A COMPREHENSIVE FRAMEWORK FOR TESTING GRAPHICAL USER INTERFACES

by

Atif M. Memon

B.C.S., Computer Science, University of Karachi, 1991

M.C.S., Computer Science, K.F.U.P.M., Dhahran, 1995

Submitted to the Graduate Faculty of
Arts and Sciences in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

University of Pittsburgh

2001

UNIVERSITY OF PITTSBURGH

FACULTY OF ARTS AND SCIENCES

This dissertation was presented

by

Atif M. Memon

It was defended on

July 26, 2001

and approved by

Prof. Mary Lou Soffa (**co-advisor**)

Prof. Martha Pollack (**co-advisor**) (University of Michigan)

Prof. Rajiv Gupta (University of Arizona)

Prof. Adele E. Howe (Colorado State University)

Prof. Lori Pollock (University of Delaware)

Committee Chairperson(s)

A COMPREHENSIVE FRAMEWORK FOR TESTING GRAPHICAL USER INTERFACES

Atif M. Memon, Ph.D.

University of Pittsburgh, 2001

The widespread recognition of the usefulness of graphical user interfaces (GUIs) has established their importance as critical components of today's software. Although the use of GUIs continues to grow, GUI testing has remained a neglected research area. Since GUIs have characteristics that are different from those of conventional software, such as user events for input and graphical output, techniques developed to test conventional software cannot be directly applied to test GUIs. This thesis develops a unified solution to the GUI testing problem with the particular goals of automation and integration of tools and techniques used in various phases of GUI testing. These goals are accomplished by developing a GUI testing framework with a GUI model as its central component. For efficiency and scalability, a GUI is represented as a hierarchy of components, each used as a basic unit of testing. The framework also includes a test coverage evaluator, test case generator, test oracle, test executor, and regression tester. The test coverage evaluator employs hierarchical, event-based coverage criteria to automatically specify what to test in a GUI and to determine whether the test suite has adequately tested the GUI. The test case generator employs plan generation techniques from artificial intelligence to automatically generate a test suite. A test executor automatically executes all the test cases on the GUI. As test cases are being executed, a test oracle automatically determines the correctness of the GUI. The test oracle employs a model of the expected state of the GUI in terms of its constituent objects and their properties. After changes are made to a GUI, a regression tester partitions the original GUI test suite into valid test cases that represent correct input/output for the modified GUI and invalid test cases that no longer represent correct input/output. The regression tester employs a new technique to reuse some of the invalid test cases by repairing them. A cursory exploration of extending the framework to handle the new testing requirements of web-user interfaces (WUIs) is also done. The framework

has been implemented and experiments have demonstrated that the developed techniques are both practical and useful.

Acknowledgements

I would like to thank my parents whose constant efforts, encouragement and hard work made achieving the goal of obtaining a Ph.D. possible.

I thank all my teachers in schools, colleges, and universities whose dedication and hard work helped lay the foundation for this work. Special thanks to Dr. Subbarao Ghanta who helped develop my initial interest in research, showed me an example of a truly dedicated researcher and a wonderful person.

I am greatly indebted to my exceptional thesis advisors, Prof. Mary Lou Soffa and Prof. Martha Pollack, for their advice, support and encouragement throughout this dissertation. They taught me how to reason about important problems and present my ideas. I thank the members of my dissertation committee Rajiv Gupta, Adele E. Howe, and Lori Pollock for their help and advice.

This dissertation greatly benefited from discussions with and comments from Brian Malloy (Clemson University), Mary Jean Harrold (Georgia Tech.) Somesh Jha (Univ. of Wisconson), David Kasik (Boeing), Michael Ernst (MIT), Alberto Savoia (Velogic), Jean Hartmann (Siemens), and Sadik Esmelioglu (Lucent). Thank you for all your suggestions.

Special thanks to Dr. Edward Miller and Guillermo Sandoval from Software Research Inc. for providing me with a free license of their testing tools, which helped me gain a better understanding of the state-of-the-art in testing technology.

My stay at Pitt was made more enjoyable because of great colleagues, especially Tarun Nakra, Clara Jaramillo, Ras Bodik, Yasir Khalifa, and Majd Sakr.

Thank you, Bob Hoffman for solving my many tech related problems, Debbie Holzhauser and Loretta Shabatura for solving all other graduate school and administrative problems.

I would like to thank my loving wife, Vidya, for always being there to support me and be a constant source of encouragement during my Ph.D. She taught me to always look at the positive side of things, to stop and smell the roses once in a while, to be contented and happy.

Family and friends have played an important role in the completion of this dissertation. Special thanks to Aanand, Laxmi, Safiullah, Neaz, Parthasarathy Mama, Chitra, Kashif, Sadaf, Imran, and of course the kids, for all their love.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Graphical user interfaces (GUIs) have become nearly ubiquitous as a means of interacting with software systems. GUIs make software easy to use and, recognizing the importance of user-friendly software, today's software developers are dedicating an increasingly large portion of software code to implementing GUIs. A GUI is the front-end to underlying code (Figure 1.1), and a software user interacts with the software using the GUI. The user performs *events* such as mouse movements, object manipulation, menu selections, and opening and closing of windows. The GUI, in turn, interacts with the underlying code through messages and/or method calls. GUIs constitute as much as 45-60% of the total software code [49, 54, 56, 57, 52]. The widespread use of GUIs is leading to the construction of increasingly complex GUIs. Their use in safety-critical systems is also growing [92].

Although the use of GUIs continues to grow, GUI testing has remained a neglected research area. Adequately testing a GUI is required to help ensure the safety, robustness and usability of an entire software system [55]. Testing is, in general, labor and resource intensive, accounting for 50-60% of the total cost of software development [30, 64]. GUI testing is especially difficult today because GUIs have characteristics different from those of traditional software, and thus, techniques typically applied to software testing are not adequate. Current GUI testing techniques are incomplete, ad-hoc, and largely manual.

When testing the underlying code, the code for the GUI may also be tested. However, it is important to separate the testing of the GUI from that of the underlying code. Multiple GUIs and multiple versions of GUIs are increasingly being used as front-ends to the same underlying code. The increased use of mobile devices interacting with software places limitations on the capabilities of GUIs that are used with some of these devices [44]. Device restrictions such as display resolution may require that different interfaces be implemented to access the same underlying application, such as a web application. Also, security restrictions may require that restricted views of the same software be provided to users with different security privileges. For example, the GUI for the MS Windows

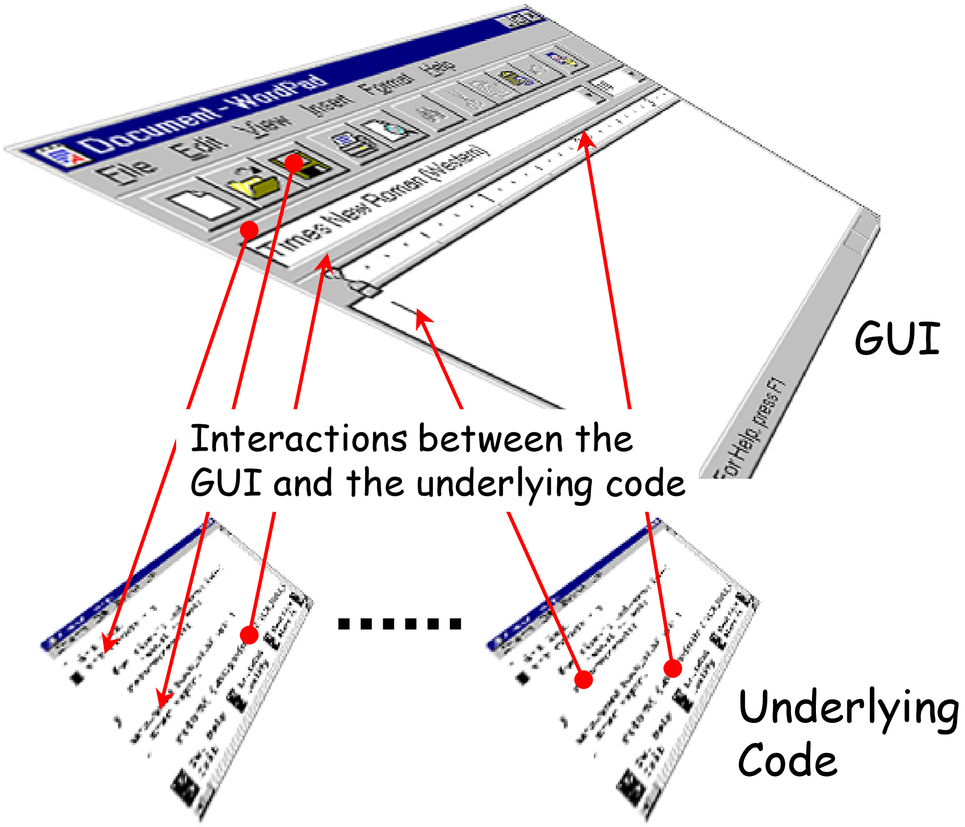

Document - WordPad
File Edit View Insert Format Help
Times New Roman (Western)
For Help, press F1
GUI
Interactions between the GUI and the underlying code
......
Underlying Code

Figure 1.1: The GUI is the Front-end to Underlying Code.

2000 control panel of a system administrator has many more features than that of an ordinary user. Finally, the increased use of customizable interfaces provides different views to the same underlying code. A common example is customizable tool-bars available in most of today's software. By separately testing the underlying code (employing code-based testing techniques) and separately testing each GUI (employing GUI testing techniques), the final software can be composed by plugging-in the appropriate GUI as demanded by the application.

The focus of this research is to develop techniques and tools to test GUIs. Before designing such tools, it is important to describe the GUI testing process. The next section presents the steps that a test designer must perform for GUI testing.

1.1 GUI Testing Steps

Although GUIs have characteristics, such as user events for input and graphical output, that are different from those of conventional software and thus require the development of different testing techniques, the overall process of testing GUIs is similar to that of testing conventional software. The testing steps for conventional software, extended for GUIs, follow:

- *Determine what to test*

  During this first step of testing, *coverage criteria*, which are sets of rules used to determine what to test in a software, are employed. In GUIs, a coverage criterion may require that each event be executed to determine whether it behaves correctly.

- *Generate test input*

  The test input is an important part of the test case and is constructed from the software's specifications and/or from the structure of the software. For GUIs, the test input consists of events such as mouse clicks, menu selections, and object manipulation actions.

- *Generate expected output*

  Test oracles generate the expected output, which is used to determine whether or not the software executed correctly during testing. A *test oracle* is a mechanism that determines whether or not the output from the software is equivalent to the expected output. In GUIs, the expected output includes screen snapshots and positions and titles of windows.

- *Execute test cases and verify output*

  Test cases are executed on the software and its output is compared with the expected output. Execution of the GUI's test case is done by performing all the input events specified in the test case and comparing the GUI's output to the expected output as given by the test oracles.

- *Determine if the GUI was adequately tested*

  Once all the test cases have been executed on the implemented software, the software is analyzed to check which of its parts were actually tested. In GUIs, such an analysis is needed to identify the events and the resulting GUI states that were tested and those that were missed. Note that this step is important because it may not always be possible to test in a GUI implementation what is required by the coverage criteria.

After testing, problems are identified in the software and corrected. Modifications then lead to regression testing, i.e., re-testing of the changed software.

- *Perform regression testing*

  Regression testing is used to help ensure the correctness of the modified parts of the software as well as to establish confidence that changes have not adversely affected previously tested parts. A regression test suite is developed that consists of (1) a subset of the original test cases to retest parts of the original software that may have been affected by modifications, and (2) new test cases to test affected parts of the software, not tested by the selected test cases. In GUIs, regression testing involves analyzing the changes to the layout of GUI objects, selecting test cases that should be rerun, as well as generating new test cases.

Any GUI testing method must perform all of the above steps. Currently, GUI test designers typically rely on record/playback tools to test GUIs [83, 32]. The process involved in using these tools is largely manual, making GUI testing slow and expensive.

Automated GUI testing techniques for each of the above steps are needed in order to efficiently and effectively test GUIs. One approach is to develop independent tools and techniques to automate each GUI testing step. There have been several research efforts at designing some automated tools, for example, using finite-state machine (FSM) models to generate test cases [14, 13, 21, 8], programming the test case generator [43], and using a Latin square method for regression testing [90]. Although such independent tools are useful for automating some aspects of GUI testing, they do not address all aspects. A test designer who makes use of these independent tools will need to learn the various idiosyncrasies of each tool. Moreover, since these tools are independently developed, they may not be compatible. Hence, in practice, it may be difficult to use these tools in a testing problem.

This thesis takes an alternative approach: developing a comprehensive *GUI testing framework* that includes techniques and tools to perform all of the GUI testing steps. The goals for the framework and its testing techniques are:

- All the techniques must be **integrated**, employing a common representation so that results of one tool are compatible with the others.

- The GUI testing tasks should be as **automated** as possible so that the test designer's work is simplified.

- The overall testing cycle defined by the techniques should be **efficient** since software testing is usually a tedious and expensive process. Inefficiency may lead to frustration and abandonment of the techniques.

- The techniques should be **robust**. Whenever the GUI enters an unexpected state, the testing algorithms should detect the error state and report all information necessary to debug the GUI.

