Advances in Automated Model-Based System Testing of Software Applications with a GUI Front-End

Bao N. Nguyen and Atif M. Memon
Department of Computer Science,
University of Maryland,
College Park, MD 20742.
{baonn,atif}@cs.umd.edu

Abstract

Despite the ubiquity of software applications that employ a Graphical-User Interface (GUI) front-end, functional system testing of these applications has remained, until recently, an understudied research area. During “GUI testing,” test cases, modeled as sequences of user input events, are created and executed on the software by exercising the GUI’s widgets. Because each possible sequence of user events may potentially be a test case and because today’s GUIs offer enormous flexibility to end users, in principle, GUI testing requires a prohibitively large number of test cases. Any practical test case generation technique must sample the vast GUI input space. Existing techniques are largely manual, and hence extremely resource intensive. Several new model-based automated techniques have been developed in the past decade. All these techniques develop, either manually or automatically, a model of the GUI and employ it to generate test cases. This chapter presents the first detailed taxonomy of these techniques. A small GUI application is used as a running example to demonstrate each technique and illustrate its relative strengths and weaknesses.

Contents

1 Introduction 4
2 Running Example of GUI Application Under Test 7
3 Test Case Generation Techniques 8
   3.1 State Machines ................................................. 8
      3.1.1 Finite State Machines ................................... 9
      3.1.2 Variable Finite State Machines .......................... 11
      3.1.3 Complete Interaction Sequences .......................... 13
      3.1.4 Off-nominal Finite State Machines ....................... 17
   3.2 Workflows ....................................................... 19
3.2.1 Event Flow Graph .................................................. 19
3.2.2 Event Interaction Graph .......................................... 21
3.2.3 Event Semantic Interaction Graph ......................... 23
3.2.4 Off-nominal Event Graph ....................................... 29
3.3 Pre-Post-Condition Models ...................................... 31
3.4 Event Sequence-Based Models .................................. 34
3.5 Probabilistic Models .................................................. 35
3.6 Combinatorial Interaction Models ......................... 39
  3.6.1 Latin Squares .................................................. 39
  3.6.2 Covering Arrays ............................................... 40
3.7 Hierarchical Models ................................................... 41
  3.7.1 Keyword-driven Models .................................... 42
  3.7.2 Hierarchical Finite State Machines ................... 45
  3.7.3 UML Diagram-based ......................................... 47

4 Conclusions ................................................................. 52
Biography

Bao N. Nguyen is a Ph.D. student at the Department of Computer Science, the University of Maryland. He received his B.S. degree in Computer Science with first class honors from Vietnam National University, Hanoi, in 2005. Before that, he was awarded the ‘The Bridge over Asian Countries’ scholarship to study in Japan for one year. He received the Vietnam Education Foundation fellowship for his Ph.D. research. His research interests include software engineering, software testing and reverse engineering.

Atif M Memon is an Associate Professor at the Department of Computer Science, University of Maryland. His research interests include program testing, software engineering, artificial intelligence, plan generation, reverse engineering, and program structures. He is the inventor of the GUITAR system (http://guitar.sourceforge.net/) for automated model-based GUI testing. He is the founder of the International Workshop on TESTing Techniques & Experimentation Benchmarks for Event-Driven Software (TESTBEDS). He serves on various editorial boards, including that of the Journal of Software Testing, Verification, and Reliability. He has served on numerous National Science Foundation panels and program committees, including the International Conference on Software Engineering (ICSE), International Symposium on the Foundations of Software Engineering (FSE), International Conference on Software Testing Verification and Validation (ICST), Web Engineering Track of The International World Wide Web Conference (WWW), the Working Conference on Reverse Engineering (WCRE), International Conference on Automated Software Engineering (ASE), and the International Conference on Software Maintenance (ICSM). 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.
1 Introduction

As computers play an increasingly important role aiding end-users, researchers, and businesses in today’s inter-networked world, the class of software that has a graphical user interface (GUI) front-end has become ubiquitous [34, 22, 38]. A GUI takes events (mouse clicks, selections, typing in text-fields) as input from users, and then changes the state of its widgets. GUIs have become popular because of the advantages this “event-handler architecture” offers to both developers and users [19, 46, 13]. From the developer’s point of view, the event handlers may be created and maintained fairly independently; hence, complex system may be built using these loosely coupled pieces of code. From the user’s point of view, GUIs offer many degrees of usage freedom, i.e., users may choose to perform a given task by inputting GUI events in many different ways in terms of their type, number and execution order.

Quality Assurance (QA) is becoming increasingly important for GUIs as their functional correctness may affect the quality of the entire system in which the GUI operates [7]. Software testing is a popular QA technique employed during software development and deployment to help improve its quality [24, 29]. During software testing, test cases are created and executed on the software. One way to test a GUI is to execute each event individually and observe its outcome, thereby testing each event handler in isolation [20, 31]. However, the execution outcome of an event handler may depend on its internal state, the state of other entities (objects, event handlers) and the external environment. Its execution may lead to a change in its own state or that of other entities. Moreover, the outcome of an event’s execution may vary based on the sequence of preceding events seen thus far. Consequently, in GUI testing, each event needs to be tested in different states. GUI testing therefore involves generating and executing sequences of events [33, 46].

The event-driven nature of GUIs creates several challenges for testing. One important challenge stems from the enormous space of possible event interactions with the GUI [41, 9]. Because each possible event sequence may potentially be a test case, GUI testing, in principle, may require a prohibitively large number of test cases. Practical GUI testing techniques
attempt to sample the vast input space of all possible sequences with the goal of detecting
faults; for effective testing, it is important to sample this space carefully [45].

In practice, GUI testing is done in two ways. First, testers employ unit testing tools
[21, 41] such as JFCUnit [2], Abbot [1], Pounder [5] and Jemmy Module [3] to manually
create unit test cases for GUIs. A unit test case consists of method calls to an instance
of the class under test. Assertions are inserted in the test cases to determine whether the
classes/methods executed correctly. The test cases are automatically executed on the GUI
under test. Assertion violations are reported as failures. The parts of the GUI state space
explored depends largely on the nature of the test cases. Because manual coding of test cases
can be tedious, an alternative, which is the second popular technique, “captures” sequences
of events that testers perform manually on the GUI. Hence this technique treats a test case
as a sequence of input events. These test cases can be “replayed” automatically on the GUI.
Tools used for this “capture” and “replay” are called capture/replay tools [26, 4]. As was the
case with unit testing, the test case creation is manual (in terms of the event sequence) and
the tools facilitate only the execution of test cases. The “goodness” of the test cases depends
on the tester’s ability to obtain fault-exposing sequences [9, 23].

The last decade has seen some advances in automated model-based GUI testing tech-
niques. In this chapter, we provide a taxonomy of these techniques, which are shown in
Table 1. As the table shows, we discuss 16 techniques. All these techniques require the
creation of a model of the software or its GUI, and algorithms to use the model to generate
test cases. The techniques of interest to us employ 6 distinct models, shown in Column
1 of Table 1; the “hierarchical” model uses a combination of these models organized in a
hierarchy.

There are two important aspects of each technique that we discuss. First is the model
that it employs. In some cases, the models are created manually; in others, they are derived
in an automated manner. The second important aspect is the test-case generation approach,
which, for some techniques is manual; but for most is automated. Figure 1 shows the set
of techniques discussed in this chapter, partitioned along two dimensions: (model creation {manual, automated}, test generation {manual, automated}).

<table>
<thead>
<tr>
<th>Model</th>
<th>Technique</th>
<th>Abbreviation</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>State machine</td>
<td>Finite State Machine</td>
<td>FSM</td>
<td>3.1.1</td>
</tr>
<tr>
<td></td>
<td>Variable Finite State Machine</td>
<td>VFSM</td>
<td>3.1.2</td>
</tr>
<tr>
<td></td>
<td>Complete Interaction Sequence</td>
<td>CIS</td>
<td>3.1.3</td>
</tr>
<tr>
<td></td>
<td>Faulty Complete Interaction</td>
<td>FCIS</td>
<td>3.1.4</td>
</tr>
<tr>
<td>Workflow</td>
<td>Event Flow Graph</td>
<td>EFG</td>
<td>3.2.1</td>
</tr>
<tr>
<td></td>
<td>Event Interaction Graph</td>
<td>EIG</td>
<td>3.2.2</td>
</tr>
<tr>
<td></td>
<td>Feedback based</td>
<td>ESIG</td>
<td>3.2.3</td>
</tr>
<tr>
<td></td>
<td>Faulty Event Sequence Graph</td>
<td>FESG</td>
<td>3.2.4</td>
</tr>
<tr>
<td>Pre- Post-condition</td>
<td>AI Planning</td>
<td>AI</td>
<td>3.3</td>
</tr>
<tr>
<td>Event sequence</td>
<td>Genetic Models</td>
<td>GA</td>
<td>3.4</td>
</tr>
<tr>
<td>Probabilistic</td>
<td>Probabilistic Event Flow Graph</td>
<td>PEFG</td>
<td>3.5</td>
</tr>
<tr>
<td>Combinatorial</td>
<td>Latin Squares</td>
<td>LS</td>
<td>3.6.1</td>
</tr>
<tr>
<td></td>
<td>Coverage Arrays</td>
<td>CA</td>
<td>3.6.2</td>
</tr>
<tr>
<td>Hierarchical</td>
<td>Keyword-driven Model</td>
<td>KW</td>
<td>3.7.1</td>
</tr>
<tr>
<td></td>
<td>Hierarchical Finite State Machines</td>
<td>HFSM</td>
<td>3.7.2</td>
</tr>
<tr>
<td></td>
<td>UML-Diagram Based</td>
<td>UML</td>
<td>3.7.3</td>
</tr>
</tbody>
</table>

Table 1: GUI Testing Techniques Discussed in this Chapter.

