SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2007; **17**:137–157
Published online 2 January 2007 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr.364

An event-flow model of GUI-based applications for testing

Atif M. Memon*,†

Department of Computer Science, University of Maryland, College Park, MD 20742, U.S.A.

SUMMARY

Graphical user interfaces (GUIs) are by far the most popular means used to interact with today's software. The functional correctness of a GUI is required to ensure the safety, robustness and usability of an entire software system. GUI testing techniques used in practice are resource intensive; model-based automated techniques are rarely employed. A key reason for the reluctance in the adoption of model-based solutions proposed by researchers is their limited applicability; moreover, the models are expensive to create. Over the past few years, the present author has been developing different models for various aspects of GUI testing. This paper consolidates all of the models into one scalable *event-flow model* and outlines algorithms to semi-automatically reverse-engineer the model from an implementation. Earlier work on model-based test-case generation, test-oracle creation, coverage evaluation, and regression testing is recast in terms of this model by defining *event-space exploration strategies* (ESESs) and creating an end-to-end GUI testing process. Three such ESESs are described: for checking the event-flow model, test-case generation, and test-oracle creation. Two demonstrational scenarios show the application of the model and the three ESESs for experimentation and application in GUI testing. Copyright © 2007 John Wiley & Sons, Ltd.

Received 30 November 2005; Revised 15 August 2006; Accepted 2 November 2006

KEY WORDS: event-driven software; graphical user interfaces; event-flow model; event-flow graph; integration tree; test oracles; test-case generation; model checking

1. INTRODUCTION

Graphical user interfaces (GUIs) are becoming ubiquitous as a means of interacting with today's software. Recognizing the importance of GUIs, software developers are dedicating an increasingly

*Correspondence to: Atif M. Memon, Department of Computer Science, University of Maryland, College Park, MD 20742, U.S.A.
†E-mail: atif@cs.umd.edu

Contract/grant sponsor: U.S. National Science Foundation; contract/grant number: CCF-0447864
Contract/grant sponsor: Office of Naval Research; contract/grant number: N00014-05-1-0421

large portion of software code to implementing GUIs: up to 60% of the total software code [1–5]. Although the use of GUIs continues to grow, GUI testing for functional correctness has remained, until recently, a neglected research area [6]. Adequately testing a GUI is required to help ensure the safety, robustness, and usability of an entire software system [7].

Current GUI testing techniques used in practice (discussed in detail in Section 2) require a substantial amount of manual effort on the part of the test designer. Some models and techniques have been developed to address the automation of specific aspects of the GUI testing process (e.g. test-case generation [8–13], test-oracle creation [14], and regression testing [15,16]). The author's earlier work has used *goal-directed search* for GUI test case generation [10,11], *function composition* for automated test-oracle creation [14], *metrics from graph theory* to define test coverage criteria for GUIs [17], *graph-traversal* to obtain *smoke test cases for GUIs* that are used to stabilize daily software builds [12,13], and *graph matching algorithms* to repair previously unusable GUI test cases for regression testing [16]. However, because the models were developed to address specific (often one) GUI testing problem(s), they have narrow focus; some of these models are expensive to create and have limited applicability; therefore, they are unable to address the full scope of the problem of GUI testing.

This paper consolidates all of the author's above existing GUI models into one general *event-flow model* that represents events and event interactions. In much the same way as a control-flow model represents all possible execution paths in a program [18], and a data-flow model represents all possible definitions and uses of a memory location [19], the event-flow model represents all possible sequences of events that can be executed on the GUI. More specifically, a GUI is decomposed into a hierarchy of *modal dialogues* (discussed in Section 3); this hierarchy is represented as an *integration tree*; each modal dialogue is represented as an *event-flow graph* that shows all possible event execution paths in the dialogue; individual events are represented using their *preconditions* and *effects*. An overview of the event-flow model with associated algorithms to semi-automatically reverse-engineer the model from an executing GUI software is presented.

As the event-flow model is not tied to a specific aspect of the GUI testing process, it may be used to perform a wide variety of testing tasks by defining specialized model-based techniques called *event-space exploration strategies* (ESESs). These ESESs use the event-flow model in a number of ways to develop an end-to-end GUI testing process. All of the previously published techniques are recast as ESESs. This paper describes three ESESs that address three complex problems faced during model-based testing, namely checking the model, test case generation, and test-oracle creation.

Two demonstrational scenarios show the application of this model for research and practice in automated testing. More specifically, these scenarios show that: (1) once the event-flow model is created, it can be used to generate a large number of GUI test cases with very little cost and effort; (2) automated tools, built around the event-flow model, greatly simplify both model building as well as many GUI testing tasks; and (3) the model and tools enable large experiments in GUI testing.

The contributions of this paper include the following.

- The consolidation of several existing GUI testing models into one event-flow model and its separation from specific testing tasks.
- The enhancement of the general event-flow model with customized ESESs to perform specific GUI testing tasks, leading to a more general solution to the GUI testing problem.
- The employment of three ESESs (for model checking, test-case generation, and test-oracle creation) to develop an end-to-end GUI testing process and the demonstration of this process on non-trivial GUI-based applications via a scenario.

DOI: 10.1002/stvr

- Demonstration that the tools developed around the event-flow model are useful for experimentation in GUI testing.

Structure of the paper

The next section provides an assessment of models, techniques, and tools currently used for GUI testing. Section 3 describes the event-flow model. Section 4 describes algorithms to create the event-flow model. Section 5 outlines some ways of using the model via the development of three ESESs for checking the model, test-case generation, and test-oracle creation. Section 6 describes two scenarios that demonstrate the usefulness of the event-flow model. Finally, Section 7 concludes with a summary of its limitations and on-going work.

2. ASSESSMENT OF CURRENT GUI MODELS, TECHNIQUES, AND TOOLS

There have been a few attempts to develop models to automate some aspects of GUI testing. The most popular amongst them are state-machine models that have been proposed to generate test cases for GUIs [20–23]. The key idea of using these models is that a test designer represents a GUI's behaviour as a state machine; each input event may trigger an abstract state transition in the machine. A path, i.e. sequence of edges followed during transitions, in the state machine represents a test case. The state machine's abstract states may be used to verify the GUI's concrete state during test case execution. Shehady and Siewiorek [24] argue that the commonly used finite-state machines (FSMs) have scaling problems for large GUIs; they have developed variations called variable FSMs that reduce the number of abstract states by adding variables to the model. In practice, these models are created manually. Also, if these models are used for verification, via an automated test oracle, effective mappings between the machine's abstract states and the GUI's concrete state also need to be developed manually.

White *et al*. present a different state-machine model for GUI test-case generation that partitions the state space into different machines based on user tasks [8,9]. The test designer/expert identifies a *responsibility*, i.e. a user task that can be performed with the GUI. For each responsibility, the test designer then identifies a machine model called the 'complete interaction sequence' (CIS). The CIS is then used to generate focused test cases. This approach is successful in partitioning the GUI into manageable functional units that can be tested separately. The definition of responsibilities and the subsequent CIS is relatively straightforward; the large manual effort is in designing the FSM model for testing, especially when code is not available.

Models of user behaviour have also been used to generate test cases, specifically to mimic *novice users'* inputs [25]. This approach relies on an expert to first manually generate a sequence of GUI events for a given user task. A genetic algorithm technique is then used to modify and lengthen the sequence, thereby mimicking a novice user [26,27]. The underlying assumption is that novice users often take indirect paths through GUIs, whereas expert users take shorter, more direct paths. This technique requires a substantial amount of manual effort, and cannot be used to generate other types of test cases.