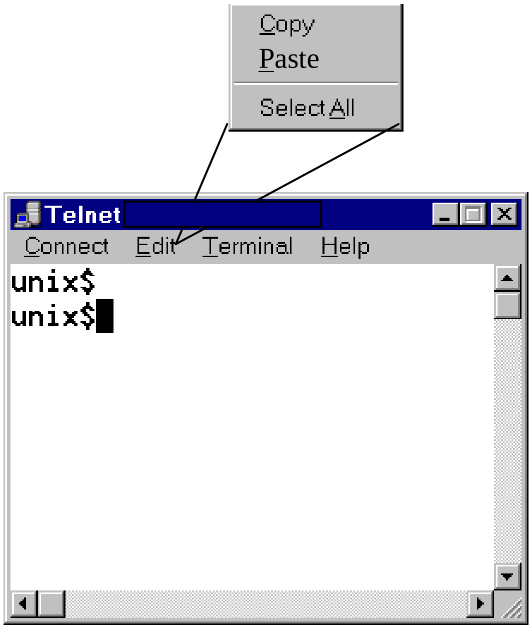
- The tools/techniques should be **portable**. Test information (e.g., test cases, oracle information, coverage report, and error report) generated and/or collected on one platform should be usable on all other platforms on which the GUI can be executed.

- Finally, the techniques should be **general** enough to be applicable to a wide range of GUIs.

1.2 Challenges of Developing a GUI Testing Framework

Developing a GUI testing framework with the above goals offers a number of challenges. First, a **representation** of a GUI must be created that can be used across the various techniques and tools. A representation must be developed at a sufficiently high level of abstraction that it effectively captures the GUI events and their interactions and is general enough to be applicable to a wide variety of GUIs. Yet, the same representation must capture sufficient low level details of the GUI to enable a test oracle to verify the correctness of the GUI. An additional challenge for the representation is scalability; GUIs are large, containing huge bit-maps and a large number of events. If the representation is not scalable, then all phases of testing that employ it will also fail to scale.

For conventional software, **coverage** is evaluated using the amount and type of underlying code tested. Traditional coverage criteria may not work well for GUI testing, because what matters is not only how much of the code is tested, but whether the tested code corresponds to potentially problematic user interactions. Consider the example of a *Telnet* application's `Edit` menu shown in Figure 1.2. Traditional code-based coverage criteria evaluate the amount of underlying code tested. GUIs and the underlying code are conceptually at different levels of abstraction. Therefore, it is difficult to obtain a mapping between GUI events and the underlying code. If code-based coverage criteria are used when testing GUIs, then problematic event interactions might be missed. For example, in the absence of sufficient memory, the events `Edit + Copy` generate a memory error but allow the user to continue after closing the error window. If the user continues to use the application, another `Edit + Copy` results in a system crash. If traditional code-based coverage criteria are employed, it may be difficult to test the code for such an interaction. This example illustrates that it is important to develop coverage criteria based on user events.

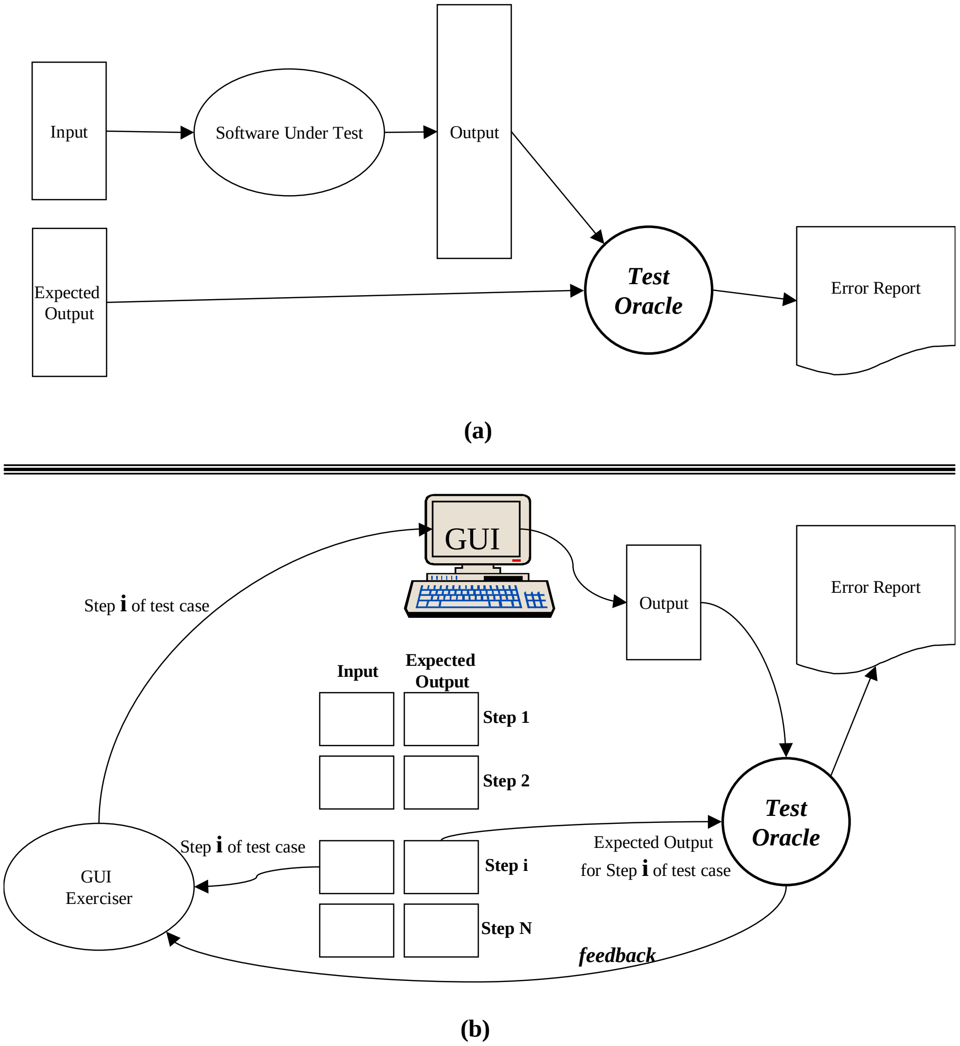


```
unix$
unix$
```

Figure 1.2: A Telnet Application's GUI

A third challenge is that even though the coverage criteria may help focus on specific parts of a GUI, it may be impractical to generate all possible **test cases** for these selected parts. A subset of these test cases must be generated for testing. The subset selection decision may have to be made by the test designer during test case generation. Another problem related to test case generation is called the *controllability problem*, i.e., bringing the GUI to a state in which a test case may be executed on it [12]. For each test case, appropriate events may need to be performed on the GUI to bring it to the desired state.

Fourth, **test oracles** for GUIs are different from those for conventional software. Test oracles determine whether or not the software executed correctly during testing. In conventional software testing, the test oracle is invoked after the end of test case execution, as shown in Figure 1.3(a). The test case is executed by the software, and the final output is compared with the expected output. In contrast, GUI test case execution, shown in Figure 1.3(b), requires that the test oracle invocation and test case execution be interleaved because an incorrect GUI state can lead to an unexpected screen. This screen may make further execution of the test case useless since events in the test case may not match any button on the GUI screen. Thus, execution of the test case should be terminated as soon as an error is detected. Also, if verification is not done after each step of test case execution, it may become difficult to pinpoint the actual cause of the error since in some cases the final



(a)

(b)

Figure 1.3: Comparing the Test Case Execution of (a) Conventional Software, and (b) GUIs.

output may be correct whereas the intermediate outputs may be incorrect. Consequently, in GUI test case execution, the inputs are given one step at a time, and the expected output is compared with the GUI's output after each step. This interleaving of verification and test case execution makes GUI testing more complex because (1) the expected output needs to be generated for each event, and (2) the correctness of the GUI is checked after each event is executed.

Finally, **regression testing** presents special challenges for GUIs. Both inputs and outputs to a GUI depend on positions of graphical elements on the screen. The input-output mapping may not remain constant across successive versions of the software [53]. Movement of buttons, changes in the bit-maps, and organization of menus may render older test cases useless. Moreover, the expected output used by the test oracles may become obsolete.

Regression testing is especially important for GUIs as they are typically designed using *rapid prototyping* [53]. The GUI software is modified and tested on a continuous basis. Efficient regression testing mechanisms are needed to detect the frequent modifications to the GUI and adapt the old test cases.

1.3 GUI Testing Framework

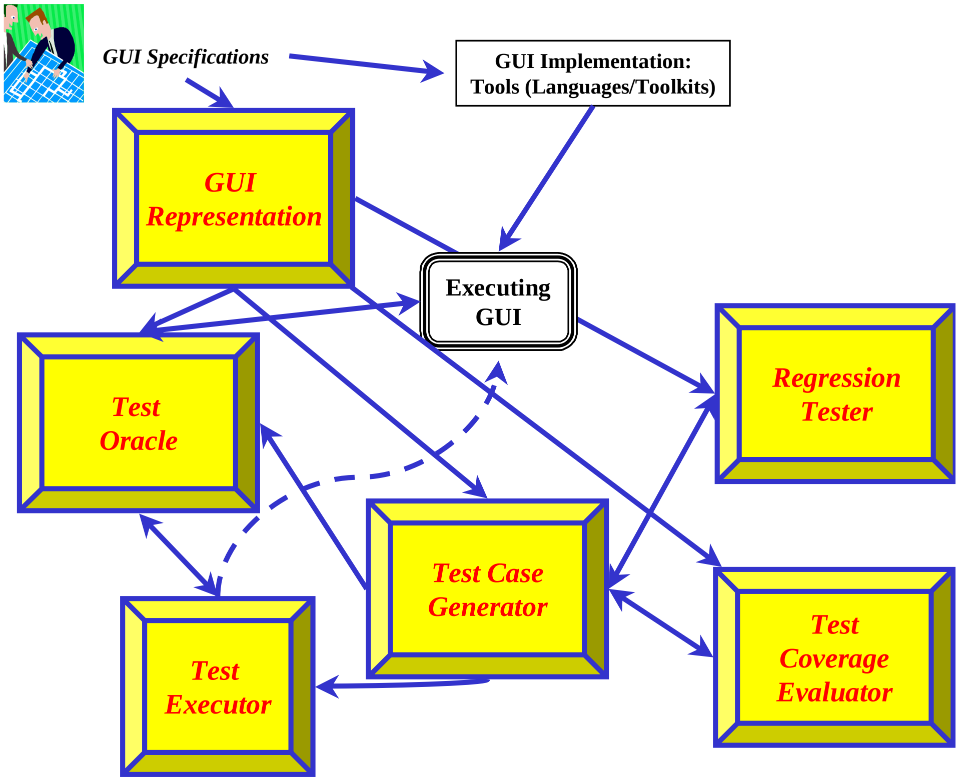


Figure 1.4: An Overview of the GUI Testing Framework.

This dissertation presents the design and implementation of a comprehensive framework for testing GUIs. As shown in Figure 1.4, the framework consists of several interacting components: a GUI representation, test coverage evaluator, test case generator, test oracle, test executor, and regression tester. These components are briefly described next.

1. A GUI is **represented** as a set of objects, a set of properties of those objects, and a set of events that change the properties of certain objects. For efficiency and scalability, the GUI is decomposed into a hierarchy of components that is used by the test case generator, coverage evaluator, test oracle, and regression tester.

2. The **coverage evaluator** employs a new class of coverage criteria called *event-based coverage criteria*. These criteria use events and event sequences to specify a measure of test adequacy. The coverage evaluator employs (1) intra-component criteria for events within a component and (2) inter-component criteria for events across components.
3. The **test case generator** is based on a new algorithm that exploits **planning** [87, 86], a well-developed and used technique in artificial intelligence (AI). The motivating idea is that GUI test designers will often find it easier to specify typical goals that users of their software might have than to specify sequences of GUI events that users might perform to achieve those goals. Given a specification of initial and goal states for a GUI, a planner is used to generate "plans" that become test cases for the GUI.
4. The GUI **test oracle** employs the GUI representation and, for each test case, automatically derives the expected state for each event in the test case. The actual state of an executing GUI is also represented in terms of objects and their properties derived from the GUI's execution. Using the actual state acquired from an *execution monitor*, the oracle automatically compares the expected and actual states after each event to verify the correctness of the GUI for the test case.
5. Execution automation is achieved by designing/implementing an automated **test executor**. Test cases (that may be generated off-line by the test case generator) are input to the test executor, which executes each event in the test case. The test executor generates physical events, such as mouse and keyboard events, thereby mimicking a GUI user.
6. The **regression tester** partitions the original GUI test suite into *valid test cases* that represent correct input/output for the modified GUI and *invalid test cases* that no longer represent correct input/output. The regression tester employs a new technique to reuse some of the invalid test cases by *repairing* them. The repaired test cases are more likely to reveal faults in the modified GUI since they test specific sequences of events that were affected by modifications.

All the above components of the GUI testing framework have been implemented as part of this dissertation. The GUI testing framework was used to test a newly implemented word processor, which is similar to Microsoft's WordPad (except for the **Help** menu, which was not modeled). WordPad was chosen because it has a moderately complex GUI, containing events that are common across many GUIs. For example, WordPad contains editing events such as cut, copy, and paste; file events used to open, and save files; various dialog types such as modal and modeless dialogs; complex functions to find and replace text. On the other hand, the WordPad GUI contains text objects that are straightforward to represent. It is expected that results of experiments performed on WordPad will also hold for most of today's GUIs. The entire WordPad software can be implemented by one person in a reasonable amount of time. Moreover it is widely used and most readers are familiar with its functionality. In this dissertation, when scaling issues are discussed, the much larger GUI of MS Word is also considered. Details of the WordPad software and testing algorithms are presented in subsequent chapters.