The remainder of this chapter presents these techniques. But first, we present a small GUI application, that we use as a running example, to illustrate the important aspects of each technique, and its relative strengths and weaknesses.
2 Running Example of GUI Application Under Test

The simple running example application called “Radio Button Demo” is seen in Figure 2. The GUI contains 9 widgets labeled $w_0$ through $w_8$. A user can perform events on almost all the widgets (there is no event available on $w_4$). Table 2 shows the events associated with each widget. We note that in this simple example, each widget has at most one associated event. In a more complex GUI, a widget may have multiple associated events.

![Diagram of the Radio Button Demo Application]

Figure 2: The Radio Button Demo Application.

<table>
<thead>
<tr>
<th>Widget</th>
<th>Event name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w_1$</td>
<td>circle</td>
</tr>
<tr>
<td>$w_2$</td>
<td>square</td>
</tr>
<tr>
<td>$w_3$</td>
<td>create</td>
</tr>
<tr>
<td>$w_5$</td>
<td>reset</td>
</tr>
<tr>
<td>$w_0$</td>
<td>exit</td>
</tr>
<tr>
<td>$w_6$</td>
<td>(un)check</td>
</tr>
<tr>
<td>$w_7$</td>
<td>yes</td>
</tr>
<tr>
<td>$w_8$</td>
<td>no</td>
</tr>
</tbody>
</table>

Table 2: Events available on each widget.

The application’s functionality is very straightforward – the initial state has Circle (corresponding to $w_1$) selected, the Rendered Shape area (widget $w_4$) is empty and the
Reset button is disabled. Events are used to change the state of the GUI. Event *circle* sets the radio button setting to circle; if there is already a square in the Rendered Shape area, then the shape is immediately changed to a circle. Event *square* is similar to *circle*, except that it changes the shape to a square. Event *create* creates a shape in the Rendered Shape area according to current settings of $w_1$ and $w_2$. Event *reset* resets the entire software to its initial state. This event is only available when there is an existing shape. Event *exit* opens a modal “Exit Confirmation” window that contains widgets $w_6$, $w_7$, and $w_8$. This window blocks all widgets in the main window when open. Event *(un)check* changes the status of the check-box $w_6$ (originally unchecked) so that when it is checked the exiting time will be logged to a file before the application is terminated. Event *no* closes the window and moves focus back to the main window; and event *yes* closes the entire application.

The GUI of this application is simple, yet quite flexible. The numbers of 1-, 2-, 3-, 4-, and 5-way unique event sequences (and hence possible test cases) that may be executed in the initial state of the GUI are 4 (remember that the Exit Confirmation window is initially closed and $w_5$ is disabled), 17, 66, 253, and 798 respectively.

## 3 Test Case Generation Techniques

This section presents an overview of all the techniques listed in Table 1. The techniques are classified according to the underlying model used.

### 3.1 State Machines

Because GUIs are composed of objects (i.e., the widgets) that maintain state, in terms of widget-properties (e.g., Enabled, Caption, Width) and their values (e.g., TRUE, “Cancel”, 20), many researchers have found it natural to model GUIs using state machines [20, 37, 40, 9]. For example, the GUI of Figure 2 starts in an “initial state” in which, among other widgets, widget $w_3$ is not selected and $w_5$ is disabled. If one were to model the state of the
GUI as a set of triples \((\text{widget}, \text{property}, \text{value})\), the initial state could be represented as \(\{\ldots, (w_3, \text{Selected}, \text{FALSE}), (w_3, \text{Enabled}, \text{TRUE}), (w_5, \text{Enabled}, \text{FALSE}), \ldots\}\). As can be imagined, depending on how one models the state, such machines can get extremely large for non-trivial GUIs. In this section, we present several techniques that researchers have employed to control this state space explosion. Esmelioglu et al. [20] use constraints, Shehady et al. [37] use global variables, White et al. [40] focus on a part of the state machine, and Belli et al. [9] develop off-nominal test cases. We present these techniques next.

### 3.1.1 Finite State Machines

In this section, we present details of the approach taken by Esmelioglu et al. [20], who model the GUI as a finite state machine (FSM). Formally, a FSM can be represented as a quintuple \(\text{FSM} = (S, I, O, T, \Phi)\), where \(S\) is the finite set of GUI states, \(I\) is the set of inputs, i.e., events that may be performed on the GUI, \(O\) is the finite set of outputs, \(T\) is the transition function \(S \times I \rightarrow S\) that specifies the next state as a function of the current state and input event, \(\Phi\) is the output function \(S \times I \rightarrow O\) that specifies the resulting output from a transition.

For GUI testing, a tester is free to select certain aspects of the software to model in the state. For example, we choose to represent the state of the GUI using 4 of its elements: (1) \(\log\), which is 1 if \(w_6\) is checked, 0 otherwise; (2) \(\text{exitWinOpen}\), which is 1 if the Exit Confirmation window is open, 0 otherwise; (3) \(\text{created}\) which is 1 if a shape is created, 0 otherwise; (4) \(\text{shape}\) which is either Circle or Square.

We can then represent the state of the GUI using a length 4 vector consisting of the above 4 elements in the order specified above. For example, state \(S_{000C}\) is the initial state in which \(w_6\) is unchecked, the exit confirmation window is closed, no shape is created at \(w_4\), and the shape radio button for circle is set. Similarly, \(S_{111S}\) is the state in which \(w_6\) is checked, the exit confirmation window is open, a shape is visible at \(w_4\), and the shape radio button for square is set.
We use the above definition of GUI state to create an FSM. Figure 3 shows the FSM of the GUI of Figure 2. Nodes in the graph represent states and edges represent transitions; there are two special states (shaded nodes) in the FSM: the initial state right after the software starts ($S_{000C}$), and the terminal state when the software has been terminated ($S_t$). Some of the state transitions are as follows: If the $\text{create}$ button has never been clicked, then the user can transit between $S_{000x}$ states by selecting different radio button options ($x$...
represents any value of the corresponding state element, in this case $x$ is either $C$ or $S$). Once Create has been clicked, the GUI transits to a new state where the third state element turns from 0 is 1 (i.e., a new shape has been created). The user can transit back and forth between $S_{x0x}C$ and $S_{x0x}S$ by selecting different radio button options ($x$ represents any value of the corresponding state element). However, the user cannot do the same for the pair $(S_{01x}C, S_{01x}S)$ or $S_{11x}C$ states because the Exit Confirmation window blocks all widgets in the main window.

Once the FSM has been created, test case generation from an FSM is very intuitive. The test designer may start at the initial state, traverse edges of the FSM as desired and record the transitions as events. For example, in Figure 3, a test case could be: $\langle$ square, circle, create, exit, (un)check, yes $\rangle$ which takes the software through states $S_{000}S$, $S_{000}C$, $S_{001}C$, $S_{011}C$, $S_{111}C$, and $S_t$.

Although FSMs are easy to create, they suffer from some major problems. First, they do not scale for large GUIs. Moreover, the states may not have any relationship to the structure of the GUI. Hence they can be difficult to maintain. A new model called variable finite state machines (VFSMs), developed by Shehady et al. [37], presented next, attempts to rectify some of these problems.

### 3.1.2 Variable Finite State Machines

Shehady et al. [37] use Variable Finite State Machines (VFSMs) for testing GUIs. The key difference between VFSMs and FSMs is that VFSMs allow a number of global variables, each of which takes values from a finite domain. The values of the variables are used to compute the next state and the output in response to an input. Transitions may also modify the values of these variables. In principle, the space of GUIs that can be modeled using VFSMs is the same as those that can be modeled using FSMs.

Formally, a VFSM is represented as a 7-tuple $VFSM = (S, I, O, T, \Phi, V, \zeta)$, where $S$, $I$, $O$ are similar to their counterparts in FSMs, $V = \{V_1, V_2, V_3, \ldots, V_n\}$ (each $V_i$ is the set of
values that the $i$th variable may assume) and $n$ is the total number of variables in the VFSM. Let $D = S \times I \times V_1 \times V_2 \times \ldots \times V_n$ and $D_T \subseteq D$; $T$ is the transition function $D_T \rightarrow S$ and $\Phi$ is a function $D_T \rightarrow O$. Hence the current state of each of the variables affects both the next state and the output of the VFSM. $\zeta$ is the set of variable transition functions. At each transition, $\zeta$ is used to determine whether any of the variables’ values have been modified. Each variable has an initial state at startup.

![Figure 4: Variable Finite State Machine.](image)

Figure 4 shows an VFSM of the Radio Button Demo’s GUI. The VFSM is much smaller than the corresponding FSM (Figure 3) because the states have been simplified. Each state is simply represented by a length 3 vector, i.e., that specifies whether $\text{log}$ needs to be maintained, the Exit Confirmation window is open, and the type of shape that has been selected.

The states have been simplified because the element $\text{created}$ has been removed from the state. This information is now maintained in a variable $V$ that can take values 0 and 1. Edges of the VFSM are annotated with predicates (shown in parenthesis placed before the edge label) and assignments to the variables (shown in square brackets placed after the
edge label). Initially, $V$ is set to 0. Transitions are taken depending on the outcome of the predicates. For example, the reset transition is taken from $S_{00C}$ only if $V == 1$; once taken, it changes $V$ to 0. Similarly, create changes $V$ to 1.

The VFSM created is much more concise (it has 9 states) than the original FSM in Figure 3 (which has 17 states). This is because several states in the FSM are grouped and represented by a single state in the VFSM. VFSMs can be converted into their equivalent FSMs for test case generation. The key idea is to fold the information of $V$ and $\zeta$ into $S$ and $T$. Given a VFSM’s $S$ and $V = \{V_1, V_2, \ldots, V_n\}$, the new FSM’s set of states $S_{eq}$ is obtained as $S_{eq} = \{S_i | S_i \in S \times V_1 \times V_2 \times V_3 \times \ldots \times V_n\}$, i.e., this creates a set of states that combines the information of the states and the variables into one state. Similarly, the new FSM’s transition function $T_{eq}$ : $S_{eq} \times I \rightarrow S_{eq}$ may be created by combining the $T$ and $\zeta$ functions of the VFSM. Since the range of $T$ is $S$ and the range of $\zeta$ is $V = \{V_1, V_2, \ldots, V_n\}$, $S_{eq}$ is the Cartesian product of the two ranges; also $T$ and $S$ have the same domain.