All of the above techniques have been developed for test-case generation, side-stepping complex issues of test-oracle creation, test coverage, and regression testing. The only exception is White [15]

DOI: 10.1002/stvr

who proposes a Latin square method to reduce the size of a GUI regression test suite. The underlying assumption is that it is enough to check pairwise interactions between components of the GUI. The technique requires that each menu item appears in at least one test case. Redundant test cases, i.e. those that do not use any new menu items, are subsequently removed, resulting in a smaller regression test suite. The technique needs to be extended to GUI items other than menus. In another research reported by White *et al*. [28], the CIS is extended to develop a selective regression approach based on identifying the changed and affected objects and CISs. This technique is aided by memory diagnostic tools such as *Memory Doctor* and *WinGauge* to reveal complex problems.

As the above models were developed with one aspect of GUI testing in mind (most for test-case generation), they suffer from their lack of applicability to other aspects of GUI testing. Moreover, all of the above techniques require substantial manual effort. Consequently, none of these techniques are in common use in industry.

Current GUI testing techniques *used in practice* are incomplete, *ad hoc*, and largely manual. The most popular tools used to test GUIs are capture/replay tools such as WinRunner[‡] that provide very little automation [29], especially for *creating* test cases and test oracles, and to evaluate test coverage. A tester uses these tools in two phases: a capture and then a replay phase. During the capture phase, a tester manually interacts with the GUI being tested and performs events. The tool records the interactions and a part of the GUI's response/state as specified by the tester. The recorded test cases can be replayed automatically on (a modified version of) the software using the replay part of the tool. The previously captured state can (with some limitations) be used to check the GUI's execution for correctness. As can be imagined, these tools require a significant amount of manual effort. Testers who employ these tools typically come up with a small number of test cases [10]. Moreover, as the GUI evolves, many test cases become *obsolete* and should be removed from the test suite [16].

Other tools used for GUI testing require *programming* each GUI test case. These tools include extensions of *JUnit* such as *JFCUnit*, *Abbot*, *Pounder*, and *Jemmy Module*[§]. Not only must the tester manually program the event interactions to test, the tester must also specify expected GUI behaviour.

All of the above models and techniques have been leveraged to develop a new consolidated model that is general, in that it can be used throughout the GUI testing process. Automated tools are developed around the event-flow model that can be used by a tester to semi-automatically create the model and then perform GUI testing. The event-flow model and the tools are described in subsequent sections.

3. EVENT-FLOW MODEL

The event flow model contains two parts. The first part encodes each event in terms of *preconditions*, i.e. the state in which the event may be executed, and *effects*, i.e. the changes to the state after the event has executed. The second part represents all possible sequences of events that can be executed on the GUI as a set of directed graphs. As the details of each part of the representation have been presented in earlier reported work, a brief overview is given here.

[‡]See http://mercuryinteractive.com.
[§]See http://junit.org/news/extension/gui/index.htm.

DOI: 10.1002/stvr

Both of these parts play important, often complementary roles for various GUI testing tasks. For example, the preconditions/effects have been used for goal-directed test-case generation [11], test-oracle creation [14], and checking the model; the second part has been used for graph-traversal-based test-case generation [30], test-coverage evaluation [17], genetic-algorithm-based test-case generation, and Bayesian-network-based test-case generation. Together, the two parts are used to represent user profiles [31] and check for run-time consistency of the GUI. This paper combines these two parts of the event-flow model to detect two, complementary, classes of faults: *Type 1*, the GUI software does not perform as documented in the specifications and/or design; and *Type 2*, unacceptable software behaviour for which there may be no explicit specifications or design (e.g. software crashes, screen 'freezing').

3.1. What is a GUI?

To provide focus, this paper will discuss the 'core' event-flow model that is sufficient to represent an important class of GUIs. The important characteristics of GUIs in this class include their graphical orientation, event-driven input, hierarchical structure of menus and windows, the objects (widgets, windows, frames) they contain, and the properties (attributes) of those objects. Formally, the class of GUIs of interest may be defined as follows.

Definition 1. A GUI is a hierarchical, graphical front-end to a software system that accepts as input user-generated and system-generated events from a fixed set of events and produces deterministic graphical output. A GUI contains graphical *objects*; each object has a fixed set of *properties*. At any time during the execution of the GUI, these properties have discrete values, the set of which constitutes the state of the GUI.

The above definition specifies a class of GUIs that have a fixed set of events with deterministic outcome that can be performed on objects with discrete-valued properties. This definition would need to be extended for other GUI classes such as Web-user interfaces that have synchronization/timing constraints among objects, movie players that show a continuous stream of video rather than a sequence of discrete frames, and non-deterministic GUIs in which it is not possible to model the state of the software in its entirety (e.g. due to possible interactions with the system's memory or other system elements) and, hence, the effect of an event cannot be predicted. This paper focuses on the event-flow model and techniques to test the class of GUIs defined above.

3.2. Modelling events

An important part of the event-flow model is the behaviour of each event. Each event is represented in terms of how it modifies the *state* of a GUI when it is executed. The state of the GUI is the collective state of each of its widgets (e.g. buttons, menus) and containers (e.g. frames, windows) that contain other widgets (all widgets and containers will be referred to as GUI objects). Each GUI object is represented by a set of *properties* of those objects (`background-colour`, `font`, `caption`, etc.). The set of objects and their properties is used to create a model of the *state* of the GUI. The *state* of a GUI at a particular time t is the set P of all of the properties of all of the objects O that the GUI contains. A *complete* description of the state would contain information about the types of *all* of

DOI: 10.1002/stvr

the objects currently extant (not all possible objects) in the GUI, as well as *all* of the property values (not all possible property values) of each of those objects.

Events performed on the GUI change its state and, hence, are modelled as state transducers. The *events* $E = \{e_1, e_2, \ldots, e_n\}$ associated with a GUI are functions from one state of the GUI to another state of the GUI. As events may be performed on different types of objects, in different contexts, yielding different behaviour, they are parametrized with objects and property values. It is, of course, infeasible to give exhaustive specifications of the state mapping for each event: in principle, as there is no limit to the number of objects a GUI can contain at any point in time, there can be infinitely many states of the GUI[¶]. Hence, GUI events are represented using *operators*, which specify their preconditions and effects. This representation for encoding operators is used in the artificial intelligence (AI) planning literature [32–34]. This representation has been adopted for GUI testing because of its power to express complex actions, and the ability to reuse a large library of tools (e.g. parsers, checkers) that are available from the AI planning community. Finally, this representation allows the use of AI planning for test case generation [10] and, as will be seen later in the demonstrational scenarios, for checking the model.

3.3. Modelling event interactions

The goal of the event-flow model is to represent all possible event interactions in the GUI. Such a representation of the event interaction space will enable the customization of various ESESs that traverse parts of this space for automated testing. A graph model of the GUI is constructed: each vertex represents an event (e.g. click-on-Edit, click-on-Paste)[‖]. An edge from vertex x to vertex y shows that an event y can be performed immediately after event x (i.e. y `follows` x). This graph is analogous to a control-flow graph in which vertices represent program statements (in some cases basic blocks) and edges represent possible execution ordering between the statements. Also note that a state machine model that is equivalent to this graph can be constructed: the state would capture the possible events that can be executed on the GUI at any instant; transitions cause state changes whenever the number and type of available events changes. For a pathological GUI that has no restrictions on event ordering and no windows/menus, such a graph would be fully connected. In practice, however, GUIs are hierarchical, and this hierarchy may be exploited to identify groups of GUI events that may be modelled in isolation. One hierarchy of the GUI and that is used in this research is obtained by examining the structure of *modal windows*[**] in the GUI. A *modal window* is a GUI window that, once invoked, monopolizes the GUI interaction, restricting the focus of the user to a specific range of events within the window, until the window is explicitly terminated. Other windows in the GUI that do not restrict the user's focus are called *modeless windows*; they merely expand the set of GUI events available to the user.