The next chapter provides the background necessary to understand the context and details of the techniques developed in this dissertation. Chapter 3 presents the GUI representation that is employed by all the other components of the framework to perform their respective tasks. In Chapter 4, the coverage evaluator is described that employs new event-based coverage criteria to help determine whether a GUI has been adequately tested. Chapter 5 describes the design of the test case generator that employs AI plan generation techniques. The design of test oracles that verify the correctness of the GUI as it is being tested is presented in Chapter 6. Chapter 7 presents a new method of performing regression testing by repairing existing test cases. Chapter 8 explores how the framework may be extended to test web-user interfaces. Finally, Chapter 9 concludes with a discussion of the merits of this research and possible future directions.

Chapter 2

Background and Related Work

The research presented in this dissertation focuses on developing a testing framework for GUIs and thus spans the areas of testing environments, test coverage criteria development, test case generation, test oracles, and regression testing. This chapter introduces the relevant terms, presents the background and prior related research in each of these areas. Since GUI testing is still in its infancy, very little research has been done in this area. However, there is a potential to use techniques from general software testing and tailor them for GUI testing. Hence, in each subsequent section, some of the terms and approaches used for testing non-GUI software are described, and their possible adaptation for GUI testing is discussed. The approaches used to automate some aspects of testing are also presented. Among them, AI planning has been used to automate test case generation; a brief description of the system that uses planning for test case generation, and a detailed discussion of AI planning is presented. Subsequent sections present the background and related work in testing environments, test coverage, test case generation, test oracles, regression testing, and AI planning.

2.1 Testing Environments

Ostrand et al. [58] present the design of the only environment for GUI testing reported in the available literature. Their *visual test development environment* (TDE) links a test designer, a test design library, and a test generator to a capture/replay tool. By using this environment, the test designer captures sequences of interactions with the GUI and visually modifies them. However, most of the tasks are done manually, except for minimal support for modeling the GUI and using the model to tailor regression tests. A test designer creates a GUI model consisting of a top-level graph with representations for individual windows. *Data variations* and *path variations* are introduced by the test designer to create multiple test cases. Ostrand et al. indicate the need to develop a facility for defining result comparison actions in test scenarios using which the test designer can augment test scripts with oracles to check the state of the GUI.

2.2 Test Coverage

An important question in testing is, "what constitutes an adequate test suite?" This question, posed by Goodenough and Gerhart in 1975, was declared as the central question of software testing [27]. Since then, much research has been done to define test coverage, resulting in the development of several dozen criteria.

Coverage criteria are sets of rules used to help determine whether a test suite has adequately tested a program and to guide the testing process. The most well-known coverage criteria are statement coverage, branch coverage, and path coverage. Zhu et al. provide a comprehensive survey of existing test coverage criteria [97]. One classification of coverage presented therein is based on the source of information used to specify the testing requirements. This classification defines a coverage criterion as either *specification based*, *program based*, or *interface based*. Of interest to this research are interface based coverage criteria that specify testing requirements in terms of the type and range of software input without reference to any internal features of the program code or the specifications. Developing interface based coverage criteria remains an open area for research.

None of the test coverage criteria surveyed by Zhu et al. are directly applicable to GUI testing. In fact, almost no research has been reported on developing coverage criteria for GUIs. The only exception is the work by Ostrand et al., mentioned in Section 2.1, which briefly indicates that a *model-based method* may be useful for improving the coverage of a test suite [58]. However, this prior research deferred a detailed study of the coverage of the generated test cases using this type of GUI model to future work. In practice, since there are no well-established coverage criteria for GUIs, *ad hoc* techniques are employed. One example criterion is "stop testing when no more than 50 new defects are found per 1,000 test hours" [82].

There is a close relationship between test-case generation techniques and the underlying coverage criteria used. Much of the literature on GUI test case generation focuses on describing the algorithms used to generate the test cases; little or no discussion about the underlying coverage criteria is presented [79, 91, 38]. The next section presents a discussion of some of the test case generation techniques.

2.3 Test Case Generation

Test cases contain the input supplied to the software being tested. For example, a test case for a GUI software system may contain a sequence of mouse events. Techniques for generating test cases depend on the type of testing being conducted. The test case generation technique developed in this dissertation employs a model of the GUI derived from its specifications (as opposed to its code), i.e., a type of *black-box testing*. The remainder of this section first presents some GUI test case generation techniques, their limitations and shortcomings. Then it describes some black-box testing techniques that may be applicable to GUI testing.

Currently, test designers rely on record/playback tools to create test cases for GUIs [83, 32]. The test designer interacts with the GUI, generating mouse/keyboard events. The record tool captures these events and GUI screens during the interactive session; these recorded sessions are later played back whenever it is necessary to recreate the same events. This process is extremely labor intensive. A higher level of support is provided by programming the test case generator [43]. For comprehensive testing, programming requires that the test designer code all possible decision points in the GUI. However, this approach is time consuming and is susceptible to missing important GUI decisions. A popular alternative to performing rigorous, expensive, in-house testing is to release large number of *beta copies* of the software and let the users do part of the testing. For example, Microsoft did part of its testing of its Windows '95 software by releasing almost 400,000 beta copies [38].

A number of research efforts have addressed the automation of test case generation for GUIs. Several finite-state machine (FSM) models have been proposed to generate test cases [14, 13, 21, 8]. In this approach, the software's behavior is modeled as a FSM where each input triggers a transition in the FSM. A path in the FSM represents a test case, and the FSM's states are used to verify the software's state during test case execution. This approach has been used extensively for test generation of hardware circuits [31]. An advantage of this approach is that once the FSM is built, the test case generation process is automatic. It is relatively easy to model a GUI with a state machine model; each user action leads to a new state, and each transition models a user action. However, a major limitation of this approach, which is an especially important limitation for GUI testing, is that state machine models have scaling problems [79]. To aid in the scalability of the technique, variations such as variable finite state machine (VFSM) models have been proposed by Shehady et al. [79].

Test cases have also been generated that mimic novice users [38]. This approach relies on an expert to first manually generate a sequence of GUI events, and then uses genetic algorithm techniques [23, 24] to modify and lengthen the sequence, thereby mimicking a novice user. The assumption is that experts take a more direct path when solving a problem using GUIs whereas novice users often take longer paths. Although useful for generating multiple test cases, the technique relies on an expert to generate the initial sequence. Consequently, the final test suite depends largely on the paths taken by the expert user. Another problem with this approach is that it assumes that novices' interactions with the GUI randomly diverge from those of experts.

White et al. present a new test case generation technique for GUIs [91]. This technique also requires a substantial amount of manual work on the part of the test designer. The test designer/expert manually identifies a responsibility, i.e., a GUI activity. For each responsibility, a machine model called the "complete interaction sequence" (CIS) is identified manually.

Avritzer et al. [3] have proposed a technique for software *load testing*, which has characteristics that may be relevant to GUI testing. This technique assesses how the system performs under a given load. The goal of this technique is to generate test cases to test the software's resource allocation strategies rather than its functionality. Load testing is done after the software has been thoroughly tested for correctness of functionality. The test case generation process uses an *operational profile* that describes the expected workload of the software once it is operational. The operational profile consists of the number and types of inputs to the software, the probability distribution of each type of input, and the average input arrival rate. This type of testing is attractive for GUIs since it is possible to obtain similar profiles from user sessions recorded during usability testing. However, a major limitation of this technique is that the software has to be represented by a Markov chain model. GUIs have a large number of states, and a state description that encodes a sequence of states may be impractical.

Donat [17] presents a technique for automatically transforming formal specifications into black-box test cases. The approach requires the specifications to be written in a predicate logic with quantification. The system generates *test frames*, i.e., structures that specify combinations of conditions corresponding to a single test step. Each test step demonstrates that a specified test requirement has been implemented. An important limitation of this approach is that the test designer has to manually refine the test frame into a test step by entering data values.

AI Planning has been found to be useful for generating focused test cases for a robot tape library command language [35]. The main idea is that test cases for command language systems are similar to plans. Given an initial state of the tape library and a desired goal state, the planner generates a "plan", which is executed on the software as a test case. Each command in the language is modeled as a planning operator. This approach works well for systems with a small command language. Since GUIs typically have a large number of operations such as menus, buttons, and windows, the approach needs to be extended to handle a large number of operators. The test case generator presented in this dissertation employs planning to generate test cases. Section 2.6 gives a brief introduction to planning and different planning techniques.

2.4 Test Oracles

Once test cases have been generated, they are executed on the GUI, and the GUI's output needs to be verified for correctness. A *test oracle* is a mechanism for determining whether or not the output from the GUI is equivalent to the expected output derived from the GUI's specifications.

Very few techniques have been developed to automatically generate the expected output for conventional software. Hence, software systems rarely have an automated test oracle [65, 70, 69, 16]. In most cases, the expected behavior of the software is assumed to be provided by the test designer. The expected behavior is specified by the test designer in the form of a table of pairs *(actual output, expected output)* [65], as temporal constraints that specify conditions that must not be violated during software execution [69, 15, 16, 70], or as logical expressions to be satisfied by the software [18]. This expected behavior is then used by the verifier by either performing a table lookup [65], FSM creation [36, 16], or boolean formula evaluation [18] to determine the correctness of the actual output.

Richardson in TAOS (Testing with Analysis and Oracle Support) [69] proposes several levels of test oracle support. One level of test oracle support is given by the `Range-checker` which checks for ranges of values of variables during test-case execution. A higher level of support is given by the `GIL` and `RTIL` languages in which the test designer specifies temporal properties of the software. Siepmann et al. in their TOBAC system [80] assume that the expected output is specified by the test designer and provide seven ways of automatically comparing the expected output to the software's actual output. A popular alternative to manually specifying the expected output is by performing reference testing [82, 85]. Actual outputs are recorded the first time the software is executed. The recorded outputs are later used as expected output for regression testing.

2.5 Regression Testing

Regression testing is an important software maintenance activity and can account for as much as one-third of the total cost of software production [67, 78, 6]. The goal of regression testing is to help ensure the correctness of the modified parts of the software as well as to establish confidence that changes have not adversely affected previously tested parts.

Although regression testing of conventional software has received a lot of attention [10, 73, 75, 76], there has been almost no reported research on GUI regression testing. The exception is White [90] who proposes a Latin square method to reduce the size of the 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. This strategy seems promising since it also employs GUI events. However, the technique needs to be extended to GUI items other than menus. Moreover, detailed studies need to be conducted to verify whether the pairwise interactions checking assumption is sufficient.

Several strategies for regression testing of conventional software have been proposed [4, 33, 71, 47]. One regression testing strategy proposes rerunning all test cases that have not become obsolete. Since this *retest-all strategy* is resource intensive, numerous efforts have been made to reduce its cost. *Selective retest techniques* [1, 7, 34] attempt to reduce the cost of regression testing by testing only selected parts of the software. These techniques have traditionally focused on two problems: (1) *regression test selection problem*, i.e., selecting a subset of the existing test cases [75], and (2) *coverage identification problem*, i.e., identifying portions of the software that require additional testing. Solutions to the regression test selection problem traditionally compare structural representations (e.g., control-flow graphs [75], control-dependence graphs [74]) of the original and modified software. Test cases that cause the execution of different paths in these structures are likely to be selected for re-testing. Among selective retest strategies, the *safe approaches* require the selection of every existing test case that exercises any program element that could be affected by a given program change. Although computationally less expensive than the retest-all strategy, safe approaches still make heavy demands on resources. At the other end of the spectrum of selective retest strategies are *minimization approaches* that attempt to select the smallest set of test cases necessary to test affected program elements at least once [77]. These techniques attempt to assure that some structural coverage criterion is met by the test cases that are selected. Practical strategies fall between the safe strategies

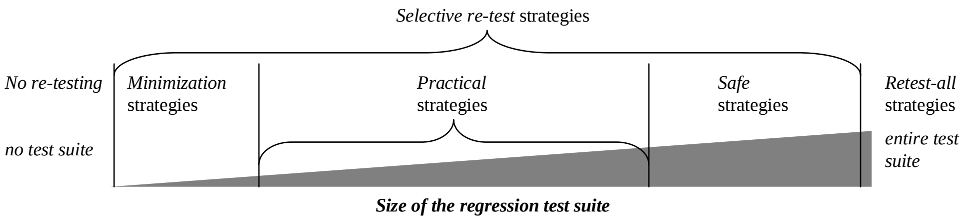


Figure 2.1: The Spectrum of Regression Testing Strategies.

and minimization strategies (see Figure 2.1). The test designer may be satisfied with using near-minimal sets of test cases [72].

Other regression testing techniques include analyzing changes in functions, types, variables, and macro definitions [71], using def-use chains [33], constructing procedure dependence graphs [9], and analyzing code and class hierarchy for object-oriented programs [47]. These techniques are not directly applicable to GUI regression testing because regression information is derived from changes made to the software's code. However, if a logical structure of the user event sequences can be constructed, then some of the ideas from these techniques may be applicable.