3.1.3 Complete Interaction Sequences

Another approach to restrict the state space of a state machine is by employing software usage information. The method proposed by White et al. [40] solves the FSM’s state explosion problem by focusing on a subset of interactions performed on the GUI. They key idea is to identify responsibilities for a GUI; a responsibility is a GUI activity that involves one or more GUI objects and has an observable effect on the surrounding environment of the GUI, which includes memory, peripheral devices, underlying software, and application software. For each responsibility, a complete interaction sequence (CIS), which is a sequence of GUI objects and selections that will invoke the given responsibility, is identified. Parts of the CIS are then used for testing the GUI.

The GUI testing steps for CIS are as follows.

1. Manually identify responsibilities in the GUI.

2. For each responsibility, identify its corresponding CIS.
3. Create an FSM for each CIS.

4. Apply transformations to the FSM to obtain a \textit{reduced FSM}. These transformations include the following.

   (a) Abstracting strongly connected components into a \textit{superstate}.

   (b) Merging CIS states that have structural symmetry.

5. Use the reduced FSM to test the CIS for correctness.

   The two abstractions mentioned above (Steps 4a and 4b) are interesting from a modeling point of view. They are described in more detail next.

\textbf{Definition}: A part of a FSM, called a \textit{subFSM}, is a \textit{strongly connected component} if for every pair \((S_1, S_2), S_1, S_2 \in S\), there exists a directed path from \(S_1\) to \(S_2\). Each such component is then replaced by a \textit{superstate} and tested in isolation.

A subFSM has structural symmetry if the following conditions hold.

1. it contains states \(S_1\) and \(S_2\) such that \(S_1\) has one incoming transition, \(S_2\) has one outgoing transition, and a number of paths reach \(S_2\) from \(S_1\);

2. for each path in the subFSM, context (the path taken to get to \(S_1\) from outside the subFSM) has no effect on the states/transitions or output;

3. no transition or state encountered after \(S_2\), is affected by paths taken inside the sub-FSM.

Such a subFSM can be reduced into a superstate and tested in isolation.

Given a GUI, the test designer first reduces the FSM after applying the above transformations, thereby reducing the total number of states in the FSM. This results in smaller number of paths in the FSM, hence reducing the number of test cases. Without any loss
of generality, each FSM is assumed to have a distinct start state and distinct terminating state.

As was the case before, a test is a path through the FSM. The test designer creates two types of tests: design tests that assume that the FSM is a faithful representation of the GUI’s specifications, and implementation tests that for each CIS, consider the possibility that potential transitions not described in the design may occur in the implementation.

For design tests, the test designer creates sufficient number of tests starting at the initial state and ending at the termination state so that the following conditions hold:

- all distinct paths in the reduced FSM are executed; each time a path enters a superstate corresponding to a component, an appropriate test path of the component is inserted into the test case at that point,

- all the design subtests of each component are included in at least one test, which may require additional tests of the reduced FSM to satisfy this constraint.

The key idea of conducting implementation testing is to check all GUI events in the CIS to determine whether they invoke any new transitions in the reduced FSM. To implement test the reduced FSM, the test designer must construct sufficient test sequences at the initial state and stopping at the terminal state so that the following conditions hold:

- all the paths of the reduced FSM are executed, and

- all the implementation tests for each remaining component are included at least once.

By using the CIS concept, the test designer can test a GUI from various perspectives, each defined by the CIS. These CIS can also be maintained in a library to be reused across various GUIs.

For example, in the Radio Button Demo application, the tester may design a “create a new shape” responsibility that involves 4 objects $w_1$, $w_2$, $w_3$, and $w_5$ (assuming that the Rendered Shape area is empty and the Exit Confirmation window is not opened). Figure 5
shows an FSM for this responsibility where each node represents a GUI state and each edge represents a state transition. Note that the states in this FSM are abstract states representing several states in the FSM in Section 3.1.1. For example, $S_{x00C}$ is an abstraction of all states where the Circle radio button is selected ($x$ can be replaced by any value of the corresponding state element).

![Figure 5: FSM for the create a new shape responsibility.](image)

The subFSM consisting of the two states $S_{x00C}$ and $S_{x00S}$ is a strongly connected component. Thus, this subFSM can be tested in isolation and then replaced it by a superstate $S_{x00x}$ (i.e., a shape is selected). To test the internal behaviors of the subFSM, the state sequence needed to be covered is $\langle S_{x00C}, S_{x01S} \rangle$; which is obtained by the event sequence...
\(\langle \text{square, circle} \rangle\). With an assumption that the subFSM is well tested, state sequence needed to test the reduced FSM is \(\langle \text{Initial, } S_{200x}, S_{201x}, S_{200x}, S_{201x}, \text{Terminal} \rangle\). This sequence is then translated to an executable test case taking the GUI from the initial state to the terminal state: \(\langle \text{create, reset, create} \rangle\).

### 3.1.4 Off-nominal Finite State Machines

The three approaches discussed thus far generate test cases to test the GUI for legal event sequences specified in the state machine model. However, the GUI might have been coded incorrectly to allow other sequences left unspecified in the state machine. For example, in our Radio Button Demo GUI, does the GUI allow the user to click on the \textit{reset} button when the application is launched, or after \textit{reset} has been executed once? For example, is the sequence \(\langle \text{reset, reset, reset} \rangle\) allowed?

The implicit assumption is that such off-nominal sequences are illegal and should not be allowed by the GUI. Belli \textit{et al.} \cite{9} argue that these sequences should also be tested in addition to the legal sequences. They augment the \textit{Complete Interaction Sequence} approach to test the GUI system’s robustness by generating such off-nominal test cases. The augmented model is called \textit{Faulty Complete Interaction Sequence (FCIS)}.

As was the case for the CIS, each FCIS can be specified by an FSM. This FSM is constructed by the following steps:

1. Build the CIS and the corresponding FSM consisting of all legal sequences of user-system interactions. Each edge of the FSM is called an \textit{Interaction Pair (IP)}.

2. Identify Faulty Interaction Pairs (FIPs) consisting of inputs that are not legal. These are all the “missing” IPs in the original FSM. Note that FIPs and IPs together define a complete FSM called the \textit{Complete Finite State Automata (CFSA)}.

Figure 6 shows an FSM of the FCIS corresponding to the CIS in Figure 5. The solid lines in the graph represent the FIPs and the dotted lines represent the edges in the CIS’s
Test case generation for a FCIS is straightforward. The tester can systematically design test cases for various undesired system behaviors by covering all possible FIPs. One way to do this is to select an untested FIP, i.e., an edge in the FCIS, generate a sequence of events from the FSM’s start state to the first event in the selected edge, and prepend this sequence to the edge, creating a test case that will test the selected FIP. Once this is done for each FIP, all of them would be tested and covered.

As we can see in Figure 6, there is one FIP in the FSM: $\langle S_{x01x}, S_{x00S} \rangle$. By prefixing this FIP with the state sequence $\langle \text{Initial}, S_{x00C}, S_{x01x} \rangle$, we get a complete sequence in the CIS.
CFSA to examine the illegal behavior: ⟨Initial, S_{x00C}, S_{x01x}, S_{x00S}⟩. The sequence can be translated to a test case which is a sequence of events starting at the initial state: ⟨create, reset⟩.

3.2 Workflows

Some researchers have used the GUI's business workflow, i.e., a sequence of connected steps, for test case generation. A typical GUI workflow is represented by a set of events (the steps) and some type of sequencing relationship between the events. In this section, we describe the Event Flow Graph model [32], a seminal work in this category. Then, we present two variants of this model: the Event Interaction Graph [33] and the Event Semantic Interaction Graph [46]. Finally, we discuss the Faulty Event Sequence Graph [9], an off-nominal model for the workflow-based approach.

3.2.1 Event Flow Graph

Intuitively, an Event Flow Graph (EFG) represents all possible event sequences that may be executed on a GUI [32]. The graph nodes represent events in the GUI and the graph edges represent a sequencing relationship that shows the set of events events that may be performed immediately after a given event. The concept of the EFG is similar to that of a control-flow or program-flow graph [6] that capture the flow of all possible executions of program statements, except that an EFG represents the flow of events, not code, in a GUI.

Definition: An EFG for a GUI G is formally defined as a triple <V, E, B> where:

1. V is a set of vertices representing all the events in G. Each v ∈ V represents an event in G.
2. E ⊆ V × V is a set of directed edges between vertices. Event e_j follows e_i (or equivalently e_j = follows(e_i)) iff e_j may be performed immediately after e_i. An edge (v_x, v_y) ∈ E iff the event represented by v_y follows the event represented
3. $B \subseteq V$ is a set of vertices representing initial events of $G$ that are available to the user when the GUI is first invoked.

The EFG for the Radio Button Demo application is shown in Figure 7. The events are shown as oval nodes. The shaded nodes are initial events, i.e., they are available to the user when the GUI is first launched. The directed edges show the follows relationship between events. For example, a user can click on the Yes button in the Exit Confirmation window either immediately after clicking on the Exit button or immediately after clicking on $w_6$; hence there is an edge from exit to yes, and from (un)check to yes. The user cannot click on the Yes button after the No button because no closes the dialog; there is no edge from no to yes. Similarly, there is no edge from no to no; nor is there an edge from yes to yes.

An approximation of the EFG for a GUI can be automatically obtained by a reverse engineering process call GUI ripping [30]. All events available in the GUI are automatically

![Event Flow Graph](image-url)
performed to open the hidden widgets and windows in a depth-first manner. During the
GUI ripping process, the key attributes of each widget are captured (e.g., whether it opens
a modal/modeless window, it opens a menu, it closes a window). These attributes are then
used to automatically construct the EFG. Because such a process is unable to infer complex
state-based relationships between events, e.g., one enables/disables the other, a tester has
to manually check and edit it to obtain the final EFG.