[¶]Of course, in practice, there are memory limits on the machine on which the GUI is running and, hence, only finitely many states are actually possible, but the number of possible states will be extremely large.

[‖]In subsequent discussion, for brevity, the names of events will be abbreviated, e.g. Edit and Paste.

[**]Standard GUI terminology; see detailed explanations at
http://msdn.microsoft.com/library/en-us/vbcon/html/vbtskdisplayingmodelessform.asp and
http://documents.wolfram.com/v4/AddOns/JLink/1.2.7.3.html.

DOI: 10.1002/stvr

At all times during interaction with the GUI, the user interacts with events within a *modal dialogue*. This modal dialogue consists of a modal window X and a set of modeless windows that have been invoked, either directly or indirectly from X. The modal dialogue remains in place until X is explicitly terminated. By definition, a GUI user cannot interleave events of one modal dialogue with events of other modal dialogues; the user must either explicitly terminate the currently active modal dialogue or invoke another modal dialogue to execute events in different dialogues. This property of modal dialogues enables the decomposition of a GUI into parts: each part can be tested separately. Event interactions within a modal dialogue may be represented as an event-flow graph. An *event-flow graph* of a modal dialogue represents all possible event sequences that can be executed in the dialogue. Once all of the modal dialogues of the GUI have been represented as event-flow graphs, the remaining step is to identify interactions between modal dialogues. A structure called an *integration tree* is constructed to identify interactions (invocations) between modal dialogues. Modal dialogue MD_x `invokes` modal dialogue MD_y if MD_x contains an event e_x that invokes MD_y. The integration tree shows the `invokes` relationship among all of the modal dialogues in a GUI. This decomposition of the GUI makes the overall testing process intuitive for the test designer because the test designer can focus on a specific part of the GUI. Moreover, it simplifies the design of algorithms and makes the overall testing process more efficient.

4. OBTAINING THE EVENT-FLOW MODEL

This section describes techniques to construct the event-flow model for a given GUI. It contains three parts: (1) algorithms to create event-flow graphs and an integration tree; (2) a technique to partially create operators from the event-flow graphs; (3) outline of a reverse-engineering process that is used to obtain the event-flow model semi-automatically.

4.1. Construction of the event-flow graphs and integration tree

The construction of the event-flow graphs and integration tree is based on the identification of modal dialogues and, hence, the identification of modal and modeless windows and their `invokes` relationships. A classification of GUI events is used to identify modal and modeless windows. *Restricted-focus events* open *modal windows*. *Unrestricted-focus events* open *modeless windows*. *Termination events* close modal windows. *Menu-open events* are used to open menus. *System-interaction events* are not used to manipulate the structure of the GUI; rather, they are used to interact with the underlying software to perform some action.

The classification of events is used by an algorithm to construct event-flow graphs for a GUI. The algorithm computes the set of `follows` for each event. These sets are then used to create the edges of the event-flow graph.

The set of `follows`($\mathbf{v}$) can be determined using the algorithm in Figure 1 for each vertex $\mathbf{v}$. The recursive algorithm contains a switch structure that assigns `follows`($\mathbf{v}$) according to the type of each event. If the type of the event $\mathbf{v}$ is a menu-open event (`line 2`) and $\mathbf{v} \in \mathbf{B}$ (events that are available when a modal dialogue is invoked), then the user may either perform $\mathbf{v}$ again, its sub-menu choices, or any event in $\mathbf{B}$ (`line 4`). However, if $\mathbf{v} \notin \mathbf{B}$, then the user may either perform all sub-menu choices of $\mathbf{v}$, $\mathbf{v}$ itself, or all events in `follows(parent(`$\mathbf{v}$`))` (`line 6`); `parent(`$\mathbf{v}$`)` is defined as any event that makes $\mathbf{v}$ available. If $\mathbf{v}$ is a system-interaction event, then after performing $\mathbf{v}$,

DOI: 10.1002/stvr

```
ALGORITHM : GetFollows(
  v: Vertex or Event){                                          1
  IF EventType(v) = menu-open                                   2
   IF v ∈ B of the modal dialogue that contains v               3
    return(MenuChoices(v) ∪ {v} ∪ B)                            4
   ELSE                                                         5
    return(MenuChoices(v) ∪ {v}
      ∪ follows(parent(v)));                                    6
  IF EventType(v) = system-interaction                          7
   return(B);                                                   8
  IF EventType(v) = termination                                 9
   return(B of Invoking modal dialogue);                        10
  IF EventType(v) = unrestricted-focus                          11
   return(B ∪ B of Invoked modal dialogue);                     12
  IF EventType(v) = restricted-focus                            13
   return(B of Invoked modal dialogue);                         14
}
```

Figure 1. Computing follows(**v**) for a vertex **v**.

the GUI reverts back to the events in **B** (line 8). If **v** is a termination event, i.e. an event that terminates a modal dialogue, then follows(**v**) consists of all of the top-level events of the invoking modal dialogue (line 10). If the event type of **v** is an unrestricted-focus event, then the available events are all top-level events of the invoked modal dialogue available as well as all events of the invoking modal dialogue (line 12). Lastly, if **v** is a restricted-focus event, then only the events of the invoked modal dialogue are available.

It is relatively straightforward to obtain the integration tree from the computation of follows. Modifying Lines 13...14 of the algorithm shown in Figure 1, one can keep track of the modal dialogues invoked. Once all of the modal dialogues in the GUI have been identified, the integration tree may be constructed by adding, for each restricted-focus event e_x, the element (MD_x, MD_y) to a new structure $\mathcal{B}$ where MD_x is the modal dialogue that contains e_x and MD_y is the modal dialogue that it invokes.

4.2. Operators

Some of the information encoded in the event-flow graphs and integration tree can be represented as preconditions and effects. This information is retrieved and used to partially create the operators automatically. More specifically, templates are automatically generated for each operator and structural properties of the GUI are automatically filled-in. The test designer has to hand-code the other non-structural properties of the GUI.

In practice, the test designer may use an iterative process to code the operators. Such a process would involve coding a set of operators for a group of related events (i.e. those that could be used together to perform a task) followed by model checking, a well-known quality assurance activity commonly used to check the correctness of such models. As correctness problems (called *errors*) are discovered during model checking, the operators are fixed and rechecked. A similar process is used in the demonstrational scenarios in Section 6.

DOI: 10.1002/stvr

4.3. Reverse-engineering

Developing the event-flow model can be tedious and error-prone. Therefore, a tool has been developed called the 'GUI Ripper' to automatically obtain event-flow graphs and the integration tree. A detailed discussion of the tool is beyond the scope of this paper; the interested reader is referred to previously published work [35] for details. In short, the GUI Ripper combines reverse-engineering techniques with the algorithms presented in previous sections to automatically construct the event-flow graphs and integration tree. During 'GUI Ripping', the GUI application is executed automatically; the application's windows are opened in a depth-first manner. The GUI Ripper extracts all of the widgets and their properties from the GUI. During the reverse-engineering process, in addition to widget properties, additional key attributes of each widget are recovered (e.g., whether it is enabled, it opens a modal/modeless window, it opens a menu, it closes a window, it is a button, it is an editable text field). These attributes are used to construct the event-flow graphs and integration tree.