2.6 AI Plan Generation

Automated plan generation has been widely investigated and used within the field of artificial intelligence. Given an initial state, a goal state, a set of operators, and a set of objects, a planner returns a set of actions (instantiated operators) with ordering constraints to achieve the goal. Many different algorithms for plan generation have been proposed and developed. Weld presents an introduction to least commitment planning [86] and a survey of the recent advances in planning technology [87].

Formally, a planning problem $P(\Lambda, D, I, G)$ is a 4-tuple, where Λ 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

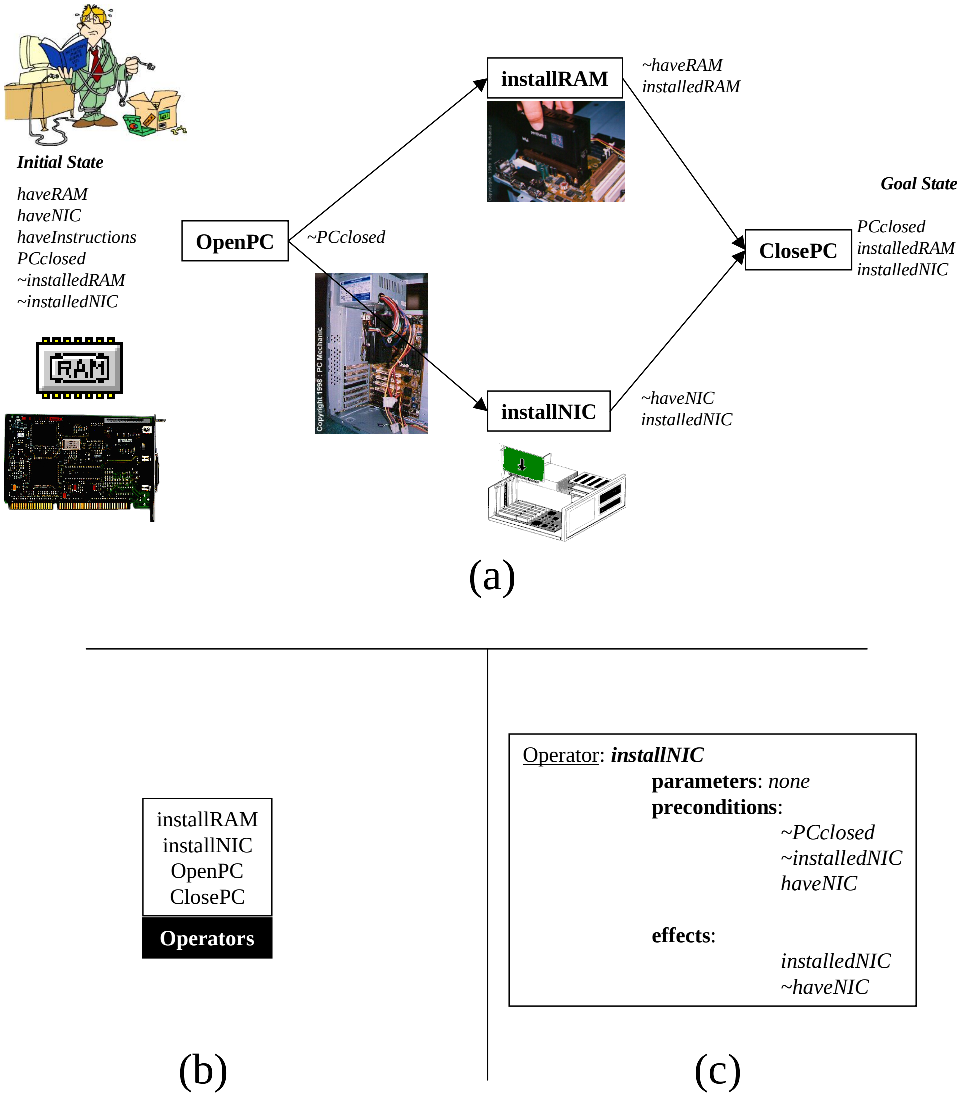


Figure 2.2: (a) A Plan to Install RAM and a Network Interface Card in the Computer, (b) The Operators Used in the Plan, and (c) Detailed Definition of the installNIC Operator.

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 $\neg c$ can temporally intervene between steps S_i and S_j.

Figure 2.2(a) shows an example plan for a problem in which memory (RAM) and a network interface card (NIC) need to be installed in a computer system (PC). The initial and goal states describe the problem to be solved. Plan steps (shown as boxes) represent the actions that must be carried out to reach the goal state from the initial. For ease of understanding, partial state descriptions (italicized text) are also shown in the figure. Note that the plan shown is a partial-order plan, i.e., the RAM and NIC can be installed in any order once the PC is open. Figure 2.2(b) shows the four operators used by the planner to construct the plan. Each operator is defined in terms of preconditions and effects. Preconditions are the necessary conditions that must be true before the operator could be applied. Effects are the result of the operator application. Figure 2.2(c) shows the details of the `installNIC` operator. This operator can only be applied (i.e., the NIC can only be installed) when a NIC is available (*haveNIC*), the PC is open (∼*PCclosed*), and there is no NIC already installed (∼*installedNIC*). Once all these conditions are satisfied, the `installNIC` operator can be applied resulting in an installed NIC (*installedNIC*).

As mentioned above, most AI planners produce *partially-ordered* plans, in which only some steps are ordered with respect to one another. 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 2.3(a) shows another partial-order plan, this one for a GUI interaction. The nodes (labeled S_i, S_j, S_k, and S_l) represent the plan steps (instantiated operators) and the edges represent the causal links. The bindings are shown as parameters of the operators. Figure 2.3(b) lists the ordering constraints, all directly induced by the causal links in this example. In general, plans may include additional ordering constraints. The ordering constraints specify that the `DeleteText()` and `TypeInText()` actions can be performed in either order, but they must precede the `FILE_SAVEAS()` action and must be performed after the `FILE_OPEN()` action. Two legal orders shown in Figure 2.3(c) are obtained.

2.6.1 Action Representation

The output of the planner is a set of actions with certain constraints on the relationships among them. An action is an instance of an operator with its variables bound to values. One well-known action representation uses the STRIPS[1] language [22] that specifies operators in terms of parameterized preconditions and effects. STRIPS was developed more

[1] STRIPS is an acronym for STanford Research Institute Problem Solver.

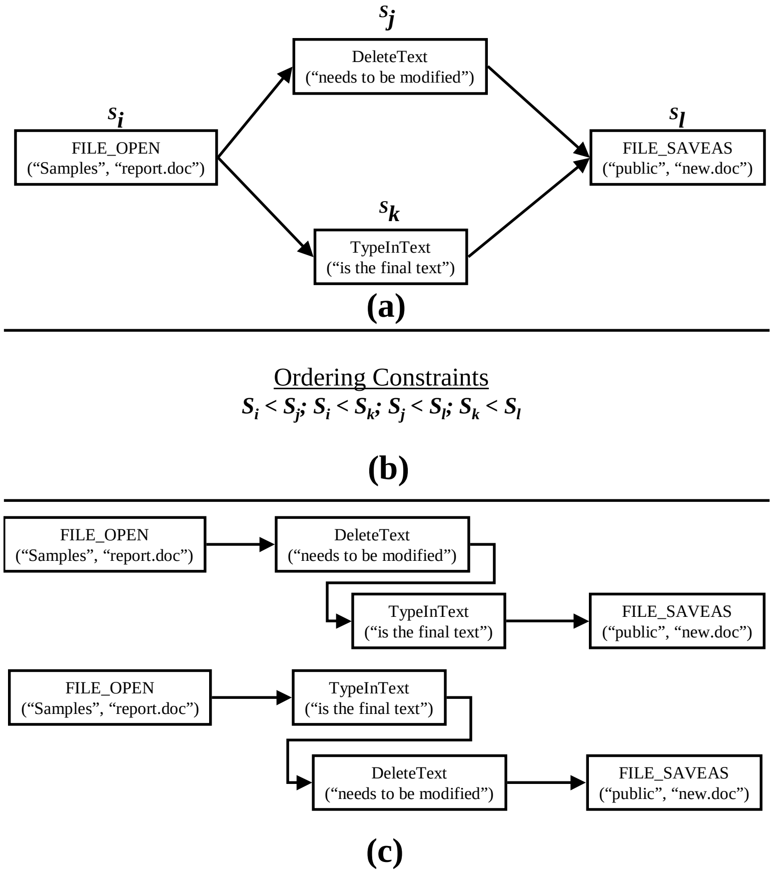


Figure 2.3: (a) A Partial-order Plan, (b) the Ordering Constraints in the Plan, and (c) the Two Linearizations.

than twenty years ago and has limited expressive power. For instance, no conditional or universally quantified effects are allowed. Although, in principle, sets of STRIPS operators could be defined to encode conditional effects, such encodings lead to an exponential number of operators, making even small planning problems intractable. A more powerful representation is ADL [62, 61], which allows conditional and universally quantified effects in the operators. This facility makes it possible to define operators in a more intuitive manner. A more recent representation is the Planning Domain Definition Language[2] (PDDL). The goals of designing the PDDL language were to encourage empirical evaluation of planner performance and the development of standard sets of planning problems. The language has roughly the expressiveness of ADL for propositions.

[2] Entire documentation available at `http://www.cs.yale.edu/pub/mcdermott/software/pddl.tar.gz`

2.6.2 Plan Generation as a Search Problem

The roots of AI planning lie in problem solving by using search. This search can either be through a space of domain states or plans. A *state space search* starts at the initial state, and applies operators one at a time until it reaches a state containing all the requirements of the goal. This approach - as is the case with all search problems - requires good heuristics to avoid exploring too much of the huge search space. State space planners typically produce totally-ordered plans. A *plan space planner* searches through a space of plans. It starts with a simple incomplete plan that contains a representation of only the initial and goal states. It then refines that plan iteratively until it obtains a complete plan that solves the problem. The intermediate plans are called "partial plans". Typical refinements include adding a step, imposing an ordering that puts one step before another, and instantiating a previously unbound variable. Plan space planners produce *partial-order plans*, introducing ordering constraints into plans only when necessary. A solution to the planning problem is any linearization of the complete plan that is consistent with the ordering constraints specified there. 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. Usually, the performance of plan space planners is better than that of state space planners because the branching factor is smaller (but *cf.* Veloso and Stone [84]). Again, however, heuristic search strategies have an important effect on efficiency.

A popular example of a plan space planner is UCPOP [63]. UCPOP and other earlier planning systems rely on graph search requiring unification of unbound variables. Unification considerably slows down the planning process. Consequently, these planners are useful for solving small problems and studying the behavior of different search strategies [66]. Results of experiments conducted by Memon et al. have in fact shown that these planners are much faster than their modern counterparts in finding short plans in domains containing a large number of objects [51].

2.6.3 Graphplan and IPP

Recently developed planning technology based on propositionalization of the search space has greatly increased the efficiency of plan generation. A well-known planner based on this technology is the Interference Progression Planner (IPP) [45], a system that extends the ideas of the Graphplan system [11] for plan generation. Graphplan introduced the idea of performing plan generation by converting the representation of a planning problem into a propositional encoding. Plans are then found by means of a search through a leveled graph, in which *even levels* $(0, 2, \ldots, i)$ represent all the (grounded) propositions that might be true at stage i of the plan, and *odd levels* $(1, 3, \ldots i+1)$ represent actions that might be performed at time $i+1$. The planners in the Graphplan family, including IPP, have shown increases in planning speeds of several orders of magnitude on a wide range of problems compared to earlier planning systems (but *cf.* [51]).

IPP uses ADL for the representation of actions in which preconditions and effects can be parameterized: subsequent processing does the conversion to propositional form. In fact, IPP generalizes Graphplan precisely by increasing the expressive power of its representation language, allowing for conditional and universally quantified effects. As is common in planning, IPP produces *partial order plans.*

2.6.4 Plan Generation as Propositional Satisfiability

Another promising planning system, namely SATPLAN, also based on propositionalization of the search space, uses *satisfiability* (SAT) [39] to find a plan. SATPLAN has now evolved into a complete planning system called BLACKBOX [41]. This system has been shown to be superior to IPP on several problems [40]. It also allows specification of domain knowledge to help speed up the planning process [42]. It makes use of very fast random SAT solvers [26]. One current limitation of this planning system is its restrictive input language, namely STRIPS, which as noted earlier does not allow quantification, making it unsuitable for efficient specification of complex actions.

2.6.5 Hierarchical Planning

The planners described in the previous section form plans at a single level of abstraction. Planning at one level of abstraction may be impractical for complex systems which consist of a large number of objects and operators. Techniques have been developed to generate plans at multiple levels of abstraction, typically called Hierarchical Task Network (HTN) planning [95, 20, 19]. In HTN planning, domain actions are modeled at different levels of abstraction, and each operator at level n specifies one or more “methods” at level $n-1$. A method is a single-level partial plan, and an action is said to “decompose” into its methods. HTN planning focuses on resolving conflicts among alternative methods of decomposition at each level.

2.7 Conclusions

This chapter presented an overview of the research that serves as the foundation for some of the concepts developed in this dissertation. In particular, the coverage evaluator extends the ideas of path-based coverage criteria to develop new event-based coverage criteria for GUIs; the test case generator develops a restricted form of hierarchical planning to generate GUI test cases; the test oracles extend the idea of using the expected output, to verify the correctness of a software, to using an expected state, automatically derived from the specifications, to verify the behavior of the GUI; the regression tester uses the idea of comparing graph-representations of the original and modified GUI to perform regression testing of the GUI. Details of the GUI representation that integrates all these GUI testing tools into one comprehensive framework are presented in the next chapter.