Because the EFG captures all possible sequences of events that may be executed by a
user on the GUI, any path in the EFG is a valid user-executed event sequence, and hence, a
potential test case. Moreover, any graph traversal technique on the EFG can be used to yield
test cases. Examples of some techniques that have been used in the past are goal-directed
search [31], random-walk [33], and bounded breadth-first search [42]. For example, a random
walk of the EFG of Figure 7 may yield the test case ⟨square, square, circle, square, create, reset, exit, yes⟩.

3.2.2 Event Interaction Graph

Because the EFG captures all possible event sequences that may be executed on the GUI, the
number of event sequences that may be generated from an EFG becomes extremely large. In
fact, the number grows exponentially with sequence length [39, 17]. It is important to reduce
this number for practical reasons. To address this issue, Xie et al. [33, 43] conducted several
empirical studies on the characteristics of test cases derived from the EFG model. The
experiments showed that a large number of faults were detected by the test cases that tested
interactions between certain type of events which (1) close a modal window (termination
events) or (2) interact with the underlying code (system-interaction events). Other events
used to manipulate the GUI structure such as open or close menu/modeless windows, called
structural events, are unlikely to reveal faults. One possible explanation for these results was
that the code for structural events is usually simple and generated automatically by visual
GUI-building tools; therefore it is less likely to be faulty. Based on these results, a new
model called the Event Interaction Graph (EIG) was developed.

21
Intuitively, an EIG contains only termination and system-interaction events; an edge between two nodes in the EIG shows that one event might be executed after (not necessarily immediately after) the other along some execution path. Formally, EIG edges are defined by an interacts-with relation through the following definitions:

**Definition:** There is an event-flow-path from node \( n_x \) to node \( n_y \) iff there exists a (possibly empty) sequence of nodes \( n_{j}; n_{j+1}; n_{j+2}; \ldots; n_{j+k} \) in the event-flow graph \( E \) such that \( \{(n_x, n_j), (n_j, n_y)\} \subseteq \text{edges}(E) \) and \( \{(n_{j+i}, n_{j+i+1}) \mid 0 \leq i \leq (k - 1)\} \subseteq \text{edges}(E). \)

**Definition:** An event-flow-path \(< n_1; n_2; \ldots; n_k >\) is interaction-free iff none of \( n_2, \ldots, n_{k-1} \) represent termination or system-interaction events.

**Definition:** A system-interaction (or termination) event \( e_x \) interacts-with system-interaction and termination event \( e_y \) iff there is at least one interaction-free event-flow-path from the node \( n_x \) (that represents \( e_x \)) to the node \( n_y \) (that represents \( e_y \)).

The EIG edges actually represent the above interacts-with relationship between events. An EFG can be automatically transformed into an EIG by using graph-rewriting rules (details are presented in [44]). The EIG for the Radio Button Demo application is shown in Figure 8. Note that the EIG does not contain the window-opening exit event. The graph-rewriting rule used to obtain this EIG was to (1) delete exit because it is a window-open event, (2) for all remaining events \( e_x \) replace each edge \((e_x, \text{exit})\) with edge \((e_x, e_y)\) for each occurrence of edge \((\text{exit}, e_y)\), and (3) for all \( e_y \), delete all edges \((\text{exit}, e_y)\).

As was the case with EFGs, a test case in the EIG model is also a path in the EIG, starting with an initial event. One possible test case might be \(<\text{square}, \text{square}, \text{circle}, \text{yes}>\). Because EIG nodes do not represent events to open or close menus/windows, the sequences obtained from the EIG may not be executable. For example, the test case \(<\text{square}, \text{square}, \text{circle}, \text{yes}>\) will not execute because \text{yes} is not available for execution after \text{circle}. For that reason, at execution time, other events needed to reach the EIG events are automatically inserted using
the original EFG. During the test-case execution, the EIG test case above will be expanded to \((\text{square}, \text{square}, \text{circle}, \text{exit}, \text{yes})\)

### 3.2.3 Event Semantic Interaction Graph

Although the EIG model is smaller than the EFG, it is still a dense graph and suffers from the same problems as does the EFG – the number of generated event sequences grows exponentially with length. In more recent work, Yuan et al. \cite{46} create a sparse graph, where events are connected by edges only if they were shown to influence each other’s execution behavior. Consider the Radio Button Demo example. The top-left GUI in Figure 9 shows the initial state \(S_0\) of the application. After an event square is executed, the GUI changes its state to the one shown in the top-right \((\text{square}(S_0))\). In this state, the Square radio button is selected. Starting from \(S_0\), one can execute another event \((\text{create})\) and obtain the state shown in the bottom-left \((\text{create}(S_0))\); a circle is created by clicking the Create button. If, however, the sequence \((\text{square}; \text{create})\) is executed in \(S_0\), a new state \((\text{create}(\text{square}(S_0)))\), shown in the bottom-right is obtained; a square has been created. This execution is equiv-
alent to the execution of event \textit{create} in the state \textit{square}(S_0). The event \textit{square} clearly influences the event \textit{create}. We say that event \textit{square} “interacts with” event \textit{create}, and should be tested together to check for interaction problems.

The main idea behind observing GUI run-time states and using them to determine which events to test together can also be justified by examining the code of event handlers. For example, the event handlers for \textit{square} and \textit{create} share two variables \texttt{created}, which indicates if a shape is created, and \texttt{currentShape}, which specifies the current selected shape; \textit{create} sets \texttt{created} to TRUE and influences \textit{square}’s flow of control; \textit{square} sets \texttt{currentShape} to a square, which \textit{create} uses as a parameter to create a shape; hence it’s not surprising that they influence each other’s execution.

The example illustrated in Figure 9 is just one case of how the GUI state may be used
to pinpoint interactions between event handlers. Yuan et al. formally define six cases that
describe (as evaluative predicates) situations in which two events, called \(e_1\) and \(e_2\), interact,
\textit{i.e.}, \(e_1\) influences \(e_2\). In these six cases, \(e_1\) and \(e_2\) are system-interaction events in modeless
windows; this situation is referred as Context 1.

**Case 1:** \(\mathcal{P}_{1(1)}(e_1, e_2) = \exists w \in W, p \in P_w, v \in V_p, v' \in V_p, \text{s.t.} \ (v \neq v') \land ((w, p, v) \in \{S_0 \cap e_1(S_0) \cap e_2(S_0)\}) \land ((w, p, v') \in e_1(S_0))\); there is at least one widget \(w\) with
property \(p\) with initial value \(v\) (hence the triple \((w, p, v)\) is in \(S_0\)), which is not affected by
the individual events \(e_1\) or \(e_2\) (the triple is also in \(e_1(S_0)\) and \(e_2(S_0)\)); however, it is modified
when the sequence \((e_1; e_2)\) is executed, \textit{i.e.}, the value of \(w\)'s property \(p\) changes from \(v\) to \(v'\).

**Case 2:** \(\mathcal{P}_{2(1)}(e_1, e_2) = \exists w \in W, p \in P_w, v \in V_p, v' \in V_p, v'' \in V_p, \text{s.t.} \ (v \neq v') \land (v' \neq
v'') \land ((w, p, v) \in \{S_0 \cap e_2(S_0)\}) \land ((w, p, v') \in e_1(S_0)) \land ((w, p, v'') \in e_2(S_0))\) there is at
least one widget \(w\) with property \(p\) that has an initial value \(v\), which is not modified by the
event \(e_2\); it is modified by \(e_1\); however, it is modified differently by the sequence \((e_1; e_2)\).

In our running example, widget \(w_4\), in the GUI’s initial state, is not modified by event
square, \textit{i.e.}, it remains empty; it is modified by event create, \textit{i.e.}, a circle is shown; however,
\(w_4\) is modified differently by the sequence \((create; square)\). Hence, Case 2 applies to create
and square.

**Case 3:** \(\mathcal{P}_{3(1)}(e_1, e_2) = \exists w \in W, p \in P_w, v \in V_p, v' \in V_p, v'' \in V_p, \text{s.t.} \ (v \neq v') \land (v' \neq
v'') \land ((w, p, v) \in \{S_0 \cap e_1(S_0)\}) \land ((w, p, v') \in e_2(S_0)) \land ((w, p, v'') \in e_2(S_0))\) there is at
least one widget \(w\) with property \(p\) that has an initial value \(v\), which is not modified by the
event \(e_1\); it is modified by \(e_2\); however, it is modified differently by the sequence \((e_1; e_2)\).
Note that this case is different from Case 2 because the event sequence remains the same,
\textit{i.e.}\(e_1\) is executed before \(e_2\).

In our running example, widget \(w_4\), in the GUI’s initial state, is not modified by event
square, \textit{i.e.}, it remains empty; it is modified by event create, \textit{i.e.}, a circle is shown; however,
\(w_4\) is modified differently by the sequence \((square; create)\). Hence, Case 3 applies to square
and create.
Case 4: $P_{4(1)}(e_1, e_2) = \exists w \in W, p \in P_w, v \in V_p, v' \in V_p, v'' \in V_p, \tilde{v} \in V_p, s.t. ((v \neq v') \land (v \neq v'')) \land (v'' \neq \tilde{v}) \land ((w, p, v) \in S_0) \land ((w, p, v') \in e_1(S_0)) \land ((w, p, v'') \in e_2(S_0)) \land ((w, p, \tilde{v}) \in e_2(e_1(S_0))))$; there is at least one widget $w$ with property $p$ that has an initial value $v$, which is modified by individual events $e_1$ and $e_2$; however, it is modified differently by the sequence $\langle e_1; e_2 \rangle$.

The above four cases all handle widgets that persist across the four states being considered, i.e., $S_0$, $e_1(S_0)$, $e_2(S_0)$, and $e_2(e_1(S_0))$. In many cases, event execution “creates” new widgets, e.g., by opening menus; the next case handles newly created widgets.

Case 5: $P_{5(1)}(e_1, e_2) = \exists w \in W; p \in P_w; v \in V_p; v' \in V_p; s.t.: ((v \neq v') \land ((w, p, v) \in e_x(S_0)) \land ((w, p, v) \notin S_0) \land ((w, p, v') \in e_2(e_1(S_0))))$; there is at least one new widget $w$ with property $p$ and value $v$ in $e_x(S_0)$, i.e., it was created by event $e_x$ (either $e_1$ or $e_2$) but did not exist in state $S_0$; it was created by the sequence $\langle e_1; e_2 \rangle$ but with a different value for some property.

