, USA: Springer-Verlag New York, Inc., 1997, pp. 244–261.

[8] P. Mehlitz, O. Tkachuk, and M. Ujma, "JPF-AWT: Model checking GUI applications," in *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, nov. 2011, pp. 584 –587.

[9] S. Ganov, C. Killmar, S. Khurshid, and D. E. Perry, "Event Listener Analysis and Symbolic Execution for Testing GUI Applications," in *Proceedings of the 11th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering*, ser. ICFEM '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 69–87.

[10] T.-H. Chang, T. Yeh, and R. C. Miller, "GUI testing using computer vision," in *Proceedings of the 28th international conference on Human factors in computing systems*, ser. CHI '10. New York, NY, USA: ACM, 2010, pp. 1535–1544.

[11] B. Daniel, Q. Luo, M. Mirzaaghaei, D. Dig, D. Marinov, and M. Pezzè, "Automated GUI refactoring and test script repair," in *Proceedings of the First International Workshop on End-to-End Test Script Engineering*, ser. ETSE '11. New York, NY, USA: ACM, 2011, pp. 38–41.

[12] C.-Y. Huang, J.-R. Chang, and Y.-H. Chang, "Design and analysis of GUI test-case prioritization using weight-based methods," *J. Syst. Softw.*, vol. 83, no. 4, pp. 646–659, Apr. 2010.

[13] C. Bertolini and A. Mota, "Using Probabilistic Model Checking to Evaluate GUI Testing Techniques," in *Software Engineering and Formal Methods, 2009 Seventh IEEE International Conference on*, nov. 2009, pp. 115 –124.

[14] N. R. Krishnaswami and N. Benton, "A semantic model for graphical user interfaces," in *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, ser. ICFP '11. New York, NY, USA: ACM, 2011, pp. 45–57.

[15] R. K. Shehady and D. P. Siewiorek, "A Method to Automate User Interface Testing Using Variable Finite State Machines," in *Proceedings of The Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing (FTCS'97)*. Washington - Brussels - Tokyo: IEEE Press, Jun. 1997, pp. 80–88.

[16] Lee White and Husain Almezen, "Generating Test Cases for GUI Responsibilities Using Complete Interaction Sequences," in *ISSRE '00: Proceedings of the 11th International Symposium on Software Reliability Engineering*. Washington, DC, USA: IEEE Computer Society, 2000, p. 110.

[17] Fevzi Belli, "Finite-State Testing and Analysis of Graphical User Interfaces," in *ISSRE '01: Proceedings of the 12th International Symposium on Software Reliability Engineering*. Washington, DC, USA: IEEE Computer Society, 2001, p. 34.

[18] T. Pajunen, T. Takala, and M. Katara, "Model-Based Testing with a General Purpose Keyword-Driven Test Automation Framework," in *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, march 2011, pp. 242 –251.

[19] Marlon Vieira, Johanne Leduc, Bill Hasling, Rajesh Subramanyan, and Juergen Kazmeier, "Automation of GUI testing using a model-driven approach," in *AST '06: Proceedings of the 2006 international workshop on Automation of software test*. New York, NY, USA: ACM, 2006, pp. 9–14.

[20] P. L. M. Navarro, D. S. Ruiz, and G. M. Pérez, "A proposal for automatic testing of GUIs based on annotated use cases," *Adv. Soft. Eng.*, vol. 2010, pp. 5:1–5:13, Jan. 2010.

[21] F. Belli, C. J. Budnik, and L. White, "Event-based modelling, analysis and testing of user interactions: approach and case study: Research Articles," *Softw. Test. Verif. Reliab.*, vol. 16, pp. 3–32, March 2006.

[22] Atif M. Memon, Adithya Nagarajan, and Qing Xie, "Automating regression testing for evolving GUI software," *Journal of Software Maintenance*, vol. 17, no. 1, pp. 27–64, 2005.

[23] Atif M. Memon, Ishan Banerjee, and Adithya Nagarajan, "GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing," in *Proceedings of the 10th Working Conference on Reverse Engineering*, ser. WCRE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 260–.

[24] Qing Xie and Atif M. Memon, "Designing and comparing automated test oracles for GUI-based software applications," *ACM Trans. Softw. Eng. Methodol.*, vol. 16, no. 1, p. 4, 2007.

[25] Xun Yuan and Atif M. Memon, "Generating Event Sequence-Based Test Cases Using GUI Runtime State Feedback," *IEEE Transactions on Software Engineering*, vol. 36, pp. 81–95, 2010.