Chapter 3
GUI Representation

In the development of the integrated testing framework in this dissertation, a representation of the GUI that models its behavior is created from the GUI's specifications and/or from the structure of the GUI. All the other components of the framework employ the representation to perform a wide variety of tasks such as generating test cases and expected output, evaluating coverage and performing regression testing.

The GUI representation must satisfy a number of requirements. First, it should be at a conceptually high level of abstraction, free from platform-specific details, so that the generated testing information is portable across platforms. Second, it should be expressive enough so that a wide variety of GUIs can be represented. Third, it should be able to capture low-level details of GUIs so that a test oracle can be developed to determine whether an implemented GUI is executing correctly during testing. Fourth, it should be scalable so that large GUIs can be represented and tested efficiently. Finally, it should be intuitive, easy to develop and use.

The GUI representation developed in this dissertation models the GUI's state in terms of the specific objects that it contains and the values of their properties. Events that are performed on the GUI are modeled as state transducers and are represented as *operators*. These operators are defined in terms of the *preconditions* and *effects* of the events they represent. For efficiency and scalability, the GUI representation includes a hierarchy of *components*, each of which is used as a basic unit of testing. A new representation of a GUI component called an *event-flow graph* identifies events and their interactions. An *integration tree* represents the interactions among components. Subsequent sections in this chapter provide a formal definition of a GUI and details of the GUI representation, including algorithms to construct event-flow graphs and the integration tree.

3.1 What is a GUI?

A GUI is a graphical user interface to a program. Most of today's software interacts with a user through a graphical user interface. A GUI uses one or more metaphors for objects familiar in real life, such as buttons, menus, a desktop, the view through a window, trash-can, and the physical layout in a room. Objects of a GUI include elements such as windows, pull-down menus, buttons, scroll bars, iconic images, and wizards. The software user performs events to interact with the GUI, manipulating GUI objects as one would real objects. For example, dragging an item, discarding an object by dropping it in a trash-can, and selecting items from a menu are all familiar actions available in today's GUI. These events cause deterministic changes to the state of the software that may be reflected by a change in the appearance of one or more GUI objects.

GUIs, by their very nature, are hierarchical. This hierarchy is reflected in the grouping of events in windows, dialogs, and hierarchical menus. A typical GUI user focuses on events related by their functionality by opening a particular window or clicking on a pull-down menu. For example, all the "options" in MS Internet Explorer can be set by interacting with events in one window of the software's GUI.

The important characteristics of GUIs include their graphical orientation, event-driven input, hierarchical structure, the objects they contain, and the properties (attributes) of those objects. Formally, the class of GUIs of interest may be defined as follows:

Definition: A *Graphical User Interface (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 and hence the effect of an event cannot be predicted. This dissertation focuses on techniques to test the class of GUIs defined above. In Chapter 8, the framework is extended to test web user interfaces (WUIs).

In order to create a representation for the GUI, a model must be created for the GUI's state in terms of GUI objects, their properties, values, and the events that can be performed on the GUI. The GUI's hierarchical structure must also be modeled. The next section describes how to model the state of GUIs.

3.2 Representing the GUI's State

A GUI's state is modeled as a set of *objects*, (`label`, `form`, `button`, `text`, etc.) and a set of *properties* of those objects (`background-color`, `font`, `caption`, etc.). Each GUI will use certain types of objects with associated properties; at any specific point in time, the GUI can be described in terms of the specific objects that it contains and the values of their properties.

Formally, a GUI is modeled at a particular time t in terms of:

- its **objects** $O = \{o_1, o_2, \ldots, o_m\}$, and
- the **properties** $P = \{p_1, p_2, \ldots, p_l\}$ of those objects. Each property p_i is an n_i-ary Boolean relation, for $n_i \geq 1$, where the first argument is an object $o_1 \in O$. If $n_i > 1$, the last argument may either be an object or a property value, and all the intermediate arguments are objects. Figure 3.1(a) shows the structure of properties. The (optional) property value is a constant drawn from a set associated with the property in question: for instance, the property "`background-color`" has an associated set of values, {`white`, `yellow`, `pink`, etc.}. A distinguished set of properties, the *object types*, which are unary relations, ("`window`", "`button`") is assumed to be available. Figure 3.1(b) shows a `button` object called `Button1`. One of its properties is called `Caption` and its current value is "`Cancel`".

There are several points that should be noted about the description of properties. First, properties are relations, not functions, and so there may sometimes be multiple values for the same property of a given object. For example, there may be multiple objects in a window. Next, properties as defined are *fluents* [50], i.e., relations that are true in some situations (or states of the world) and not others. An everyday example of a fluent is the relation `president(US, Bush)`, with the obvious meaning, where the state it is evaluated in is the state of the real world. The fluents are evaluated with respect to a state of the GUI. Finally, a fluent may be undefined in some states, for example, `president(US, Dole)` in the state of the world in the year 1567, or `background-color(w24, blue)` in the state of a GUI immediately after window `w24` has been destroyed.

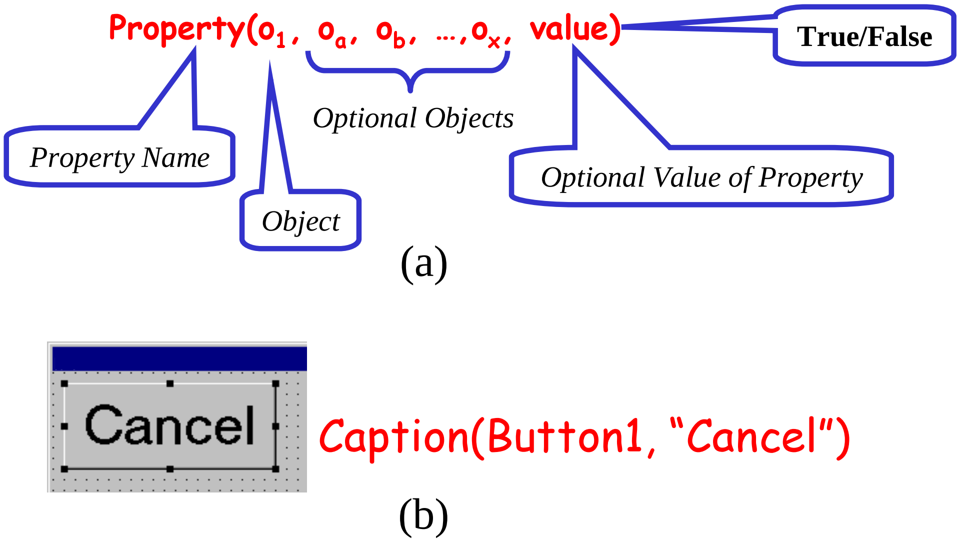

Property(o1, oa, ob, …, ox, value)
True/False
Optional Objects
Property Name
Object
Optional Value of Property
(a)
Cancel
Caption(Button1, "Cancel")
(b)

Figure 3.1: (a) The Structure of Properties, and (b) A `Button` Object with Associated Properties.

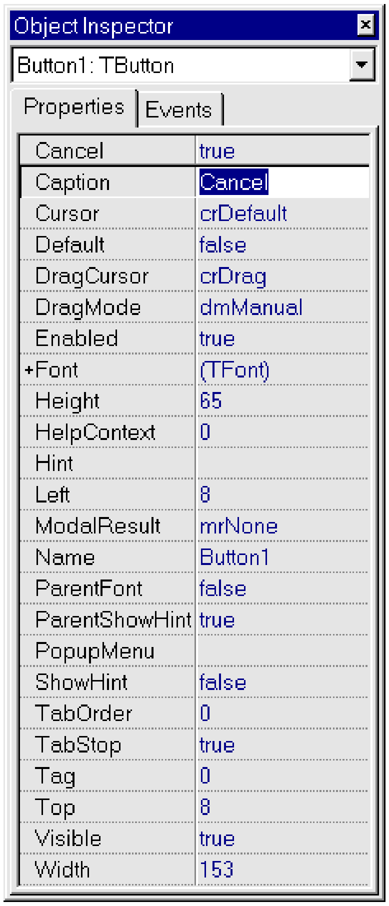
To create a model of the GUI, the objects in the GUI and their associated properties are identified. In practice, the set of object types and properties for a given GUI can be determined in several different ways.

1. *Manual examination of the GUI:* The GUI is manually examined, and all the object types and properties that can be discovered are noted. This approach is prone to incompleteness, especially since GUIs may have hidden properties that must be checked during verification. For example, the *tab order* of windows in a GUI (the order in which objects receive input focus when the `Tab` key is pressed) is a property that is not visible.

2. *Examination of the GUI's specifications:* The properties and object types are extracted from the GUI's specifications, which describe them either directly or implicitly within the descriptions of GUI events. This approach yields a more accurate set of properties and object types than does the first. However, additional properties may have been inadvertently introduced by the implementation platform, which, if not tested, may cause undesirable side-effects during GUI execution. [1]

3. *Examination of the language/toolkit used to develop GUI:* The language/toolkit is examined and all its object types and properties identified. For example, if the GUI was

[1] Note that testing platform-specific properties is done at the cost of reduced portability. For a fully portable representation, the properties should be derived only from the specifications.

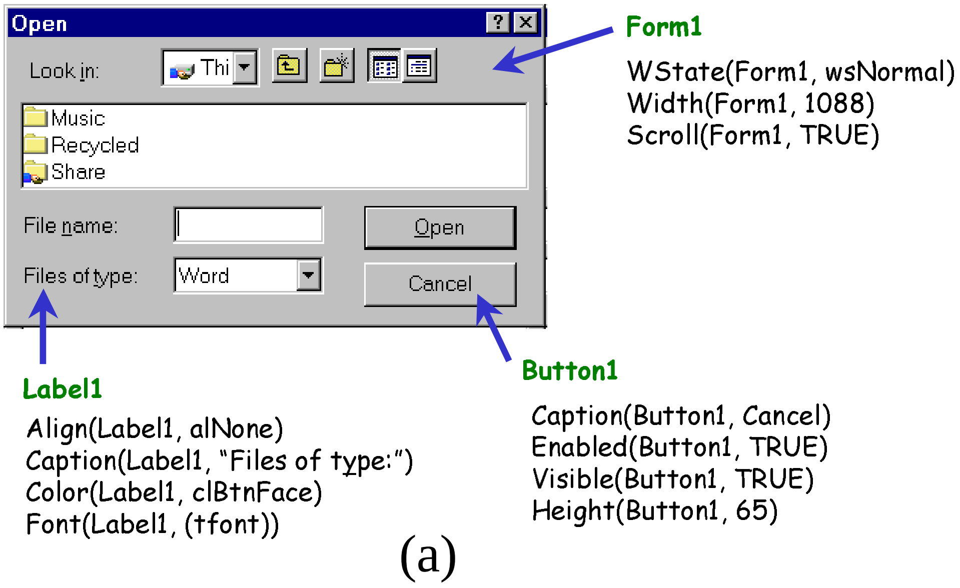


Cancel	true
Caption	Cancel
Cursor	crDefault
Default	false
DragCursor	crDrag
DragMode	dmManual
Enabled	true
+Font	(TFont)
Height	65
HelpContext	0
Hint	
Left	8
ModalResult	mrNone
Name	Button1
ParentFont	false
ParentShowHint	true
PopupMenu	
ShowHint	false
TabOrder	0
TabStop	true
Tag	0
Top	8
Visible	true
Width	153

Figure 3.2: The List of all Properties of the Button Object in Borland's C++ Builder.

developed using the Java language [28, 2], then the GUI objects would be instances of the **swing** GUI components of the Java swing package, and the properties would correspond to the instance variables (also called "data members" in C++) of each object. Visual programming environments provide a more direct interface to properties. Borland's C++ Builder presents the properties as a table for the currently selected object. An example of all the properties that Borland's C++ Builder associates with the `Button` object is seen in Figure 3.2.

The third approach above can lead to a larger set of object types and properties than does the second because the set of object types and properties made available by a language or toolkit may not all be used in the construction of a particular GUI. For example, one might use Borland's C++ builder to construct a simple GUI in which the user is not permitted to manipulate the text color, and in which the text color does not influence the execution of any other event. If a text editor similar to Microsoft's NotePad



State = **{**Align(Label1, alNone)**,** Caption(Label1, "Files of type:")**,** Color(Label1, clBtnFace), Font(Label1, (tfont)), WState(Form1, wsNormal), Width(Form1, 1088), Scroll(Form1, TRUE), Caption(Button1, Cancel), Enabled(Button1, TRUE), Visible(Button1, TRUE), Height(Button1, 65), …**}**

(b)

Figure 3.3: (a) The Open GUI with three objects explicitly labeled and their associated properties, and (b) the State of the Open GUI.

is implemented in Borland's C++ builder, then if one establishes the set of properties from the GUI's specifications, text color will not be among the properties modeled, whereas if one establishes it from the toolkit used for development, text color will be included as a property in the model. Hence, there are two sets of properties that can be obtained: the *complete set* of properties for a GUI, which are all those that would be identified by the third (language/toolkit-based) approach, and the *reduced set*, which includes only those that would be identified by the second (specifications-based) approach. Note that the reduced set is always a (possibly improper) subset of the complete set of properties.