A common occurrence of event interaction in GUIs is enabling/disabling widgets, which may be modeled as the widget’s ENABLED property being set to TRUE or FALSE.

Case 6: $P_{6(1)}(e_1, e_2) = \exists w \in W, \text{ENABLED} \in P_w, \text{TRUE} \in V_{\text{ENABLED}}, \text{FALSE} \in V_{\text{ENABLED}}, s.t. ((w, \text{ENABLED, TRUE}) \in e_1(S_0)) \land \text{EXEC}(e_2, w)$; there exists at least one widget $w$ that was disabled in $S_0$ but enabled by $e_1$. Event $e_2$ is performed on $w$, represented by a predicate EXEC$(e_2, w)$.

In our running example, create enables reset; hence Case 6 applies.

Modal windows create special situations for Cases 1 through 6 due to the presence of termination events. User actions in these windows do not cause immediate state changes; they typically take effect after a termination event has been executed, leading to contexts 2 and context 3.

Context 2: If both $e_1$ and $e_2$ are associated with widgets that are contained in one modal window with termination event TERM, then the definitions of $e_1(S_0)$, $e_2(S_0)$, and $e_2(e_1(S_0))$ are modified as follows: $e_1(S_0)$ is the state of the GUI after the execution of the event
sequence \( \langle e_1; \text{TERM} \rangle, e_2(S_0) \) is the state of the GUI after the execution of the event sequence \( \langle e_2; \text{TERM} \rangle \), and \( e_2(e_1(S_0)) \) is the state of the GUI after the execution of the event sequence \( \langle e_1; e_2; \text{TERM} \rangle \). All the predicates defined in Cases 1 through 6 apply, using these modified definitions, for \( e_1 \) and \( e_2 \) in the same modal window. The notation used for these predicates when applied in Context 2 is \( P_{n(2)}(e_1, e_2) \), where \( n \) is the case number.

**Context 3:** If \( e_1 \) is associated with a widget contained in a modal window with termination event \( \text{TERM} \), and \( e_2 \) is associated with a widget contained in the modal window’s parent window (i.e., the window that was used to open the modal window) then \( e_1(S_0) \) is the state of the GUI after the execution of the event sequence \( \langle e_1; \text{TERM} \rangle \), \( e_2(S_0) \) is the state of the GUI after the execution of the event \( e_2 \), and \( e_2(e_1(S_0)) \) is the state of the GUI after the execution of the event sequence \( \langle e_1; \text{TERM}; e_2 \rangle \). All the predicates defined in Cases 1 through 6 apply. The notation used for these predicates when applied in Context 3 is \( P_{n(3)}(e_1, e_2) \), where \( n \) is the case number.

There is an Event Semantic Interaction relationship between two events \( e_1 \) and \( e_2 \) at least one of the predicates in Cases 1 through 6 evaluates to \( \text{TRUE} \) in at least one context. If multiple cases apply, then one of the case numbers is used. Due to the specific ordering of the events in the sequence \( \langle e_1; e_2 \rangle \), the ESI relationship is not symmetric. As demonstrated earlier, for our Radio Button Demo application: \( \text{square} \rightarrow \text{create} \), \( \text{create} \rightarrow \text{square} \), and \( \text{create} \rightarrow \text{reset} \).

Once all of the cases have been implemented, the feedback-based process execution is straightforward. The steps of the execution are as follows.

1. The seed suite consisting of all 2-way interactions \( \langle e_x; e_y \rangle \) between GUI events is executed on the software in state \( S_0 \); these test cases are simple enumerations of all EIG edges. All events \( e_y \) are also executed in \( S_0 \). The state information \( e_x(S_0) \), \( e_y(S_0) \), \( e_y(e_x(S_0)) \) is collected and stored.

2. The above predicates are evaluated for each pair of system-interaction events in the EIG that are either (1) directly connected by an edge (Context 1) or (2) connected by a path that does not contain any intermediate system-interaction events (contexts 2
and 3), i.e., there is at least one termination event that closes a modal window on this path. If one of the predicates evaluates to TRUE, the two events are ESI-related.

Once all the ESIs in a GUI have been identified, a graph model called the ESI graph (ESIG) is created. The ESIG contains nodes that represent events; a directed edge from node $n_x$ to $n_y$ shows that there is an ESI relationship from the event represented by $n_x$ to the event represented by $n_y$. Figure 10 shows the ESIG of the Radio Button Demo GUI. The solid lines are ESIG edges; for comparison, we also show the EFG edges (dotted lines) and EIG edges (dashed lines).

As was the case for EFGs and EIGs, the ESIG may be traversed using different graph traversal algorithms to generate test cases. For our example ESIG in Figure 10, two test cases are $(create; reset)$ and $(square; create; square; create; reset)$.
3.2.4 Off-nominal Event Graph

Belli et al. develop a technique to generate off-nominal test cases using the GUI’s workflow [9]. They define the workflow as an Event Sequence Graph (ESG).

**Definition:** An event sequence graph $ESG = (V, E)$ is a directed graph where $V \neq \emptyset$ is a finite set of vertices (nodes), $E \subseteq V \times V$ is a finite set of arcs (edges), and $\Xi, \Gamma \subseteq V$ are finite sets of distinguished vertices with $\xi \in \Xi$ and $\gamma \in \Gamma$ called entry nodes and exit nodes, respectively, wherein $\forall v \in V$ there is at least one sequence of vertices $<\xi, v_0, \ldots, v_k>$ from each $\xi \in \Xi$ to $v_k = v$ and one sequence of vertices $<v_0, \ldots, v_k>$ from $v_0 = v$ to each $\gamma \in \Gamma$ with $(v_i, v_{i+1}) \in E$, for $i = 0, \ldots, k-1$ and $v \neq \xi, \gamma$.

Intuitively, the ESG is similar to the event-flow graph, except that there is a notion of exit nodes in an ESG. Such a workflow allows the definition of an event sequence (ES).

**Definition:** Let $V, E$ be as defined above. Then any sequence of vertices $<v_0, \ldots, v_k>$ is called an event sequence ES if $(v_i, v_{i+1}) \in E$, for $i = 0, \ldots, k$.

This definition is used to define a complete event sequence (CES) in the ESG.

**Definition:** An ES is a complete ES (or, it is called a complete event sequence, CES), if $\alpha(ES) = \xi \in \Xi$ is an entry and $\beta(ES) = \gamma \in \Gamma$ is an exit.

where $\alpha$ and $\beta$ are the manually defined functions used to determine the entry and exit vertex of an ES.

These above definitions allow the formal definition of an off-nominal test case (or faulty event sequence) based on the ESG.

**Definition:** For an $ESG = (V, E)$, its completion is defined as $\widehat{ESG} = (V, \widehat{E})$ with $\widehat{E} = V \times V$.

**Definition:** The inverse (or complementary) $ESG$ is then defined as $\overline{ESG} = (V, \bar{E})$ with $\bar{E} = \bar{E} \setminus E$. 

29
Figure 11 shows the inverse ESG of the Radio Button Demo GUI. The dotted edges are ESG (EFG) edges. The oval shaded nodes represent initial events while the octagon nodes represent exit events.

Figure 11: Inverse Event Sequence Graph.

The solid edges in Figure 11 are the ones that are absent from the ESG. More formally, they represent faulty event pairs.

**Definition:** Any edge of the ESG is a faulty event pair (FEP) for the ESG.

**Definition:** Let \( ES = v_0, \ldots, v_k \) be an event sequence of length \( k + 1 \) of an ESG and \( FEP = < v_k, v_m > \) a faulty event pair of the corresponding ESG. The concatenation of the \( ES \) and \( FEP \) then forms a faulty event sequence \( FES = < v_0, \ldots, v_k, v_m >. \)

Such an FES can be used as an off-nominal test case. An example of such a test case for our running example is \( (\text{square}, \text{circle}, \text{reset}, \text{no}) \). The pair \( (\text{reset}, \text{no}) \) should not be executable because of the Exit Confirmation modal dialog.
3.3 Pre- Post-Condition Models

In an approach presented by Memon et al. [32], the test designer models the GUI in terms of pre- and post-conditions for each event. The test designer then identifies commonly used tasks for the GUI; these are then input to the test case generator. The generator employs the pre- and post-conditions and specifications to generate event sequences to achieve the tasks.

The motivating idea behind this approach is that GUI test designers will often find it easier to specify typical user goals than to specify sequences of GUI events that users might perform to achieve those goals. The software underlying any GUI is designed with certain intended uses in mind; thus the test designer can describe those intended uses. However, it is difficult to manually obtain different ways in which a user might interact with the GUI to achieve typical goals. Users may interact in idiosyncratic ways, which the test designer might not anticipate. Additionally, there can be a large number of ways to achieve any given goal, and it would be very tedious for the GUI tester to specify even those event sequences that s/he can anticipate. The test case generator described in this section uses AI planning to generate GUI test cases for commonly used tasks using a GUI model based on pre- and post-conditions of all GUI events.

The test case generation process is partitioned into two phases, the setup phase and plan-generation phase. In the first step of the setup phase, the GUI representation is employed to identify planning operators, which are used by the planner to generate test cases. By using knowledge of the GUI, the test designer defines the preconditions and effects of these operators. During the second or plan-generation phase, the test designer describes scenarios (tasks) by defining a set of initial and goal states for test case generation. Finally, the AI planning system generates a test suite for the tasks using the plans. The test designer can iterate through the plan-generation phase any number of times, defining more scenarios and generating more test cases.