As can be imagined, the GUI Ripper is not perfect, i.e. parts of the retrieved information may be incomplete/incorrect. Common examples include: (1) missing windows in certain cases, e.g. if the button that opens that window is disabled during GUI Ripping; (2) failure to recognize that a button closes a window; and (3) incorrectly identifying a modal window as a modeless window or *vice versa*. The specific problems that are encountered depend on the platform used to implement the GUI. For example, for GUIs implemented using Java Swing, the ripper is unable to retrieve the contents of the 'Print' dialogue; in MS Windows, the ripper is unable to correctly identify modal/modeless windows. Recognizing that such problems may occur during reverse-engineering, tools have been developed that may be used to manually 'edit' the event-flow graphs and integration tree and fix these problems. For example, in the demonstrational scenarios, once the 'Print' dialogue was missed during the reverse-engineering process, the test designer manually interacted with the subject application, and opened the Print dialogue; one of the tools then reverse-engineered the dialogue, created its event-flow graph automatically, and added it to the integration tree.

A test designer also does not have to code each operator from scratch because the reverse-engineering technique creates operator templates and fills in those preconditions and effects that describe the structure of the GUI. Such preconditions and effects are automatically derived during the reverse-engineering process in a matter of seconds. There are no errors in these templates because the structure has already been manually examined and corrected in the event-flow graphs and integration trees. The number of operators is the same as the number of events in the GUI, as there is exactly one operator per executable event.

5. USING THE EVENT-FLOW MODEL VIA ESESS

The definition of the event-flow model, i.e. in terms of all possible event interactions and preconditions/effects of each event, allows the use of the model for numerous applications via the definition of simple ESESs. In this section, three of these simple ESESs are described, focusing on how they have been used for automated model-based testing. These ESESs will be used in the scenarios in Section 6.

DOI: 10.1002/stvr

5.1. Goal-directed search for model checking

Model checking involves checking whether a model (i.e. the operators in this case) allows certain invalid states to be reached from a known valid state. For example, the current specifications for TerpSpreadSheet (a spreadsheet application used in the scenarios in Section 6) do not allow non-contiguous areas of the spreadsheet to be selected. This 'invalid state' (i.e. non-contiguous areas selected) is described to a model checker to determine whether the model allows this state to be reached from a known valid state. If it does, then the model is incorrect; the model checker gives a counter-example (sequence of GUI events) that shows how the invalid state was reached. If the invalid state is not reachable, the model is correct; the model checker fails to find a counter-example. A variety of model checkers may be used [36].

For model checking, previous work on using AI planning for GUI testing [11] was leveraged. Owing to this previous work, the operators were described in the PDDL language that is used by AI planners. The planner is now used as a model checker. Planning is a goal-directed search technique used to find sequences of actions to accomplish a specific task. For the purpose of model checking, given a task (encoded as a pair of initial and goal states) and a set of actions (encoded as a set of operators), the planner returns a sequence of instantiated actions that, when executed, will transform the initial state to the goal state[††]. If no such sequence exists, then the operators cannot be used for the task and thus the planner returns 'no plan'. In this way, the planner has been used as a model checker, i.e. the goal state is a description of the invalid state. If the planner is successful in finding a valid plan to reach the invalid state, the plan itself becomes the counter-example.

5.2. Graph-exploration for test-case generation

A GUI test case is of the form $\langle S_0, e_1; e_2; \ldots ; e_n \rangle$, where S_0 is a state of the GUI in which the event sequence $e_1; e_2; \ldots ; e_n$ is executed. The simplest way to generate test cases is for a tester to start from one of the vertices in the set B of the *Main* modal dialogue's event-flow graph. These events are available to a GUI user as soon as the GUI is invoked. The event corresponding to the chosen vertex becomes the first event in the test case. The tester can then use one of the outgoing edges from this vertex to perform an adjacent event. The tester can continue this process, generating many test cases. A tester who uses a capture/replay tool to generate test cases is actually executing this process manually without using the formal model.

As noted earlier, if performed manually (using capture/replay tools), the above process is extremely labor intensive. With the event-flow model, numerous graph-traversal techniques may be used to automate it. The order in which the events are covered will yield different types of test cases. For example, a tester may generate test cases that: (1) cover all of the events in the GUI at least once; (2) cover all pairs of event interactions at least once; or (3) satisfy some code coverage criteria, e.g., statement or branch coverage. In the scenarios (Section 6), a graph-traversal algorithm is used that covers each event with at least five test cases. Other graph-exploration techniques have been used in previous work [30,37].

[††]The planner actually returns a *partial-order plan*, but the complexities of partial ordering are ignored here to simplify the discussion.

DOI: 10.1002/stvr

Graph traversal may yield different types of test cases; a test oracle is used to determine whether the test case passed or failed. A 'general' test oracle, e.g., one that examines the execution for software crashes and/or freezing, may be used to detect certain failures of Type 2. Failures of Type 1, on the other hand, require oracles derived from the specifications, discussed next.

5.3. Operator execution for test-oracle creation

A test oracle is a mechanism that determines whether a piece of software executed correctly for a test case. The test oracle may either be automated or manual; in both cases, the actual output is compared with a presumably correct expected output. The design for a GUI test oracle was presented in earlier work [14]; it contains an *execution monitor* that extracts the actual state of a GUI using reverse-engineering technology [35] as a test case is executed on it, an *oracle procedure* that uses *set equality* to compare the actual state with *oracle information*, i.e. the expected state. The oracle information for a test case $\langle S_0, e_1; e_2; \ldots ; e_n \rangle$ is defined as a sequence of states $S_1; S_2; \ldots ; S_n$ such that S_i is the expected state of the GUI after event e_i is executed.

An ESES that uses the operators to obtain the oracle information was described in earlier reported work [14]; it is summarized here. Recall that the event-flow model represents events as state transducers. The preconditions–effects-style of encoding the operators makes it fairly straightforward to derive the expected state. Given the GUI in state S_{i-1}, the next state S_i (i.e. the expected state after event e_i is executed) may be computed using the effects of the operator Op representing event e_i via simple additions and deletions to the list of properties representing state S_{i-1}. This expected state is used to create a test oracle during test case execution.

A test case that uses test oracles obtained from operator execution is able to detect failures of Type 1, i.e. any deviations from the design and/or specifications. Combined with a general test oracle that detects software crashes, the event-flow model is able to detect both Type 1 and Type 2 failure classes.

6. DEMONSTRATIONAL SCENARIOS

This section describes two scenarios that demonstrate the usefulness of the event-flow model and the three ESESs. The purpose of these scenarios is not to provide a 'perfect' solution to the GUI testing problem. Practitioners and researchers will need to fine-tune the presented techniques for their specific needs to maximize fault detection. Rather, the goal of the scenarios is to: (1) present to the reader a step-by-step process of using the event-flow model, checking the model, test-case generation, and test-oracle creation to generate and execute GUI test cases; (2) show that the event-flow model and tools enable experimentation in GUI testing; and (3) show that the model promotes reusability, because, once created, it can be quickly redeployed for further testing and future maintenance of the GUI.

6.1. Scenario 1: model-based GUI testing

This first scenario is from the perspective of a GUI tester. Automated tools, developed around the event-flow model, are used to perform GUI testing. Several subject applications are selected, their intended functionality is described in terms of operators, and event-flow graphs and the integration tree are used to generate test cases. In particular, this scenario explores the following questions. (1) What is the cost

DOI: 10.1002/stvr

Table I. The subject applications.