The set of objects and their properties can be obtained using any one of the techniques described above and used to create a model of the *state* of the GUI.

Definition: The *state* of a GUI at a particular time t is the set P of all the properties of all the objects O that the GUI contains. □

A description of the state would contain information about the types of *all* the objects currently extant in the GUI, as well as *all* of the properties of each of those objects.

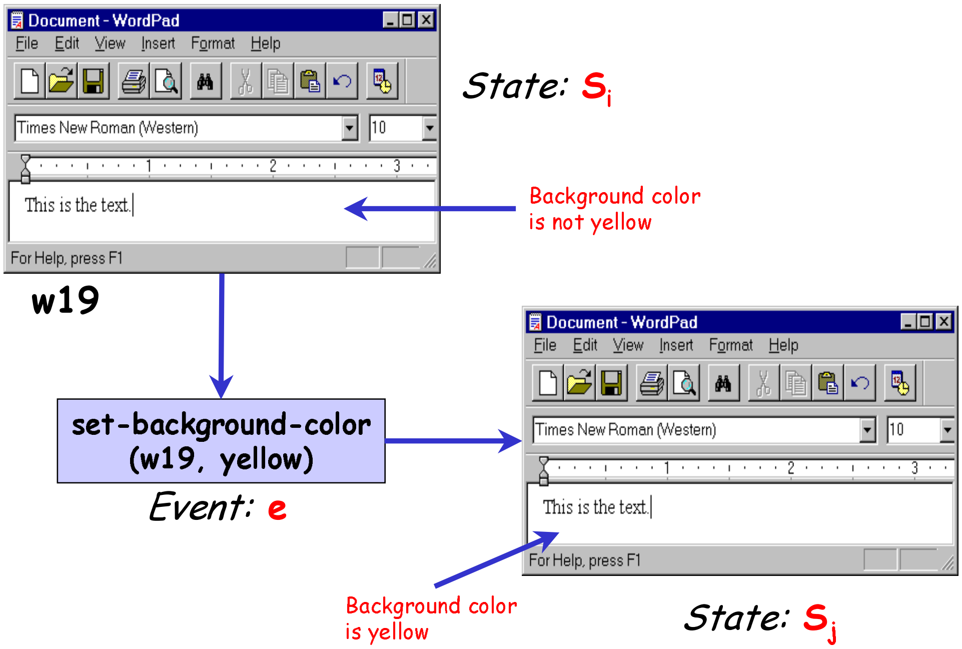


Figure 3.4: An Event Changes the State of the GUI.

For example, consider the `Open` GUI shown in Figure 3.3(a). This GUI contains several objects, three of which are explicitly labeled; for each, a small subset of its properties is shown. The state of the GUI, partially shown in Figure 3.3(b), contains all the properties of all the objects in `Open`.

3.3 Representing GUI Events

The state of a GUI is not static; events preformed on the GUI change its state. Events are modeled as state transducers.

Definition: 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. □

Since events may be performed on different types of objects, in different contexts, yielding different behavior, they are parameterized with objects and property values. For example, an event `set-background-color( w, x )` may be defined in terms of a `window w` and `color x`; `w` and `x` may take specific values in the context of a particular GUI execution. As shown in Figure 3.4, whenever the event `set-background-color( w19, yellow )` is executed in a state in which window `w19` is open, the background color of `w19` should become `yellow` (or stay `yellow` if it already was), and no other properties of the GUI should change. This example illustrates that, typically, events can only be executed in some states; `set-background-color( w19, yellow )` cannot be executed when window `w19` is not open.

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.[2] Hence, GUI events are represented using *operators*, which specify their preconditions and effects:

Definition: An **operator** is a 3-tuple <`Name`, `Preconditions`, `Effects`> where:

- `Name` identifies an event and its parameters.
- `Preconditions` is a set of positive ground literals[3] $p(arg_1, \ldots, arg_n)$, where p is an n-ary property (i.e., $p \in P$). $Pre(Op)$ represents the set of preconditions for operator Op. An operator is applicable in any state S_i in which all the literals in $Pre(Op)$ are true.
- `Effects` is also a set of positive or negative ground literals $p(arg_1, \ldots, arg_n)$, where p is an n-ary property (i.e., $p \in P$). $Eff(Op)$ represents the set of effects for operator Op. In the resulting state S_j, all of the positive literals in $Eff(Op)$ will be true, as will all the literals that were true in S_i except for those that appear as negative literals in $Eff(Op)$. □

For example, the following operator represents the `set-background-color` event discussed earlier:

Name: `set-background-color(wX: window, Col:  Color)`
Preconditions: `is-current(wX), background-color(wX, oldCol), oldCol` $\neq$ `Col`
Effects: `background-color(wX, Col)`

Going back to the example of the GUI in Figure 3.4 in which the following properties are true before the event is performed: `window(w19), background-color(w19, blue), is-current(w19)`. Application of the above operator, with variables bound as `set-background-color( w19, yellow )`, would lead to the following state: `window(w19), background-color(w19, yellow), is-current(w19)`, i.e., the background color of window `w19` would change from `blue` to `yellow`.

[2] 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.

[3] A literal is a sentence without conjunction, disjunction or implication; a literal is ground when all of its arguments are bound; and a positive literal is one that is not negated. It is straightforward to generalize the account given here to handle partially instantiated literals. However, it needlessly complicates the presentation.

The above scheme for encoding operators is the same as what is standardly used in the AI planning literature [62, 86, 87]; the persistence assumption built into the method for computing the result state is called the "STRIPS assumption". A complete formal semantics for operators making the STRIPS assumption has been developed by Lifschitz [48].

One final point to note about the representation of effects is the inability to efficiently express complex events when restricted to using only sets of literals. Although in principle, multiple operators could be used to represent almost any event, complex events may require the definition of an exponential number of operators, making planning inefficient. In practice, a more powerful representation that allows conditional and universally quantified effects is employed. For example, the operator for the `Paste` event would have different effects depending on whether the clipboard was empty or full. Instead of defining two operators for these two scenarios, a conditional effect could be used instead. In cases where even conditional and quantified effects are inefficient, *procedural attachments*, i.e., arbitrary pieces of code that perform the computation, are embedded in the effects of the operator [37]. One common example is the representation of computations. A calculator GUI that takes as input two numbers, performs computations (such as addition, subtraction) on the numbers, and displays the results in a text field will need to be represented using different operators, one for each distinct pair of numbers. By using a procedural attachment, the entire computation may be handled by a piece of code, embedded in a single operator.

3.4 Representing Executable Event Sequences

In this section, the representation of an event in terms of its preconditions and effects is used to develop a formal representation of an executable event sequence. The function notation $S_j = e(S_i)$ is used to denote that S_j is the state resulting from the execution of event e in state S_i. Events can be strung together into sequences.

Definition: $e_1 \circ e_2 \circ \ldots \circ e_n$ is an *executable event sequence for a state* S_0 iff there exists a sequence of states $S_0; S_1; \ldots; S_n$ such that $S_i = e_i(S_{i-1})$, for $i = 1, \ldots, n$. □

Figure 3.5 shows MS WordPad in a state S_0 and an executable event sequence corresponding to S_0. Extending the function notation above, $S_j = (e_1 \circ e_2 \circ \ldots \circ e_n)(S_i)$, where $e_1 \circ e_2 \circ \ldots \circ e_n$ is an executable event sequence, denotes that S_j is the state that results from executing the specified sequence of events starting in state S_i.

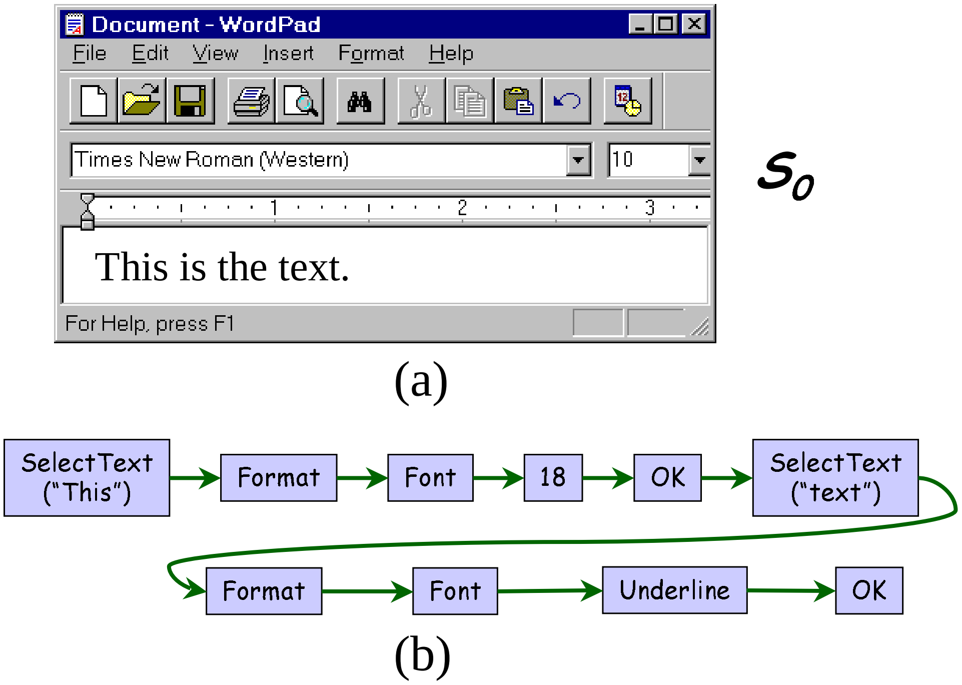


Figure 3.5: (a) A State S_0 for MS WordPad, and (b) an Executable Event Sequence for S_0.

As mentioned earlier in Section 1.2, the controllability problem in GUIs requires that the GUI be brought into a valid state before performing events on it. With each GUI is associated a distinguished set of states called its *valid initial states*.

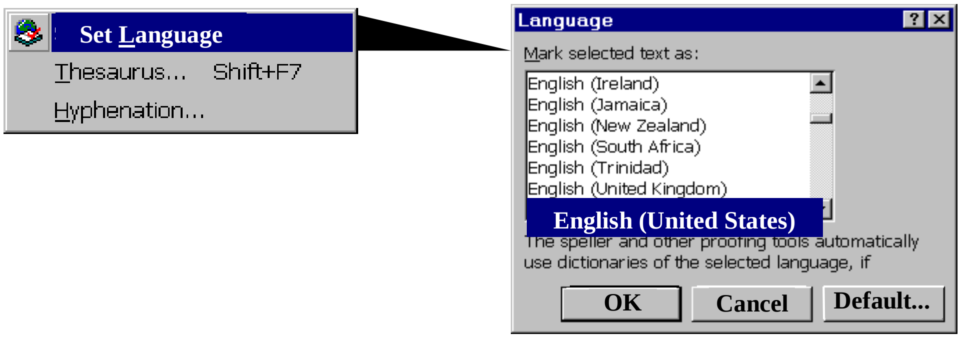
Definition: A set of states S_I is called the *valid initial state* set for a particular GUI iff the GUI may be in any state $S_i \in S_I$ when it is first invoked. □

Given a GUI in state $S_i \in S_I$, i.e., in a valid initial state of the GUI, new states may be obtained by performing events on S_i. These states are called the *reachable* states of the GUI. Formally, a reachable state is defined as follows.

Definition: The state S_j is a *reachable state* iff either $S_j \in S_I$ or there exists an executable event sequence $e_x \circ e_y \circ \ldots \circ e_z$ such that $S_j = (e_x \circ e_y \circ \ldots \circ e_z)(S_i)$, for any $S_i \in S_I$. □

3.5 GUI Components and Event Classification

Since today's GUIs are large and contain a large number of events, any scalable representation must decompose a GUI into manageable parts. As mentioned previously, GUIs are hierarchical, and this hierarchy may be exploited to identify groups of GUI events



Mark selected text as:

The speller and other proofing tools automatically use dictionaries of the selected language, if

Figure 3.6: The Event `Set Language` Opens a Modal Window.

that can be analyzed in isolation. One hierarchy of the GUI and the one used in this research is obtained by examining the structure of *modal windows* in the GUI.

Definition: 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. □

The language selection window is an example of a modal window in MS Word. As Figure 3.6 shows, when the user performs the event `Set Language`, a window entitled `Language` opens and the user spends time selecting the language, and finally explicitly terminates the interaction by either performing `OK` or `Cancel`.

Other windows in the GUI are called *modeless windows* that do not restrict the user's focus; they merely expand the set of GUI events available to the user. For example, in the MS Word software, performing the event `Replace` opens a modeless window entitled `Replace` (Figure 3.7).

At all times during interaction with the GUI, the user interacts with events within a modal dialog. This modal dialog consists of a modal window X and a set of modeless windows that have been invoked, either directly or indirectly by X. The modal dialog remains in place until X is explicitly terminated. Intuitively, the events within the modal dialog form a *GUI component*.

Definition: A *GUI component* C is an ordered pair $(\mathcal{RF}, \mathcal{UF})$, where $\mathcal{RF}$ represents a modal window in terms of its events and $\mathcal{UF}$ is a set whose elements represent modeless windows also in terms of their events. Each element of $\mathcal{UF}$ is invoked either by an event in $\mathcal{UF}$ or $\mathcal{RF}$. □

Note that, by definition, events within a component do not interleave with events in other components without the components being explicitly invoked or terminated.