Formally, a planning problem \( P(\Lambda, D, I, G) \) is a 4-tuple, where \( \Lambda \) is the set of operators,
D is a finite set of objects, I is the initial state, and G is the goal state. Note that an operator definition may contain variables as parameters; typically an operator does not correspond to a single executable action but rather to a family of actions, one for each different instantiation of the variables. The solution to a planning problem is a plan: a tuple < S, O, L, B > where S is a set of plan steps (instances of operators, typically defined with sets of preconditions and effects), O is a set of ordering constraints on the elements of S, L is a set of causal links representing the causal structure of the plan, and B is a set of binding constraints on the variables of the operator instances in S. Each ordering constraint is of the form S_i < S_j (read as “S_i before S_j”) meaning that step S_i must occur sometime before step S_j (but not necessarily immediately before). Typically, the ordering constraints induce only a partial ordering on the steps in S. Causal links are triples < S_i, c, S_j >, where S_i and S_j are elements of S and c represents a proposition that is the unification of an effect of S_i and a precondition of S_j. Note that corresponding to this causal link is an ordering constraint, i.e., S_i < S_j. The reason for tracking a causal link < S_i, c, S_j > is to ensure that no step “threatens” a required link, i.e., no step S_k that results in ¬c can temporally intervene between steps S_i and S_j.

For the Radio Button Demo application, one possible task may be to create a square shape for w_4. This task is shown in Figure 12. Even with this simple application, there are several ways to perform this task. In fact, there are an infinite number of ways—in principle, a user can click on the Square radio button an arbitrary number of times. This task is input to the planner by describing the state of all the widgets in the initial and goal states.

Together with a specification of all pre- and postconditions of the events, the task is used by the planner to output the plan shown in Figure 13(a). As mentioned above, most AI planners produce partially-ordered plans, in which only some steps are ordered with respect to one another. The plan in Figure 13(a) is one such plan. The ordering constraints are shown as edges and also explicitly stated in Figure 13(b).

A total-order plan can be derived from a partial-order plan by adding ordering constraints,
induced by removing threats. Each total-order plan obtained in such a way is called a linearization of the partial-order plan. A partial-order plan is a solution to a planning problem if and only if every consistent linearization of the partial-order plan meets the solution conditions. Figure 13(c) shows the two linearizations of the plan; each of these linearizations can be used as a test case.
3.4 Event Sequence-Based Models

Because GUI test cases are sequences of events, Kasik et al. [27] manipulate such sequences of events to obtain new test cases. Their approach is based on genetic algorithms.

The key motivation behind using genetic algorithms is that there is a need to test the GUI from the perspective of different groups of users, e.g., experts and novice users. Unsophisticated and novice users often exercise GUI applications in ways that the designer, the developer, and the tester did not anticipate. An expert user or tester usually follows a predictable path through an application to accomplish a familiar task. The developer knows where to probe, to find the potentially problematic parts of an application. Consequently, applications are well tested for state transitions that work well for predicted usage patterns but become unstable when given to novice users. Novice users follow unexpected paths in the application, causing program failures. Such failures are difficult to predict at design and testing time.

One approach to test the GUI for novice interactions is to release the software to a small community for beta testing. However, this approach is expensive and time-consuming. Kasik et al.’s approach generates test cases that mimic a novice user. The key idea behind this approach is that expert users take short paths through an application’s GUI, using shortcuts when available and perform their tasks quickly. Novice users, on the other hand, take longer, exploratory paths to complete a task and gradually build better ways as they learn more about the application. It is challenging to automatically generate these paths for GUI testing.

In its simplest form, a genetic algorithm manipulates a table of random numbers; each row of the table represents a gene. The individual elements of a row (gene) contain a numeric genetic code and are called alleles. Allele values start as numbers that define the initial genetic code. The genetic algorithm lets genes that contain “better” alleles survive to compete against new genes in subsequent generations.

The basic genetic algorithm is as follows:
• Initialize the alleles with valid numbers.

• Repeat the following until the desired goal is reached:
  
  – Generate a score for each gene in the table.
  
  – Reward the genes that produce the best results by replicating them and allowing them to live in a new generation. All others are discarded using a death rate.
  
  – Apply two operators, mutation and crossover, to create new genes.

For GUIs testing, the event sequence is represented by a gene, each element being an event. The primary task of setting up the genetic algorithm is to set the death rates, crossover styles, and mutation rates so that novice behavior is generated. Also, to use genetic algorithms to generate meaningful interactions mimicking novice users, a clear and accurate specification of both the user interface dialog and the program state information is needed. The state information controls the legality of specific dialog components and the names of a legal command during an interaction. Without access to the state information, the generator may produce many meaningless input events.

For our running example, the Radio Button Demo GUI, an expert might use \langle square; create \rangle to create a square. The genetic algorithm may convert this sequence into the longer sequence \langle circle; create; square; create \rangle, thereby mimicking a novice user.

### 3.5 Probabilistic Models

As seen in this chapter, there are several techniques to generate GUI test cases based on a model of the GUI. In practice, a GUI test designer may use a mix of these techniques to obtain several test suites. The test designer is faced with two significant challenges:

• **Overlaps in test suites:** As can be imagined, many of these techniques often overlap in what they test. A test designer who uses two or more GUI testing techniques may waste valuable resources testing and retesting the same parts of the GUI. Ideally, the
test designer would like to consolidate all the test suites and obtain one suite that minimizes overlaps.

- Large number of short tests and few long tests: The sheer size of the individual suites presents practical problems for test execution. Because each test case requires significant overhead in terms of setup and teardown, having a large number of short tests is inefficient. Ideally, the test designer would like to obtain longer sequences that combine the strengths of individual short-sequence suites.

Consider for example, the three test suites shown in Figure 14, each generated using a different technique. It may be expensive to execute and maintain all these test cases. Brooks et al. [10] employ a probabilistic model of the GUI to combine these suites.

![Figure 14: Example test cases.](image)

The probabilistic model is based on the event-flow graph model. The model contains a collection of $R$ paths through the EFG called $r_1, r_2, \ldots, r_R$. Each path $r_i$ where $1 \leq i \leq R$, consists of a sequence of $n$ events in addition to $INIT$ and $FINAL$:

$$r_i = INIT, x_1, x_2, \ldots, x_n, FINAL;$$

$$\forall j \ e_j \in \{e_1, e_2, \ldots, e_{n-1}\} \land$$

$$follows(e_{j+1}, e_j)$$

where $x_1, x_2, \ldots, x_n$ and $e_1, e_2, \ldots, e_{n-1}$ are events in the EFG, $r_i$ denotes a path, and each path $r_i$ contains only events with a follows relationship between them. Valid paths can also be formed by the concatenation of two paths, e.g., $r_a$ and $r_b$, provided the first event of $r_b$ follows the last event of $r_a$ in the EFG.
Let \(\text{count}(e_i)\) return the number of times event \(e_i\) occurs in the paths \(r_1, r_2, \ldots, r_R\). The prior probability that a randomly selected event from any of \(r_1, r_2, \ldots, r_R\) is \(e_i\) is:

\[
P(e_i) = \frac{\text{count}(e_i)}{\sum_{j=1}^{E} \text{count}(e_j)}.
\]

Now, \(\text{count}(e_i)\) and the prior probability calculation are extended from single events to sequences of events. Let \(s\) be a length-\(S\) subsequence of some path through the EFG (not necessarily in \(r_1, r_2, \ldots, r_R\)):

\[
s_i = x_1, x_2, \ldots, x_S
\]

\(\forall j \ e_j \in \{\text{INIT}, e_1, e_2, \ldots, e_{n-1}, \text{FINAL}\} \land \text{follows}(e_{j+1}, e_j)\).

The prior probability that a randomly selected, length-\(S\) subsequence from any of \(r_1, r_2, \ldots, r_R\) turns out to be \(s\) is

\[
P(s) = \frac{\text{count}(s)}{\sum_{s_i \in \text{subs}(S)} \text{count}(s_i)},
\]

where \(\text{count}(s)\) returns the number of times \(s\) occurs as a subsequence of \(r_1, r_2, \ldots, r_R\) and \(\text{subs}(S)\) is the set of all length-\(S\) subsequences in \(r_1, r_2, \ldots, r_R\).

Given that \(s\) immediately precedes \(e_i\), the conditional probability of \(e_i\) is

\[
P(e_i|s) = \frac{P(s_1, s_2, \ldots, s_S, e_i)}{\sum_{j=1}^{E} P(s_1, s_2, \ldots, s_S, e_j)}.
\]

Note that \(P(e_i|s)\) can be thought of as \(P(e_i)\) when \(s\) has length 0. This is not the same as \(P(e_i|\text{INIT})\), which is the probability that event \(e_i\) is the first event in the sequence, occurring immediately after \(\text{INIT}\). Rather, \(P(e_i|s)\) is the probability of \(e_i\) given no information about the events that precede it.

A probabilistic EFG (PEFG) is created by annotating each event (node) in the EFG with a table containing the event’s prior probability and its probability conditioned on each
subsequence in \{r_1, r_2, \ldots, r_R\} up to some maximum subsequence length, or history, \(H\).

Figure 15 shows the PEFG obtained for the test suites of Figure 14. Column 2 of each table associated with every node shows the probability of executing the event associated with the node after the length 2 sequence shown in Column 1 of the table. For example, the entry for node (\uncheck) corresponding to row \textit{exit}, (\uncheck) is 0.5. This is because the subsequence \textit{exit}, (\uncheck) appears twice in the original test suites. Once \textit{exit}, (\uncheck) has been executed, there is a 0.5 probability that the next event will be (\uncheck). These probabilities can be used to generate event sequences. One example sequence is \textit{INIT}, \textit{exit}, (\uncheck), \textit{FINAL}. The resulting test case is \langle \textit{exit}, (\uncheck) \rangle.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{pefg_history_H2.png}
\caption{Probabilistic Event Flow Graph with history \(H = 2\).}
\end{figure}
3.6 Combinatorial Interaction Models

Software system faults are not only caused by individual components working in isolation but also caused by the interactions between them [11, 15]. In its basic form, GUI interaction testing consists of testing for interactions between all GUI components and their selections. However, since the number of GUI components is often huge, the number of tests required to cover the combinational interactions grows large very quickly [46]. Several combinational interaction models have been proposed to model GUI component interactions and reduce the number of test cases. This section presents two combinatorial models used for test case generation – a Latin square to cover pair-wise interactions [39] and a Covering Array to cover multi-way interactions with an arbitrary coverage strength [45].