Subject application	Windows	Widgets	Events	LOC	Classes	Methods	Branches
TerpCalc	4	132	92	9916	141	446	1306
TerpPaint	10	453	220	18 376	219	644	1277
TerpSpreadSheet	8	329	165	12 791	125	579	1521
TerpWord	12	211	132	4893	104	236	452
Total	34	1125	609	45 976	589	1905	4556

of developing the event-flow model? How much manual effort is required to create the model? (2) Can the event-flow model, combined with simple ESESs, be used to generate test cases and test oracles that are effective at detecting faults? The following steps are performed.

(1) Choose subject applications with GUI frontends.
(2) Use the reverse-engineering technique to obtain the event-flow graphs and integration tree. Manually examine them for correctness and fix problems.
(3) Use reverse-engineering to develop templates for the operators. Manually complete the operators and use the model-checking technique (Section 5.1) to check their correctness.
(4) Generate test cases using the ESESs outlined in Section 5.2.
(5) Generate test oracle information for each test case using the ESESs outlined in Section 5.3.
(6) Execute test cases and report detected faults.

The time and effort required for each of the above steps will be measured and reported. These steps are now discussed in detail.

6.1.1. Subject applications

The applications to be tested are part of an open-source office suite developed at the Department of Computer Science of the University of Maryland by undergraduate students of the senior Software Engineering course. It is called TerpOffice[‡‡] and includes TerpCalc (a scientific calculator with graphing capability), TerpPaint (an image editing/manipulation program), TerpSpreadSheet (a spreadsheet application), and TerpWord (a small wordprocessor). They have been implemented using Java. Table I summarizes the characteristics of these applications. These applications are fairly large with complex GUIs. In terms of number of windows/widgets, TerpPaint, TerpSpreadSheet, and TerpWord are of the same size as MS WordPad. The widget column shows the number of widgets in the GUI; the number of executable events listed in the table are those on which user events can be executed (e.g. text labels are not included).

[‡‡] See http://www.cs.umd.edu/users/atif/TerpOffice.

DOI: 10.1002/stvr

Table II. Results of GUI Ripping.

Application	Rip time (s)	Obtained windows	Missed windows	Manual effort (s)	Size (kB)
TerpCalc	29	4	0	300	15.1
TerpPaint	42	7	3	420	24.5
TerpSpreadSheet	89	7	1	420	72.8
TerpWord	40	10	2	360	53.8

6.1.2. Event-flow graphs and integration tree

The reverse-engineering tool was executed on the four subject applications without human intervention. The results are shown in Table II. The tool was able to automatically detect a large fraction of the total number of windows in all applications. The missed windows included some system-dependent windows such as 'Print' and 'Page Setup', which do not export their window handles and contents to other applications. However, the tool was able to recover all windows for TerpCalc. All of the information was recovered in less than 50 seconds per application. As noted earlier, the windows that were missed by the automated reverse-engineering process needed to be identified manually. The manual effort involved verifying the correctness of the models plus adding the missed windows using a tool. The total time required for the manual effort for the subject applications is shown in column 5 of Table II. The total time (to locate the window and then to rip it) needed per application is a few minutes. The time will be significantly smaller if testers are familiar with the software's GUI. Column 6 of Table II shows the size (stored as XML files) of the event-flow graphs and integration tree for all the subject applications. As the results show, only a few hundred kilobytes of storage space was required.

6.1.3. Operators

Having fixed the event-flow graphs and integration trees of all of the subject applications, the next step was to create the operators for each event. The reverse-engineering tool automatically generated the templates of each operator. As discussed in Section 4.2, the tool creates one template per executable event; structural preconditions/effects (e.g., event `Open` opens the `FileOpen` modal dialogue, event `File` opens a pull-down menu, event `Open` in the `FileOpen` dialogue closes the dialogue) are also automatically filled-in. Other non-structural preconditions/effects were hand-coded.

As noted earlier, the operators are checked via model checking. For model checking, 100 invalid states were created and described as goal states for the planner. The total time is computed as the sum of (a) time required to code the operators, (b) time to run the planner, and (c) time to fix the operators. A text editor was enhanced to measure time; testers had to press an `Edit` button before editing an operator and a `Fix` button to debug it. In all, approximately two days were needed for each subject application. The exact time required is shown in Figure 2. The time for model checking is very small,

DOI: 10.1002/stvr

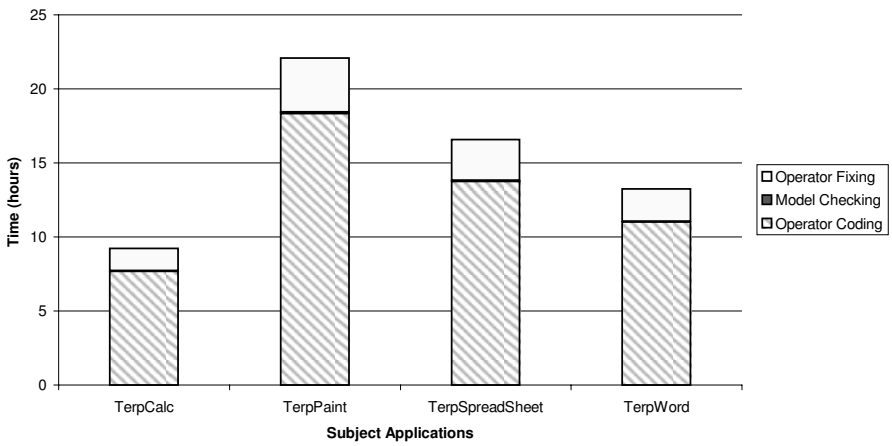

Figure 2. Time required to code the operators.

hence hardly visible. A significant portion of the time is spent on coding (in PDDL) and fixing the operators. TerpPaint required 22 hours and TerpCalc required 9 hours.

The size (in terms of lines of PDDL code) of the final operators was 1561, 2158, 2498, and 952 for TerpCalc, TerpPaint, TerpSpreadSheet, and TerpWord, respectively. The average operator size varied between 10–12 lines of PDDL code. Examples of operators and tasks are available at http://www.cs.umd.edu/users/atif. Noting that many applications share events, a library of operators for commonly used events such as file-open, file-save, edit-cut, edit-paste, and edit-copy is currently being developed. In the future, testers can use this library, thereby reducing the time required to create the model and avoid coding errors.

6.1.4. Test-case generation

As noted earlier, once a model of a GUI's event space has been created, there are many ways to traverse it. Earlier work used goal-directed search [11], random walk [37], and bounded breadth-first search [30]. In this scenario, a new ESES was developed that generated test cases so that each GUI event was covered by at least five test cases. Other ESESs, may be implemented depending on the testing situation.

The ESES itself was straightforward to encode; a graph-traversal technique was implemented that kept track of the number of times an event had been used in the test suite. Starting in one of the events in the application's main window, the event-flow graphs and integration trees were traversed. The upper bound on the number of events in an event sequence was 50, i.e. the longest test case had 50 events. Once the 'each event covered by at least five test cases' criterion had been satisfied, additional test cases were generated to obtain a total of 1000 test cases per application (the need for 1000 test cases will be explained in Scenario 2). The start state for each test case was the state of the subject application in which it was launched.

DOI: 10.1002/stvr

6.1.5. *Test-oracle creation*

The technique described in Section 5.3 was used to generate test-oracle information. The final test suite consisted of test cases with the expected state after each event. During testing, the oracle will check all properties as specified in each operator's effects. Any mismatch between the expected (from the operators) and actual (from the GUI) states will be reported as a test-case failure. Widget positions were ignored by the oracle because the windowing system launches the GUI at a different screen location each time it is invoked.