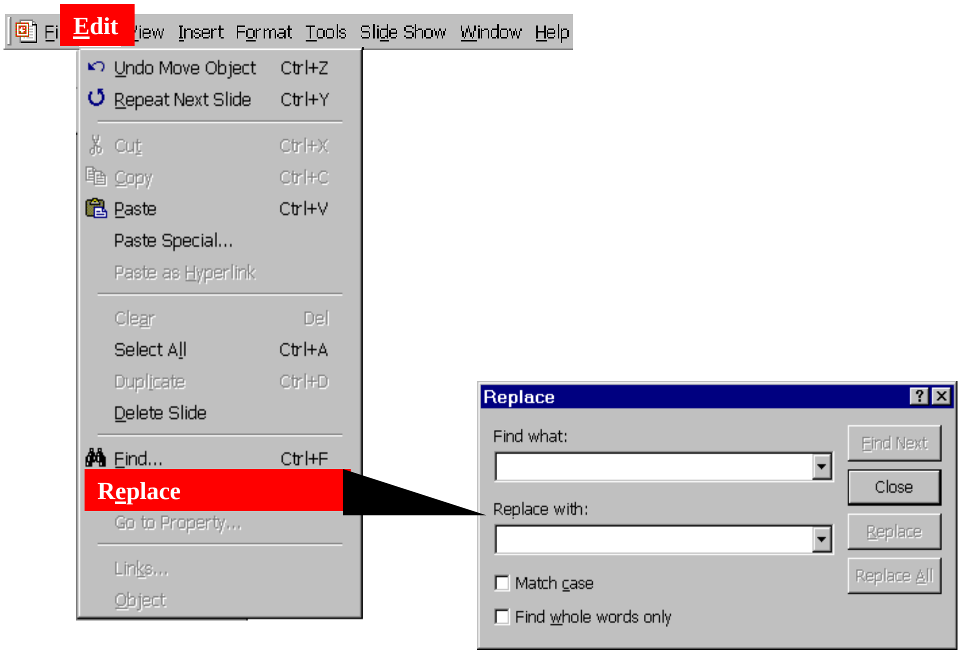


Figure 3.7: The Event `Replace` Opens a Modeless Window.

Since components are defined in terms of modal windows, a classification of GUI events is used to identify components. The classification of GUI events is as follows:

Restricted-focus events open *modal windows*. `Set Language` in Figure 3.6 is a restricted-focus event.

Unrestricted-focus events open *modeless windows*. For example, `Replace` in Figure 3.7 is an unrestricted-focus event.

Termination events close modal windows; common examples include `Ok` and `Cancel` (Figure 3.6).

The GUI contains other types of events that do not open or close windows but make other GUI events available. These events are used to open menus that contain several events.

Menu-open events are used to open menus. They expand the set of GUI events available to the user. Menu-open events do not interact with the underlying software. Note that the only difference between menu-open events and unrestricted-focus events is that the latter open windows that must be explicitly terminated. The most common example of menu-open events are generated by buttons that open pull-down menus. For example, in Figure 3.8, `File` and `SentTo` are menu-open events.

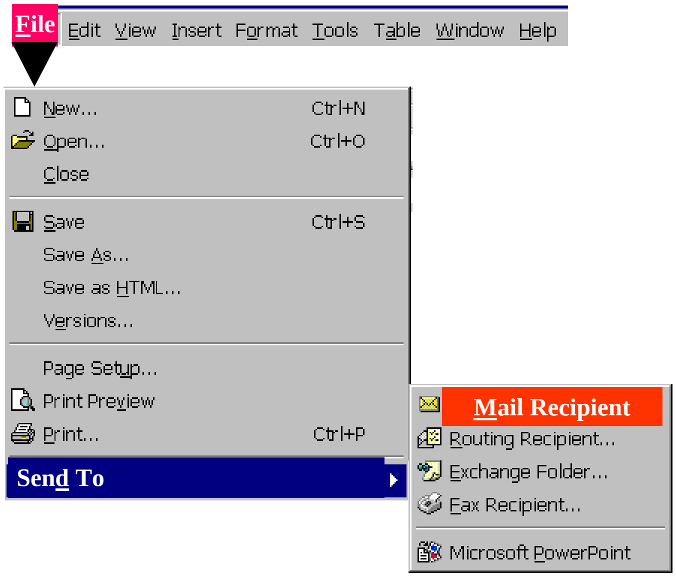


Figure 3.8: Menu-open Events: `File` and `Send To`.

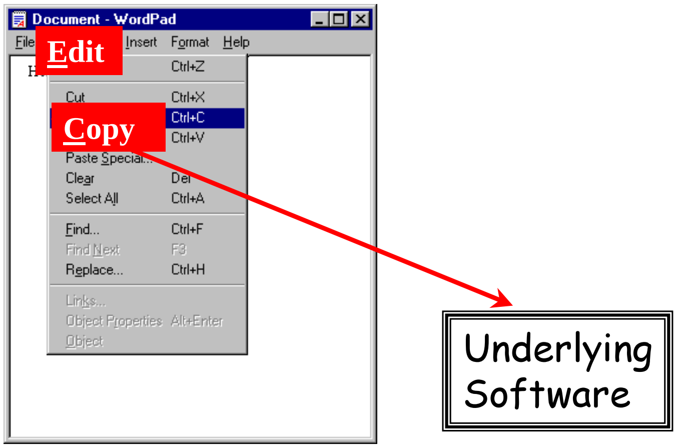
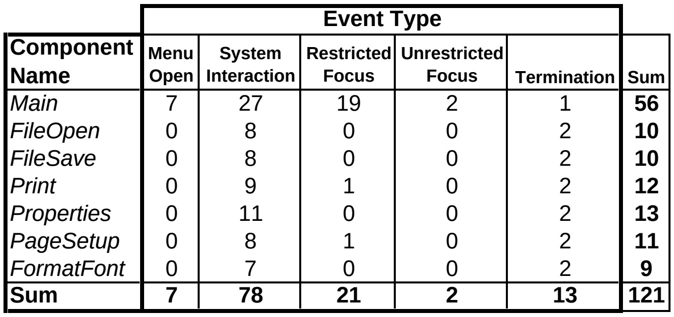


Figure 3.9: A System-interaction Event: `Copy`.

Finally, the remaining events in the GUI are used to interact with the underlying software.

System-interaction events interact with the underlying software to perform some action; common examples include the `Copy` event used for copying objects to the clipboard (see Figure 3.9).

Table 3.1 lists some of the components of WordPad. Each row represents a component and each column shows the different types of events available within each component.



	Event Type					
Component Name	**Menu Open**	**System Interaction**	**Restricted Focus**	**Unrestricted Focus**	**Termination**	**Sum**
Main	7	27	19	2	1	**56**
FileOpen	0	8	0	0	2	**10**
FileSave	0	8	0	0	2	**10**
Print	0	9	1	0	2	**12**
Properties	0	11	0	0	2	**13**
PageSetup	0	8	1	0	2	**11**
FormatFont	0	7	0	0	2	**9**
Sum	**7**	**78**	**21**	**2**	**13**	**121**

Table 3.1: Types of Events in Some Components of MS WordPad.

`Main` is the component that is available when WordPad is invoked. Other components' names indicate their functionality. For example, `FileOpen` is the component of WordPad used to open files.

3.6 Event-flow Graphs

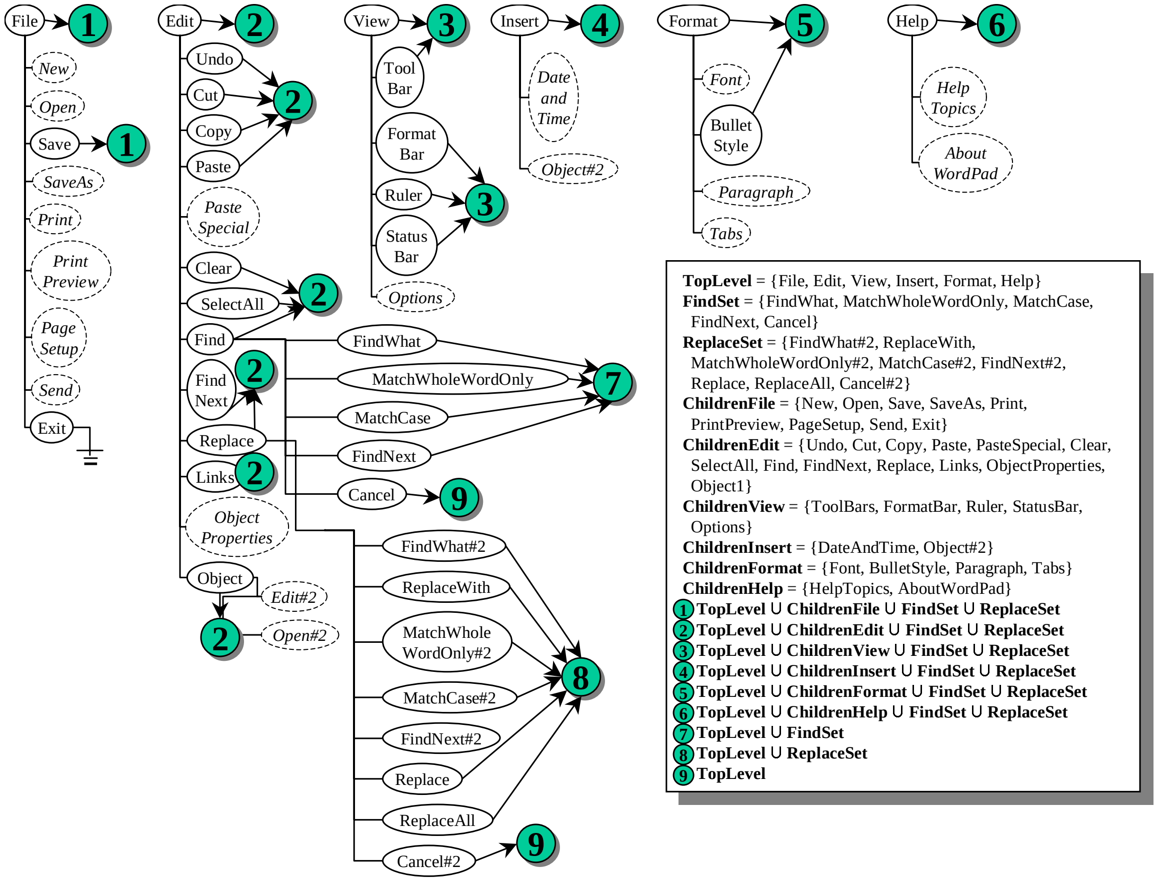
A GUI component may be represented as a flow graph. Intuitively, an *event-flow graph* represents all possible interactions among the events in a component.

Definition: An *event-flow graph* for a component C is a 4-tuple <**V**, **E**, **B**, **I**> where:

1. **V** is a set of vertices representing all the events in the component. Each $v \in \mathbf{V}$ represents an event in C.
2. $\mathbf{E} \subseteq \mathbf{V} \times \mathbf{V}$ is a set of directed edges between vertices. Event e_i **follows** e_j iff e_j may be performed immediately after e_i. An edge $(v_x, v_y) \in \mathbf{E}$ iff the event represented by v_y **follows** the event represented by v_x.
3. $\mathbf{B} \subseteq \mathbf{V}$ is a set of vertices representing those events of C that are available to the user when the component is first invoked.
4. $\mathbf{I} \subseteq \mathbf{V}$ is the set of restricted-focus events of the component.

□

An example of an event-flow graph for the `Main` component of MS WordPad is shown in Figure 3.10. To increase readability of the event-flow graph, all of the edges have not been shown. Instead, labeled circles have been used as connectors to sets of events. The legend shows the set of events represented by each circle. For example, an edge from `Save` to ① represent an edge from the event `Save` to each element of the set represented by ①. At the top of the figure are the vertices, `File`, `Edit`, `View`, `Insert`, `Format`, and `Help`,



TopLevel = {File, Edit, View, Insert, Format, Help}
FindSet = {FindWhat, MatchWholeWordOnly, MatchCase, FindNext, Cancel}
ReplaceSet = {FindWhat#2, ReplaceWith, MatchWholeWordOnly#2, MatchCase#2, FindNext#2, Replace, ReplaceAll, Cancel#2}
ChildrenFile = {New, Open, Save, SaveAs, Print, PrintPreview, PageSetup, Send, Exit}
ChildrenEdit = {Undo, Cut, Copy, Paste, PasteSpecial, Clear, SelectAll, Find, FindNext, Replace, Links, ObjectProperties, Object1}
ChildrenView = {ToolBars, FormatBar, Ruler, StatusBar, Options}
ChildrenInsert = {DateAndTime, Object#2}
ChildrenFormat = {Font, BulletStyle, Paragraph, Tabs}
ChildrenHelp = {HelpTopics, AboutWordPad}
① **TopLevel** ∪ **ChildrenFile** ∪ **FindSet** ∪ **ReplaceSet**
② **TopLevel** ∪ **ChildrenEdit** ∪ **FindSet** ∪ **ReplaceSet**
③ **TopLevel** ∪ **ChildrenView** ∪ **FindSet** ∪ **ReplaceSet**
④ **TopLevel** ∪ **ChildrenInsert** ∪ **FindSet** ∪ **ReplaceSet**
⑤ **TopLevel** ∪ **ChildrenFormat** ∪ **FindSet** ∪ **ReplaceSet**
⑥ **TopLevel** ∪ **ChildrenHelp** ∪ **FindSet** ∪ **ReplaceSet**
⑦ **TopLevel** ∪ **FindSet**
⑧ **TopLevel** ∪ **ReplaceSet**
⑨ **TopLevel**

Figure 3.10: Event-flow Graph for the `Main` Component of MS WordPad.

that represent the pull-down menu of MS WordPad. They are menu-open events that are available when the `Main` component is first invoked; they form the set **B**. Once `File` has been performed in WordPad any of the events in ① may be performed; there are edges in the event-flow graph from `File` to each of these events. Note that `Open` is shown as a dashed oval. This notation is used for restricted-focus events. Similarly, `About` and `Contents` are also restricted-focus events. Hence, for this component $\mathbf{I} = \{$*all events shown with dashed ovals*$\}$. Other events such as `Save`, `Cut`, `Copy`, and `Paste` are all system-interaction events.