3.6.1 Latin Squares

White [39] proposes the use of Latin squares to model the GUI inputs and generate test cases. He identifies two ways in which GUI interactions can arise: statically and dynamically (or a combination of both). Static interactions are restricted to one screen whereas dynamic interactions move from one screen to another to perform events on GUI objects. White makes the assumption that it is enough to test pair-wise interactions of GUI events. Similar assumptions have led to success in finding errors efficiently for conventional software [16].

The concept of Latin square is used to maintain the pairwise interaction coverage while keeping the number of test cases minimized.

Definition: A Latin square, of order \(n\), is a matrix of \(n\) symbols in a \(n \times n\) cells, arranged in \(n\) rows and \(n\) columns, such that every symbol exactly once in each row and once in each column.

Definition: A pair of Latin squares \(A = (a_{ij})\) and \(B = (b_{ij})\) are orthogonal iff the ordered pairs \((a_{ij}, b_{ij})\) are distinct for all \(i\) and \(j\). In other words, when superimposed on each other, the ordered elements pairs of two orthogonal squares created in each cell cover
Given $k$ factors $F_1, F_2, \ldots, F_k$, where each factor is a GUI component from which selections are made. The GUI inputs are modeled as follow:

- Reorder $k$ factors by cardinality: $|F_1| \geq |F_2| \geq \cdots \geq |F_k|$.
- Construct $k-2$ orthogonal Latin squares with size $n$, where $n$ is the cardinality of $|F_1|$.

To test $k$ GUI components with maximum $n$ level, we need $k-2$ orthogonal Latin squares. The cell entries of the superimposed square represent $k-2$ components in the test and the row and column indices represent the additional 2 components. Since the generated triples (row index, column index, cell entry) are unique, the pairwise coverage requirement is guaranteed.

The original model proposed by White only considered menu items. Because our running example does not have menus, we cannot use this approach to test our example GUI.

### 3.6.2 Covering Arrays

Yuan et al. [45] use covering arrays [18] to generate test cases. The key motivation behind using covering arrays is to generate longer sequences that are systematically sampled at a particular coverage strength. This approach is a generalization of the Latin square discussed in the previous section; a fundamental difference is that in covering arrays, the coverage strength is not limited to 2-way interactions. Furthermore, the use of covering arrays allows fine control over the location of each event in the test case.

**Definition:** A covering array $CA(N; t, k, v)$ is an $N \times k$ array on $v$ symbols with the property that every $N \times t$ sub-array contains all ordered subsets of size $t$ of the $v$ symbols at least once. In other words, any subset of $t$-columns of this array will contain all $t$-combinations of the symbols.
Constructing a covering array with a minimal number of rows is an optimization problem. There are both mathematical algorithms [25], as well as computational techniques such as greedy [14] and meta-heuristic search [16] for this problem.

This test case generation technique leverages covering arrays to keep the number of test cases minimized while maintaining a required $t$-way coverage between GUI events. A GUI is taken as input and first partitioned into different parts. Then, for each GUI part, a covering array is constructed to cover all events inside it. The output of this process is a set of covering arrays for all GUI partitions. Each array row becomes a GUI test case.

For our example Radio Button Demo application, we first partition the events into different groups. For example, the three events $(\text{un})\text{check}$, yes and no in the Exit Confirmation window can form the ‘Exit’ group. Suppose we are interested in 2-way coverage (i.e., test all possible 2-way interactions shown in Figure 16(a)) such that each event occupies all four positions in a length 4 sequence. If we used exhaustive enumeration, we need $3 \times 3 \times 3 \times 3 = 81$ test cases. Formulating the problem as a covering arrays $CA(N;2,4,3)$, Figure 16(b), the number of test cases is only nine, each of which becomes a test case.

![2-way covering and covering array](Figure 16)

### 3.7 Hierarchical Models

All of the testing techniques discussed thus far use a single model of the GUI. However, using only one model may be impractical for a large GUI. Several researchers have addressed this
problem by modeling the GUI at multiple levels of abstraction. The GUI is broken down into different components and modeled hierarchically. We now discuss three such hierarchies, namely Keyword-driven hierarchy [28], Hierarchical finite state machines [36], and UML-diagram based hierarchy [35].

3.7.1 Keyword-driven Models

**Keyword-driven testing** [12] is a script-based testing technique widely used in Industry. This technique divides the test case generation process into two phases: test plan and test implementation. In the test plan phase, the test designers design test cases using high-level activities called *action words*. In the test implementation phase, the test engineers transform the action words into executable events called *keywords*. To avoid ambiguities, the selected keywords are unique.

The idea behind using abstract test cases, *i.e.*, those that contain high-level action words, is that domain experts, without any implementation skills, can easily design test cases using only the action words. This step can be done early, even before the system implementation has been started. The abstract test cases are also easier to comprehend; test maintenance is also more efficient.

Inspired by the keyword-driven testing technique, Antti et al. [28] propose a GUI testing model using Label Transition Systems (LTS). A LTS is a state machine whose transition names are taken from an alphabet. Formally, a LTS is defined as:

**Definition:** A labeled transition system (LTS) is a quadruple \((S, \Sigma, \Delta, \hat{s})\) where \(S\) is a set of states, \(\Sigma\) is a set of actions (alphabet), \(\Delta \subseteq S \times \Sigma \times S\) is a set of transitions and \(\hat{s} \in S\) is an initial state.

A GUI is modeled using two sets of LTSs corresponding to the two levels of abstraction in the keyword driven approach. The LTSs for the action word level are called *action machines* and the LTSs for the keyword level are called *refinement machines*. The
action machines provide an overview of the system while each refinement machines describes GUI navigation for certain parts of the GUI.

Figure 17(a) presents an action machine $\mathcal{A}$ for the Radio Button Demo application GUI. The labels in this machine represent the action words. Figure 17(b) is a refinement machine for the main window. The labels in this machine are keywords describing the actual GUI events.

These machines are automatically composed to an executable LTS by a parallel composition operator defined as follows.

**Definition:** $||_R(\mathcal{L}_1, \ldots, \mathcal{L}_n)$ is the *parallel composition* of $n$ LTSs according to rules $R$ where LTS $\mathcal{L}_i = (S_i, \Sigma_i, \Delta_i, s_i)$ if let $\Sigma_R$ be a set of resulting actions and $\checkmark$ be a “pass” symbol
such that \( \forall i : \sqrt{\text{=}} \notin \Sigma_i \). The rule set \( R \subseteq (\Sigma_1 \cup \{\sqrt{\text{=}}\}) \times \cdots \times (\Sigma_n \cup \{\sqrt{\text{=}}\}) \times \Sigma_R \). Now \( ||_R(\mathcal{L}_1, \ldots, \mathcal{L}_n) = (S, \Sigma, \Delta, \hat{s}) \) where:

- \( S = S_1 \times \cdots \times S_n \)
- \( \Sigma = \{ a \in \Sigma_R | \exists a_1, \ldots, a_n : (a_1, \ldots, a_n, a) \in R \} \)
- \( ((s_1, \ldots, s_n), a, (s'_1, \ldots, s'_n)) \in \Delta \) if and only if there is \( (a_1, \ldots, a_n, a) \in R \) such that for every \( i \) \( (1 < i < n) \)
  - \( (s_i, a_i, s'_i) \in \Delta \) or
  - \( a_i = \sqrt{\text{=}} \) and \( s_i = s'_i \)
- \( \hat{s} = (\hat{s}_1, \ldots, \hat{s}_n) \)

\( \square \)

A rule in a parallel composition associates an array of actions (or “pass” symbol \( \sqrt{\text{=}} \)) of input LTSs to an action in the resulting LTS. The action is the result of the synchronous execution of the actions in the array. If there is a \( \sqrt{\text{=}} \) instead of an action, the corresponding LTS will not participate in the synchronous execution described by the rule.

Assuming that we have the following composition rules:

\[
R = \{ (1)\langle \text{awCreateShape}, \text{kwClickCreate}, \text{kwClickCreate} \rangle \\
(2)\langle \text{awCreateShape}, \text{kwSelectCircle}, \text{kwClickCircle} \rangle \\
(3)\langle \text{awCreateShape}, \text{kwSelectSquare}, \text{kwClickSquare} \rangle \\
(4)\langle \text{awReset}, \text{kwClickReset}, \text{kwClickReset} \rangle \\
(5)\langle \text{awCancel}, \sqrt{\text{=}}, \text{awCancel} \rangle \\
(6)\langle \text{awQuit}, \sqrt{\text{=}}, \text{awQuit} \rangle \}
\]

Figure 17(c) shows the composition machine \( \mathcal{C} \) synthesized using the above rules. As we can see, the states in \( \mathcal{C} \) are a combination (product) of \( \mathcal{A} \)’s states and \( \mathcal{R} \)’s states. By
applying rules (1)-(4), two action words *awCreateShape* and *awReset* are refined to the corresponding keywords in C. However, the action words *awCancel* and *awQuit* still remain unchanged. The rules (5) and (6) only copy them from A to C. To refine those action words we need other refinement machines and composition rules.

After the composition machine is created, the test case generation is straightforward. Each path in the composition machine will become a GUI test case, which is a sequence of keywords. For our example, one possible test case might be \{*kwClickCreate*, *kwSelectSquare*, *kwSelectCircle*, *kwClickReset*\}, which translates to \{*create*, *square*, *circle*, *reset*\).

### 3.7.2 Hierarchical Finite State Machines

Paiva *et al.* [36] use the hierarchy of GUI dialogs to create a hierarchical state-machine model for testing. In particular, the GUI is modeled as a hierarchy of FSMs whose vertices can either represent single states or groups of states in the original FSM. The model consisting of these FSMs is called a Hierarchical Finite State Machine (HFSM).