6.1.6. *Test-case execution*

All of the test cases were executed on their corresponding subject application. Each test case required approximately 5 seconds to execute. The time varied by application and the number of GUI events in the test case. The total execution time for TerpCalc, TerpPaint, TerpSpreadSheet, and TerpWord was 2, 3, 2.5, and 2 hours, respectively. The execution included launching the application under test, replaying GUI events from a test case on it and analyzing the resulting GUI states. The analysis consisted of recording the actual GUI states and determining the result of the test-case execution. The test cases executed on four machines (Pentium 4, 2.2 GHz, each with 256 MB RAM) simultaneously for 2 hours.

6.2. Observations from Scenario 1

Figure 3 summarizes the results of this scenario. The column graph shows four columns per application. (1) A test-case *failure* was a mismatch between an expected property value and observed (actual) value. (2) There were two possible reasons for a failure: (a) the application being tested was flawed or (b) the application was correct and the operators had errors; recall that problems in operators are called errors in this paper. These failures are called *false positives*. The probability that the subject application was flawed *and* the operators had errors that masked the flaw is so small that it is ignored in this scenario. The first two columns for each application show the number of (real) failures and false positives, respectively. The former correspond to failures of Type 1. The problematic operators that led to false positives were later fixed. (3) In some rare cases, the software under test 'crashed' (terminated unexpectedly) during test-case execution; the number of crashes is shown in the third column for each application. This number is not included in the failures column. As the detection of these faults does not require specifications, they are of Type 2; note that the software subjects used in this study did not freeze during test execution. (4) The software applications were later debugged to find the cause of problems. In many cases, a single fault had led to many failures/crashes. The actual number of faults that caused the failures/crashes is shown in the fourth column.

The observations from this scenario illustrate several points. First, reverse-engineering greatly helps to reduce the 'boot time' typically associated with model-based approaches. More time would have been needed to create the model from scratch. Second, having invested time and effort in creating a model of the GUI's event-space, additional ESESs may be created for test-case generation, each with its own costs and fault-detection ability. Manual testing does not promote this type of reuse. Third, if the GUI is modified, a tester can use a modified event-flow model to generate additional test cases in a matter of seconds and automatically execute them. Finally, two days were spent on each application for the overall testing process; a significant portion of the time was spent in defining the operators.

DOI: 10.1002/stvr

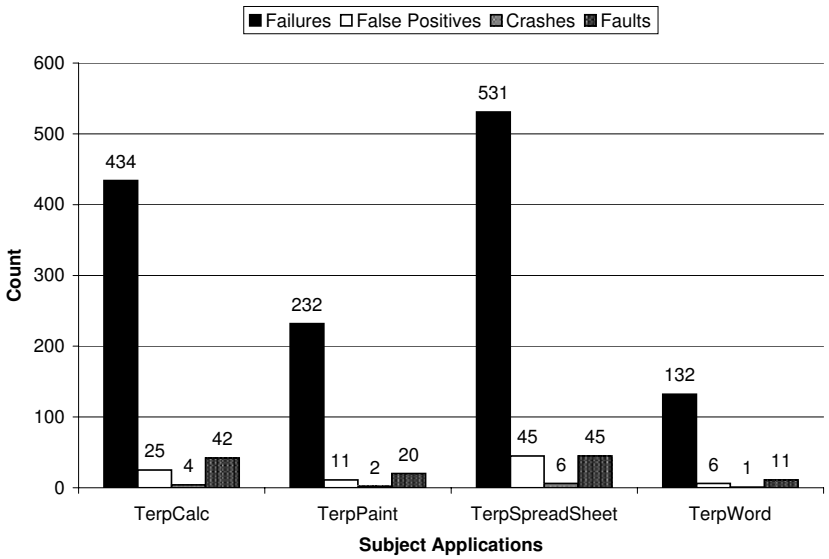

Figure 3. Number of failures, false positives, crashes, and faults found.

6.3. Scenario 2: use the event-flow model and tools for experimentation

This scenario is from the perspective of an experimenter. As was the case for the first scenario, the goal here is not to come up with a 'perfect' experiment. Rather, due to lack of space, it is to demonstrate that the event-flow model and its tools can be used to construct a small 'mock' experiment. The interested reader is referred to additional papers for more comprehensive experiments [12,38].

The goal of this mock experiment is to determine how the test cases generated from Scenario 1 compared with those created manually by using capture/replay tools. On one hand, since a human tester is involved in creating test cases with capture/replay tools, the tester can make intelligent choices, resulting in effective test cases. On the other hand, an automated technique has the advantage of being comprehensive, in that requirements such as 'cover every event with least five test cases' can be implemented into the ESES algorithms. The experiment will compare the effectiveness of these two approaches while measuring effort and time. It will also explore whether the types of faults detected by the techniques differed in fundamental ways; hence, not only will the number of faults found be evaluated but also the classes of faults found. Keeping this goal in mind, classes of known artificial faults will be seeded into the subject applications. The faults that were found in Scenario 1 were first fixed.

More specifically, the following steps were performed in this scenario.

(1) Generate 1000 test cases for each application using a capture/replay tool. The tool also allows the tester to manually specify test-oracle information during the capture phase.

DOI: 10.1002/stvr

(2) Seed artificial faults into the subject applications to create fault-seeded versions.
(3) Execute all 1000 test cases on all of the fault-seeded versions.
(4) From the previous scenario, take the 1000 test cases and oracle information and execute it on all fault-seeded versions.

The cost of generating the test cases and their fault-detection effectiveness will now be evaluated. As this is a mock experiment, factors of experiment design, threats to validity, and analysis of results will be ignored; they must be considered carefully in a real experiment.

Each of these steps is now described in more detail.

6.3.1. Manual test-case and test-oracle generation

A capture/replay tool was implemented. Four testers, one for each subject application, generated the 1000 test cases each. Two processes were used for test-case generation, both of which are used by real testers in the industry for GUI testing. First the testers were asked to spend 1 hour becoming familiar with using the subject applications. Each tester was then given 100 tasks, i.e. activities that they could complete by using the application. They had to devise five different ways to complete each task; the capture tool recorded the user interaction as a test case. The tasks were chosen carefully so that they covered most of the functionality of the applications; 500 test cases were obtained in this way. Then the testers were asked to interact with the software *without* using tasks, but were asked to cover most of the applications' windows; 500 additional test cases were generated in this way. In all, 1000 test cases were obtained per application. Each tester took 10–14 days to complete this process.

The capture/replay tool also has mechanisms to record the properties/values of specified widgets during the capture phase. There was no restriction on how much detail a tester could include in the oracle information.

Figure 4 shows the time it took to generate the test cases as distributions. The boxplots provide a concise display of each distribution. The black square inside each box marks the median value. The edges of the box mark the first and third quartiles. The whiskers extend from the quartiles and cover the entire distribution. The plots show that even though the time varied between applications, most of the test cases were generated in 5 minutes.

6.3.2. Fault seeding

Fault seeding is a common technique to introduce known faults into programs. It has several applications; in this scenario, it is used to create faulty versions of code. The goal is to execute test cases on the faulty versions and study the properties of the test cases that are successful at detecting the faults. Care is taken so that the artificially seeded faults are similar to faults that occur in programs due to mistakes made by developers [39,40]. A history-based approach was adopted to seed GUI faults, i.e. 'real' GUI faults in real applications were observed and reused. During the development of TerpOffice, a bug-tracking tool called *Bugzilla*[*] was used by the developers to report and track faults in TerpOffice version 1.0 while they were working to extend its functionality and developing version 2.0.