[26] Penelope A. Brooks and Atif M. Memon, "Automated GUI testing guided by usage profiles," in *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. New York, NY, USA: ACM, 2007, pp. 333–342.

[27] L. Baresi and M. Young, "Test Oracles," University of Oregon, Dept. of Computer and Information Science, Eugene, Oregon, U.S.A., Technical Report CIS-TR-01-02, August 2001.

[28] A. Mesbah, E. Bozdag, and A. v. Deursen, "Crawling AJAX by inferring user interface state changes," in *Proceedings of the 2008 Eighth International Conference on Web Engineering*, ser. ICWE '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 122–134.

[29] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller, "Mining object behavior with adabu," in *Proceedings of the 2006 international workshop on Dynamic systems analysis*, ser. WODA '06. New York, NY, USA: ACM, 2006, pp. 17–24.

[30] Scott McMaster and Atif M. Memon, "An Extensible Heuristic-Based Framework for GUI Test Case Maintenance," in *TESTBEDS'09: Proceedings of the First International Workshop on TESTing Techniques & Experimentation Benchmarks for Event-Driven Software*. Washington, DC, USA: IEEE Computer Society, 2009.

[31] A. Adamoli, D. Zaparanuks, M. Jovic, and M. Hauswirth, "Automated GUI performance testing," *Software Quality Control*, vol. 19, pp. 801–839, December 2011.

[32] A. Ruiz and Y. W. Price, "GUI Testing Made Easy," in *Proceedings of the Testing: Academic & Industrial Conference - Practice and Research Techniques*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 99–103.

[33] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball, "Feedback-directed random test generation," in *ICSE'07, Proceedings of the 29th International Conference on Software Engineering*, Minneapolis, MN, USA, May 23–25, 2007, pp. 75–84.

[34] A. M. Memon, M. L. Soffa, and M. E. Pollack, "Coverage criteria for GUI testing," in *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, ser. ESEC/FSE-9, vol. 26. New York, NY, USA: ACM, September 2001, pp. 256–267.

[35] A. Rountev, S. Kagan, and M. Gibas, "Evaluating the imprecision of static analysis," in *Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, ser. PASTE '04. New York, NY, USA: ACM, 2004, pp. 14–16.

[36] Atif M. Memon and Bao N. Nguyen, "Advances in automated model-based system testing of software applications with a GUI front-end," in *Advances in Computers*, M. V. Zelkowitz, Ed. Academic Press, 2010, vol. 80, pp. 121–162.

[37] A. Marchetto, P. Tonella, and F. Ricca, "State-Based Testing of Ajax Web Applications," in *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 121–130.

[38] J. Bach, "What is exploratory testing," *E. van Veenendaal, The Testing Practitioner–2nd edition, UTN Publishing*, pp. 90–72 194, 2004.

[39] A. Mesbah and A. van Deursen, "Invariant-based automatic testing of AJAX user interfaces," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 210–220.

[40] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A Symbolic Execution Framework for JavaScript," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, ser. SP '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 513–528.

[41] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip, "A framework for automated testing of javascript web applications," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 571–580.

[42] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller, "Generating test cases for specification mining," in *Proceedings of the 19th international symposium on Software testing and analysis*, ser. ISSTA '10. New York, NY, USA: ACM, 2010, pp. 85–96.

[43] S. Zhang, D. Saff, Y. Bu, and M. D. Ernst, "Combined static and dynamic automated test generation," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11. New York, NY, USA: ACM, 2011, pp. 353–363.

[44] Cesare Bartolini, Antonia Bertolino, Sebastian Elbaum, and Eda Marchetti, "Bringing white-box testing to Service Oriented Architectures through a Service Oriented Approach," *J. Syst. Softw.*, vol. 84, pp. 655–668, April 2011.

Bao N Nguyen obtained his PhD from the Department of Computer Science at the University of Maryland, College Park in 2013. His dissertation work included the development of the Observe Model Execute (OME*) paradigm described in this paper. His research interests include software testing, software modeling, and virtualization.

Atif M Memon is an Associate Professor at the Department of Computer Science, University of Maryland, where he founded and heads the Event Driven Software Lab (EDSL). His research interests include software security, program testing, software engineering, experimentation, and computational biology. He is currently working in all these areas with funding from the US National Institutes for Health, the US Defense Advanced Research Projects Agency, the US National Security Agency, and the US National Science Foundation. He is currently serving on a National Academy of Sciences panel as an expert in the area of Computer Science and Information Technology, for the Pakistan-U.S. Science and Technology Cooperative Program, sponsored by United States Agency for International Development (USAID). In addition to his research and academic interests, he handcrafts fine wood furniture.