The hierarchy is based on GUI dialogs. Consider a GUI represented by k dialogs \(D_1, D_2, \ldots, D_k\) which manipulate a set of variable \(V: V = \{v_1, \ldots, v_{|V|}\}\). From the complete FSM of the application, the tester manually specifies the state machine \(F_i\) for each dialog \(D_i\). Given the \(FSM_{D_i}\) for a dialog \(D_i\), it is possible to deduce the variables manipulated that dialog. A variable \(v_i\) is *written* by (or is affected by) a dialog \(D\) if there is a transition in \(FSM_D\) that changes the value of \(v_i\). A variable \(v_i\) is *read* by (or influences the behavior of) a dialog \(D\) if at least one of the following conditions holds:

1. there are two transitions \(T\) and \(T'\) in \(FSM_D\) and a variable \(v_k\) in \(V\) (not necessarily \(i \neq k\)) such that: (i) the source states of \(T\) and \(T'\) are different only in the value of \(v_i\); (ii) \(T\) and \(T'\) have the same triggering action (name and arguments); (iii) the destination states of \(T\) and \(T'\) have different values of \(v_k\); and (iv) at least one of the transitions (say \(T\)) changes the value of \(v_k\);
2. there are two states \( S \) and \( S' \) and a transition \( T \) with source \( S \) in \( FSM_D \) such that: (i) \( S \) and \( S' \) are different only in the value of \( v_i \); (ii) there is no transition \( T' \) with source \( S' \) and the same action as \( T \).

Let \( PFSM_{Di} \) be the projection of \( FSM_{Di} \) onto the variables manipulated by dialog \( D_i \) then we can use \( PFSM_{Di} \) to describe the internal behaviors of \( D_i \). Also from \( PFSM_{Di} \), it is possible to reconstruct \( FSM_{Di} \) by taking the union of the instances of \( PFSM_{Di} \) for all possible combinations of variable values that are not manipulated by it.

Using the notation of PFSMs, the original state machine can be organized into a 3-level HFSM:

1. The top level is an abstract FSM representing the relationships between independent dialogs.
2. The intermediate level is a set of projected FSMs representing internal behaviors for each dialog.
3. The bottom level is a complete FSM representing the behaviors of the entire GUI.

Considering the Radio Button Demo application, and its GUI states represented by a length 4 vector \{log, exitWinOpen, created, shape\} as done in Section 3.1.1, a tester may specify a subFSM for the main window (dialog \( D_{Main} \)) to include all states where \( exitWinOpen \) is set to 0 and the transitions between them. The other states make up the subFSM for the Exit Confirmation window (dialog \( D_{Exit} \)). Figure 18(c) shows the complete FSM (bottom level) for the application. The states are organized into two regions (enclosed by dashed lines) corresponding to two subFSMs. Note that the same full FSM was previous shown in Section 3.1.1, except that its layout has changed.

We can infer that that \textit{created} and \textit{shape} are two variables manipulated by \( D_{Main} \) while \textit{log} is the only variable manipulated by \( D_{Exit} \). Neither \( D_{Main} \) nor \( D_{Exit} \) manipulates \textit{exitWinOpen}. Using this analysis, the top level and the intermediate level of the HFSM can be constructed as shown in Figure 18(a) and Figure 18(b).
Two dialogs are independent if the set of variables written by one dialog is disjoint from the set of variables manipulated (read or written) by the other. In this case, instead of testing the complete FSM we only need to consider their PFSMs individually. In other words, those dialogs do not need to be tested very time there is a change on variables they do not depend on. To test a dialog $D$, the variables not manipulated by $D$ are fixed to a particular value and the test cases are generated using the PFSM of $D$.

Applying this strategy to test the Radio Button Demo’s GUI, we first realize that $D_{Main}$ and $D_{Exit}$ are two independent dialogs. So we can test $D_{Main}$ by fixing $log = 0$ ($exitWinOpen$ is already fixed) and generate test case in the PFSM$_{Main}$. Similarly, to test $D_{Exit}$ we fix $created = 0$ and $shape = C$. Two transiting actions $exit$ and $no$ also need to be tested once by fixing $created = 0$, $shape = C$ and $log = 0$. Instead of testing all possible paths of the FSM in Figure 18(c), we now only need to examine those in bold.

3.7.3 UML Diagram-based

As seen in previous sections, using formal models to represent GUIs makes it possible to systematically generate and analyze test cases. However, these models are often not intuitive, causing difficulties for test designers who are not familiar with formal Computer Science concepts. Paiva et al. [35] builds another visual layer on top of formal models to assist testers. The GUI is modeled using familiar UML notations and then automatically translated to the underlying formal model by tools. More specifically, the formal model is a set of FSMs which are encoded in a specification language called Spec# (an extension of the C# programming language) [8].

The GUI behaviors are specified by 4 UML diagrams: use case diagram, activity diagram, class diagram and state machine diagram. Those diagrams are enriched with additional stereotypes to enable automatic transformation from the visual forms to Spec# code.

*Use case diagram* provides an overview of the main functionalities and features of the GUI application. They describe the scenarios in which the GUI is used. The use case
diagrams are used to support designing other UML diagrams. However, there is no formal Spec# code directly generated from those diagrams. Figure 19 shows a use case diagram one might design for the Radio Button Demo example. The diagram consists of three main use case Edit shape, Reset, and Exit corresponding to three main scenarios the user may
interact with the GUI.

**Activity diagram** describes the business logic of use cases. The conditions and steps in the diagrams are directly encoded in Spec# syntax. Besides the user steps, they may have parameters that correspond to user inputs, pre/post-conditions (describing use case intent) and assertions. Figure 20 illustrates an activity diagram for the *Edit shape* use case in Figure 19. The diagram is shown on the left and the corresponding Spec# code translation is shown on the right. In this diagram, there are two activities *selectShape* and *create* which can be performed in parallel to complete the use case. The diagram also requires that the application should start before any activity is performed and at the end. The shape to be created is passed through the method parameter.

**Class diagram** describes the static structure of the GUI. Each top-level window is modeled as an object. The state variables are represented by class variables, while the interactive controls are represented by class methods. For GUI testing, several annotations were defined to give special meaning to the UML elements. An example of class diagram for our running example is shown in Figure 21. As we can see, two windows are annotated as modal windows. The *MainWnd* window is the initial window and from it we navigate to the *Exit-ConfirmationDlg* window. The methods under the «Action» annotation model the effect of a user action on the GUI state while the methods under «Probe» annotation model the ob-

![Figure 19: Use case diagram.](image-url)
void editShape(String shape) 
requires ApplicationStarted 
ensures Created 
&& getShape()==shape 
{
parallel{
  selectShape(shape);
  create();
}
}

Figure 20: Activity diagram for the Edit shape use case.

Observations of some GUI states by the users. Those annotations are then directly transferred to the annotations in Spec# code.

State machine diagram describe the dynamic reactive behaviors of the GUI. The diagrams show GUI states at different levels of abstraction, the user actions available at each state, their effects on the GUI states, and therefore the sequences of user actions. Each state of the state machine can be formalized by a Boolean condition on the state variables. Each transition has a triggering event that is the call of a method representing a user action. The
transitions may additionally have pre- and post-conditions on state variables and method parameters. A set of rules are developed to translate the state machine diagrams into Spec# code. Figure 22 illustrates a state machine diagram one might specify for the Radio Button Demo’s GUI (left) and the translated Spec# code for two transitions create and selectShape (right). The code translation is done automatically using the rules specified in Table 3.7.3.

After the formal specifications (e.g., Spec# code) are generated for all UML diagrams, an analyzer tool (e.g., Spec Explorer) is used to analyze the formal models and generate test cases for each diagram accordingly. For example, in Figure 22 the requires statement of the reset method and the ensures statement of the create method are the same. So, to test the reset method (hence the reset transition in the state machine) the tool might require the create method to be executed before the reset call generating a test case ⟨create, reset⟩ for the state machine diagram.

![State machine diagram and its Spec# translation.](image)

```
[Action] void create()
requires !isCreated && !ShapeOnScreen
ensures isCreated && ShapeOnScreen;
{...}

[Action] void reset()
requires isCreated && ShapeOnScreen
ensures !isCreated && !ShapeOnScreen;
{...}

[Action] void selectShape ()
requires (!ShapeOnScreen &&!isCreated)
||((ShapeOnScreen &&isCreated)
ensures (( !ShapeOnScreen &&!isCreated
)&&(!isCreated)
&&((ShapeOnScreen &&isCreated
))==> (isCreated
&& shapechanged))
{...}
```
<table>
<thead>
<tr>
<th>Rule</th>
<th>State machine transitions</th>
<th>Spec# code</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1 Simple transition</td>
<td><img src="#" alt="Simple transition diagram" /></td>
<td><img src="#" alt="Spec# code" /></td>
</tr>
<tr>
<td>R2 Multiple transitions</td>
<td><img src="#" alt="Multiple transitions diagram" /></td>
<td><img src="#" alt="Spec# code" /></td>
</tr>
</tbody>
</table>

Table 3: Transition rules from state machine specification to Spec# code.

4 Conclusions

Graphical user interfaces are by far the most popular means used to interact with software today. Unfortunately, the state-of-the-practice in GUI testing has not kept pace with the rapidly evolving GUI technology. In practice, GUI testing is largely manual, often resulting in inadequate testing. There have been several research efforts to improve GUI testing. This chapter presented some of the recent advances in automated model-based GUI testing. It also provided the first detailed taxonomy of these techniques. A small GUI application was used as a running example to demonstrate each technique.

In its very fundamental form, the goal of GUI testing is to determine whether the GUI executes as expected, as documented in the specifications, or as required by the intended user. This definition is very broad and may encompass factors such as testing the GUI’s usability, correctness, and performance. Since GUI testing is a multifaceted problem, no one technique can be used for GUI testing; in fact, in practice, a collection of techniques are almost always used.

Finally, the GUI interaction testing problem can be viewed as a search problem with the state space of the GUI being the search space and the objective of the search to find errors. Since the number of events that a user may perform on the GUI at any given time...
is very large, the search space is extremely large (even infinite in most cases). Exhaustively traversing the search space is impractical in such cases. The field of GUI testing remains ripe for the application of upcoming areas of research, such as search-based software engineering.

References