[*]See http://bugs.cs.umd.edu.

DOI: 10.1002/stvr

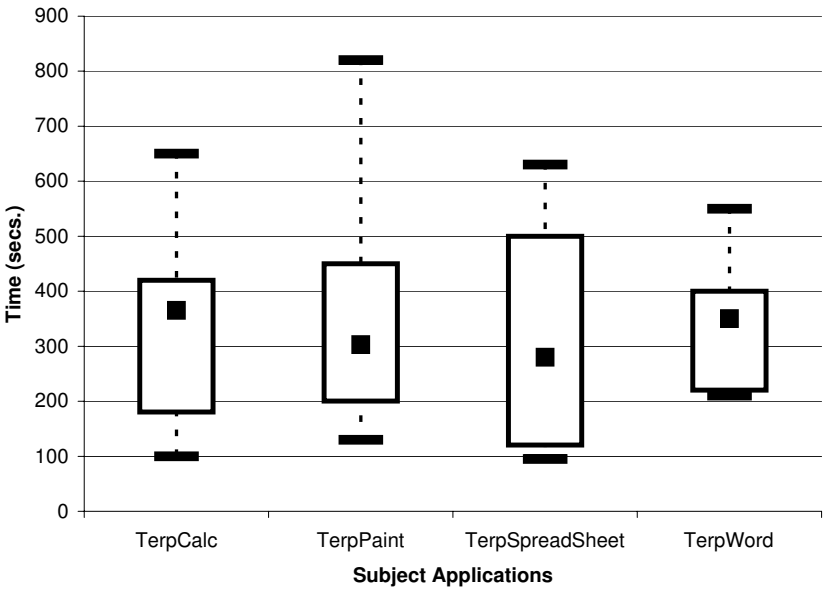

Figure 4. Time taken to record each test case.

The reported faults are an excellent representation of faults that are introduced by developers during implementation. A total of 200 faulty versions were created for each software. Only one fault was introduced in each version. This model is useful to avoid fault-interaction, which can be a thorny problem in these types of study and also simplifies the computation of the number of faults detected; they can simply be counted.

6.3.3. Test-case execution

All 2000 test cases were executed on all fault-seeded versions of each application. The event-flow model and tools enabled the quick setup of this large experiment involving $2000 \times 200 \times 4 =$ 1.6 million test runs. The replayer was the same as that used in Scenario 1. The test cases produced by the capture/replay tool and those produced by the test-case generation algorithm are compatible with the replayer. An average of 10 test cases could be executed per minute. In all, the test runs took almost 28 days on four computers.

6.4. Observations from Scenario 2

The column graph in Figure 5 summarizes the results. The x-axis shows the subject applications and the y-axis shows the total number of faults detected. The figure shows that both techniques detected a large number of faults although the automated approach did slightly better.

DOI: 10.1002/stvr

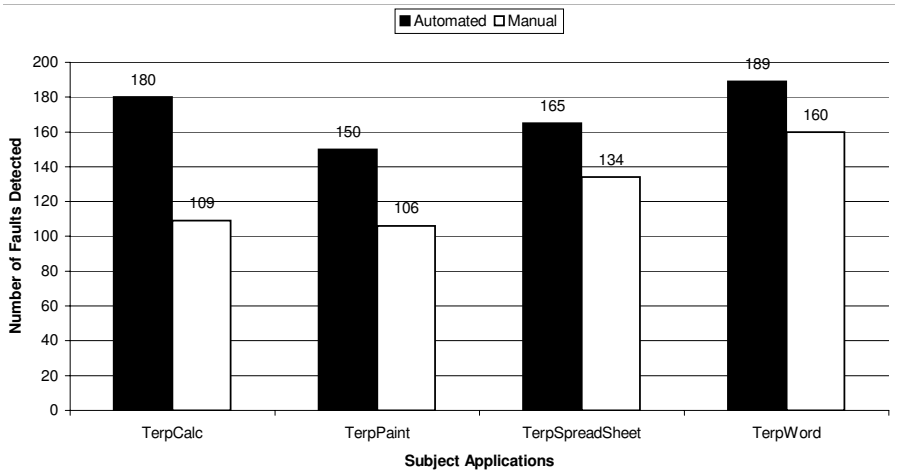

Figure 5. Number of faults detected.

Even though this was a mock experiment, this scenario showed that the manual capture/replay process was extremely resource intensive: the testers spent almost two weeks per application to obtain 1000 test cases. If an additional 1000 test cases were needed, they would need to spend another two weeks. This is in sharp contrast to the model-based approach that amortizes the cost of model creation over subsequent test-generation cycles. The test cases obtained from the automated approach were better at detecting faults than those obtained manually. This result is greatly affected by the experience of the testers, the nature and types of seeded faults, and the ESESs.

Most importantly, this scenario demonstrated that the event-flow model and its associated tools allowed the quick setup and execution of a large experiment. The interested reader is referred to additional, much more comprehensive experiments that have leveraged the same tools [12,38].

7. CONCLUSIONS

With the growing importance of event-driven software such as GUIs and Web applications, and the ever-increasing demands on their quality, new techniques and models need to be developed to test this class of software. This paper presented a model (called the event-flow model) of GUIs based on their event interactions. Combined with customized ESESs, this event-flow model may be used for various aspects of GUI testing. The usefulness of this model was demonstrated via two scenarios. These scenarios showed that the event-flow model can be used to quickly generate a large number of GUI test cases that are effective at detecting GUI faults. The model also promotes reusability because once created, it can be quickly redeployed to generate additional test cases for the same or a modified version of the GUI.

DOI: 10.1002/stvr

The event-flow model is scalable, i.e. the size of the model grows linearly with the number of events in the GUI. The biggest challenge that this model currently poses is controlling the size of the space of all possible event interactions. The number of event sequences that can be executed on the GUI grows exponentially with length. There has been some success with reducing the space via abstractions such as that described in this paper, i.e. using modal dialogues. Additional abstractions will help to further reduce the space. New ESESs are being designed that traverse the event space in more intelligent ways.

The event-flow model has already been extended. For GUIs, vertices and edges in the event-flow graphs and integration tree have been annotated with weights that record user profiles in a compact manner; vertices have been annotated with probability tables to create a Bayesian model for test-case generation.

All of the tools described in this paper have been packaged into a software called *GUITAR*; it is available for download at http://guitar.cs.umd.edu. GUITAR has been downloaded more than 10 000 times since it was first made available in 2002; several practitioners in industry provide continuous feedback that drives bug-fixes and enhancements.

ACKNOWLEDGEMENTS

This work was partially supported by the U.S. National Science Foundation under NSF grant CCF-0447864 and the Office of Naval Research grant N00014-05-1-0421. Qing Xie, Ishan Banerjee, Adithya Nagarajan, and Bin Gan participated in the execution of the scenarios. Anonymous reviewers provided valuable feedback that reshaped much of the empirical studies and presentation of results. Professor Lee White proposed the classification of Type 1 and Type 2 failures.

REFERENCES

1. Mahajan R, Shneiderman B. Visual and textual consistency checking tools for graphical user interfaces. *Technical Report CS-TR-3639*, University of Maryland, College Park, MD, May 1996.
2. Myers BA. User interface software tools. *ACM Transactions on Computer–Human Interaction* 1995; **2**(1):64–103.
3. Myers BA, Olsen DR Jr. User interface tools. *Proceedings of the ACM CHI'94 Conference on Human Factors in Computing Systems*, 1994, vol. 2 of Tutorials. ACM Press: New York, 1994; 421–422.
4. Myers BA, Olsen DR Jr, Bonar JG. User interface tools. *Proceedings of ACM INTERCHI'93 Conference on Human Factors in Computing Systems—Adjunct Proceedings*, 1993, Tutorials. ACM Press: New York, 1993; 239.
5. Myers BA. *State of the Art in User Interface Software Tools*, vol. 4. Ablex Publishing: Westport, CT, 1993; 110–150.
6. Memon AM. GUI testing: Pitfalls and process. *IEEE Computer* 2002; **35**(8):90–91.
7. Myers BA, Hollan JD, Cruz IF. Strategic directions in human–computer interaction. *ACM Computing Surveys* 1996; **28**(4):794–809.
8. White L, Almezen H. Generating test cases for GUI responsibilities using complete interaction sequences. *Proceedings of the International Symposium on Software Reliability Engineering*, 8–11 October 2000. IEEE Computer Society Press: Piscataway, NJ, 2000; 110–121.
9. White L, Almezen H, Alzeidi N. User-based testing of GUI sequences and their interaction. *Proceedings of the International Symposium on Software Reliability Engineering*, 8–11 November 2001. IEEE Computer Society Press: Piscataway, NJ, 2001; 54–63.
10. Memon AM, Pollack ME, Soffa ML. Using a goal-driven approach to generate test cases for GUIs. *Proceedings of the 21st International Conference on Software Engineering*, May 1999. ACM Press: New York, 1999; 257–266.
11. Memon AM, Pollack ME, Soffa ML. Hierarchical GUI test-case generation using automated planning. *IEEE Transactions on Software Engineering* 2001; **27**(2):144–155.
12. Memon AM, Xie Q. Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software. *IEEE Transactions on Software Engineering* 2005; **31**(10):884–896.
13. Memon A, Nagarajan A, Xie Q. Automating regression testing for evolving GUI software. *Journal of Software Maintenance and Evolution: Research and Practice* 2005; **17**(1):27–64.
14. Memon AM, Pollack ME, Soffa ML. Automated test oracles for GUIs. *Proceedings of the ACM SIGSOFT 8th International Symposium on the Foundations of Software Engineering (FSE-8)*, 8–10 November 2000. ACM Press: New York, 2000; 30–39.

DOI: 10.1002/stvr

15. White L. Regression testing of GUI event interactions. *Proceedings of the International Conference on Software Maintenance*, 4–8 November 1996. IEEE Computer Society Press: Piscataway, NJ, 1996; 350–358.
16. Memon AM, Soffa ML. Regression testing of GUIs. *Proceedings of the 9th European Software Engineering Conference (ESEC) and 11th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-11)*, September 2003. ACM Press: New York, 2003; 118–127.
17. Memon AM, Soffa ML, Pollack ME. Coverage criteria for GUI testing. *Proceedings of the 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9)*, September 2001. ACM Press: New York, 2001; 256–267.
18. Allen FE. Control flow analysis. *Proceedings of a Symposium on Compiler Optimization*. ACM Press: New York, 1970; 1–19.
19. Rosen BK. Data flow analysis for procedural languages. *Journal of the ACM* 1979; **26**(2):322–344.
20. Clarke JM. Automated test generation from a behavioural model. *Proceedings of Pacific Northwest Software Quality Conference*, May 1998. PNSQC: Portland, OR, 1998.
21. Chow TS. Testing software design modelled by finite-state machines. *IEEE Transactions on Software Engineering* 1978; **4**(3):178–187.
22. Esmelioglu S, Apfelbaum L. Automated test generation, execution, and reporting. *Proceedings of Pacific Northwest Software Quality Conference*, October 1997. IEEE Press: Piscataway, NJ, 1997.
23. Bernhard PJ. A reduced test suite for protocol conformance testing. *ACM Transactions on Software Engineering and Methodology* 1994; **3**(3):201–220.
24. Shehady RK, Siewiorek DP. A method to automate user interface testing using variable finite state machines. *Proceedings of the 27th Annual International Symposium on Fault-Tolerant Computing (FTCS'97)*, June 1997. IEEE Computer Society Press: Piscataway, NJ, 1997; 80–88.
25. Kasik DJ, George HG. Toward automatic generation of novice user test scripts. *Proceedings of the Conference on Human Factors in Computing Systems: Common Ground*, New York, 13–18 April 1996. ACM Press: New York, 1996; 244–251.
26. Fogel LJ, Owens AJ, Walsh MJ. Artificial intelligence through a simulation of evolution. *Biophysics and Cybernetic Systems: Proceedings of the 2nd Cybernetic Sciences Symposium*, Maxfield M, Callahan A, Fogel LJ (eds.). Spartan Books: Washington, DC, 1965; 131–155.
27. Fogel LJ, Owens AJ, Walsh MJ. *Artificial Intelligence Through Simulated Evolution*. Wiley: New York, 1966.
28. White L, Almezen H, Sastry S. Firewall regression testing of GUI sequences and their interactions. *Proceedings of the International Conference on Software Maintenance*, 22–26 September 2003. IEEE Computer Society Press: Piscataway, NJ, 2003; 398–409.
29. Memon AM. *Advances in GUI Testing* (*Advances in Computers*, vol. 57), Zelkowitz MV (ed.). Academic Press: New York, 2003; 149–201.
30. Memon AM, Xie Q. Empirical evaluation of the fault-detection effectiveness of smoke regression test cases for GUI-based software. *Proceedings of the International Conference on Software Maintenance 2004 (ICSM'04)*, September 2004. IEEE Computer Society Press: Piscataway, NJ, 2004; 8–17.
31. Nagarajan A, Memon AM. Refactoring using event-based profiling. *Proceedings of the 1st International Workshop on Refactoring: Achievements, Challenges, Effects (REFACE)*, November 2003. IEEE Computer Society Press: Piscataway, NJ, 2003.
32. Pednault EPD. Adl: Exploring the middle ground between strips and the situation calculus. *Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning*, San Francisco, CA, 1989. Morgan Kaufmann: San Francisco, CA, 1989; 324–332.
33. Weld DS. An introduction to least commitment planning. *AI Magazine* 1994; **15**(4):27–61.
34. Weld DS. Recent advances in AI planning. *AI Magazine* 1999; **20**(1):55–64.
35. Memon AM, Banerjee I, Nagarajan A. GUI ripping: Reverse engineering of graphical user interfaces for testing. *Proceedings of the 10th Working Conference on Reverse Engineering*, November 2003. IEEE Computer Society Press: Piscataway, NJ, 2003; 260–269.
36. Dwyer MB, Carr V, Hines L. Model checking graphical user interfaces using abstractions. *Proceedings of the European Software Engineering Conference (ESEC) and ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, 1997. ACM Press: New York, 1997; 244–261.
37. Memon AM, Xie Q. Using transient/persistent errors to develop automated test oracles for event-driven software. *Proceedings of the International Conference on Automated Software Engineering 2004 (ASE'04)*, September 2004. IEEE Computer Society Press: Piscataway, NJ, 2004; 186–195.
38. Xie Q, Memon AM. Designing and comparing automated test oracles for GUI-based software applications. *ACM Transactions on Software Testing and Methodology* 2007; to appear.
39. Offutt AJ, Hayes JH. A semantic model of program faults. *Proceedings of the International Symposium on Software Testing and Analysis*. ACM Press: New York, 1996; 195–200.
40. Harrold MJ, Offutt AJ, Tewary K. An approach to fault modelling and fault seeding using the program dependence graph. *Journal of Systems and Software* 1997; **36**(3):273–296.

DOI: 10.1002/stvr