The next section presents an algorithm to construct an event-flow graph for a given GUI.

3.6.1 Construction of Event-flow Graphs

The construction of event-flow graphs is based on the structure of the GUI. The classification of events in the previous section is used by an algorithm that constructs event-

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

Figure 3.11: Computing `follows`(**v**) for a Vertex **v**.

flow graphs for a GUI. Intuitively, 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`(**v**) can be determined using the algorithm in Figure 3.11 for each vertex **v**. The recursive algorithm contains a switch structure that assigns `follows`(**v**) according to the type of each event. If the type of the event **v** is a menu-open event (`line 2`) and $\mathbf{v} \in \mathbf{B}$ (recall that **B** represents events that are available when a component is invoked) then the user may either perform **v** again, its sub-menu choices, or any event in **B** (`line 4`). However, if $\mathbf{v} \notin \mathbf{B}$ then the user may either perform all sub-menu choices of **v**, **v** itself, or all events in `follows`(`parent`(**v**)) (`line 6`); `parent`(**v**) is defined as any event that makes **v** available. If **v** is a system-interaction event, then after performing **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 component, then `follows`(**v**) consists of all the top-level events of the invoking component (`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 component available as well as all events of the invoking component (`line 12`). Lastly, if **v** is a restricted-focus event, then only the events of the invoked component are available.

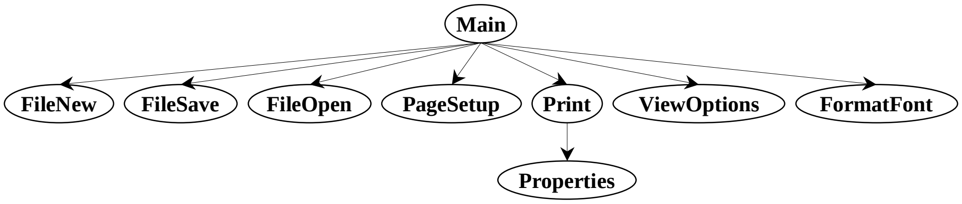


Figure 3.12: An Integration Tree for a Part of MS WordPad.

3.7 Integration Tree

Once all the components of the GUI have been represented as event-flow graphs, the remaining step is to identify interactions among components. A structure called an *integration tree* is constructed to identify interactions (invocations) among components.

Definition: Component C_x **invokes** component C_y if C_x contains a restricted-focus event e_x that invokes C_y. □

Intuitively, the integration tree shows the invokes relationship among all the components in a GUI. Formally, an integration tree is defined as:

Definition: An *integration tree* is a 3-tuple $< \mathcal{N}, \mathcal{R}, \mathcal{B} >$, where $\mathcal{N}$ is the set of components in the GUI and $\mathcal{R} \in \mathcal{N}$ is a designated component called the `Main` component. $\mathcal{B}$ is the set of directed edges showing the invokes relation between components, i.e., $(C_x, C_y) \in \mathcal{B}$ iff C_x **invokes** C_y. □

Figure 3.12 shows an example of an integration tree representing a part of the MS WordPad's GUI. The nodes represent the components of the MS WordPad GUI and the edges represent the invokes relationship between the components. The tree in Figure 3.12 has an edge from `Main` to `FileOpen` showing that `Main` contains an event, namely `Open` (see Figure 3.10) that invokes `FileOpen`.

It is relatively straightforward to obtain the integration tree from the computation of `follows`. Modifying `Lines 13..14` of the algorithm shown in Figure 3.11, one can keep track of the components invoked. Once all the components in the GUI have been identified, the integration tree may be constructed by adding, for each restricted-focus event e_x, the element (C_x, C_y) to $\mathcal{B}$ where C_x is the component that contains e_x and C_y is the component that it invokes.

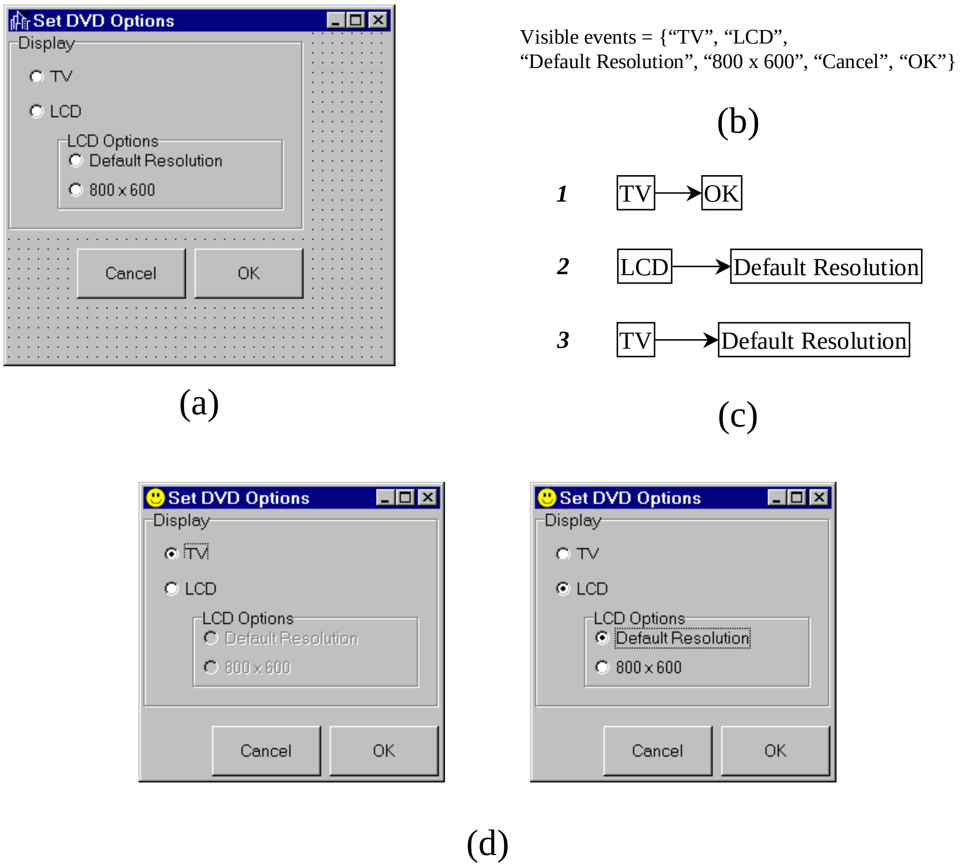


Figure 3.13: (a) A Snap-shot of the GUI at Implementation Time, (b) the Set of Visible Events, (c) a Few Legal Event-sequences, and (d) the GUI at Run-time.

3.8 Representing GUI Test Cases

The GUI representation presented in this chapter is used in this dissertation for GUI testing. To test a GUI, event sequences for the GUI must be executed.

Definition: A *legal event sequence* of a GUI is $e_1; e_2; e_3; ...; e_n$ where either $(e_i, e_{i+1}) \in$ **E**, for some component of the GUI, or e_i is a restricted-focus event that invokes component C_x and e_{i+1} is an event in C_x, for $1 \le i \le n-1$. □

Note that a legal event sequence is less restricted than an executable event sequence. Hence the set of all legal event sequences also contains all executable event sequences. Consider the example of a GUI used to select the output of a DVD player shown in Figure 3.13. A snap-shot of the GUI during implementation is shown in Figure 3.13(a). The GUI contains six events, namely “`TV`”, “`LCD`”, “`Default Resolution`”, “`800` × `600`”, “`Cancel`”, and “`OK`” (Figure 3.13(b)). Note that all these events are visible to the GUI user. Since the event-flow graph is developed using the visibility information, the three event sequences shown in Figure 3.13(c) are legal. However, note that during execution, if the event "`TV`" is performed, the two events "`Default Resolution`" and "`800` × `600`" are greyed-out, i.e., their `Enabled` properties are `False`. Hence, event sequence 3, although legal, is not executable. During testing, it is important to test legal sequences, even though they may not all be executable.

A formal representation of a GUI test case is as follows:

Definition: A **GUI test case** T is a triple $< S_0, e_1; e_2; \ldots; e_n, S_1; S_2; \ldots; S_n >$, consisting of a *reachable state* S_0, called the *initial state for* T, a legal event sequence $e_1; e_2; \ldots; e_n$ for S_0, and expected states $S_1; S_2; \ldots; S_n$, where $S_i = e_i(S_{i-1})$ for $i = 1, \ldots, n$. □

For compactness, a test case may be represented by the pair $< S_0, e_1; e_2; \ldots; e_n >$, since the expected-state sequence may be obtained from S_0 whenever needed.

3.9 Conclusions

This chapter presented the GUI representation that is the central component of the GUI testing framework developed in this dissertation. The representation models the state of the GUI in terms of the objects the GUI contains and their properties. Events and their interactions are captured at a conceptually high level of abstraction. Scalability is achieved by decomposing the GUI into manageable components, each of which can be used as a unit of testing. The developed representation is used by all the other components of the GUI testing framework. The next chapter shows how the representation is used to develop coverage criteria for GUIs.

Chapter 4

Coverage Evaluator

The coverage evaluator is an important component of the GUI testing framework and plays two roles during GUI testing. First, it employs *coverage criteria* to specify what to test in a GUI by analyzing the representation of the GUI. Second, given a generated test suite, the coverage evaluator employs coverage criteria to determine whether the test suite has adequately tested the implemented GUI. Although at first glance it may seem that one of these roles of the coverage evaluator is redundant, both roles are equally important because (1) the GUI representation derived from its specifications may not accurately represent the implementation, (2) *infeasibility* may prevent certain parts of the GUI from being tested, and (3) testing may depend on certain resources, such as time, and it may be terminated when these resources are exhausted. Consequently, it may not always be possible to test in a GUI implementation what is recommended by the coverage criteria. Also, note that in certain testing problems, the coverage evaluator may not necessarily be automated for both tasks. For example, specifying what to test in a GUI may be done manually whereas evaluating the coverage of the test suite may be done automatically.

The central mechanism of the coverage evaluator for testing software is a set of coverage criteria, which are rules used to help determine what to test in a software and whether a test suite has adequately tested a program. Common examples of coverage criteria for conventional software are structural, and include statement coverage, branch coverage, and path coverage, which require that every statement, branch and path in the program's code be executed by the test suite respectively. Existing coverage criteria developed for traditional software do not address the adequacy of GUI test cases. GUIs are typically developed using instances of precompiled elements stored in a library. The source code of these elements may not always be available to be used for coverage evaluation based on code. Moreover, the event sequences that the GUI must be tested for are conceptually at a much higher level of abstraction than the code and hence cannot be obtained from the code. For the same reason, the code cannot be used to determine whether an adequate number of these sequences have been tested on the GUI.

The above challenges suggest the need to develop coverage criteria based on events in a GUI. The development of such coverage criteria has certain requirements. First, since the GUI consists of components, coverage criteria must be developed for events within a component. Second, coverage criteria must be developed for interactions among components. Third, it should be possible to satisfy a coverage criterion by a finite-sized test suite. The *finite applicability* [96] requirement holds if a coverage criterion can always be satisfied by a finite-sized test suite. Finally, the test designer should recognize whether a coverage criterion can be fully satisfied [88, 89]. For example, it may not always be possible to satisfy path coverage because of the presence of *infeasible paths*, which are not executable because of the context of some instructions. No test case can execute along an infeasible path, perhaps resulting in loss of coverage. Detecting infeasible paths in general is a NP complete problem. Infeasibility can also occur in GUIs. Similar to infeasible paths in code, static analysis of the GUI may not reveal infeasible sequences of events. For example, by performing static analysis of the menu structure of MS Wordpad, one may construct a test case with `Paste` as the first event. However, experience of using the software shows that such a test case will not execute since `Paste` is highlighted only after a `Cut` or `Copy`.[1]

In this chapter, a new class of coverage criteria called *event-based coverage criteria* is defined. The key idea is to define the coverage of a test suite in terms of GUI events and their interactions. Since the GUI is composed of components, two kinds of coverage criteria are developed – *intra-component coverage criteria* for events within a component and *inter-component coverage criteria* for events among components. Intra-component criteria include *event coverage*, *event-interaction coverage*, and *length-n event-sequence coverage*. The *length-n event-sequence coverage* is also used for inter-component testing in addition to *invocation coverage* and *invocation-termination coverage*. Algorithms are provided to evaluate intra- and inter-component coverage of a given test suite. Experiments demonstrate the usefulness of the coverage criteria and a correlation between event-based coverage of the WordPad's GUI and the statement coverage of its underlying code.

The next section presents coverage criteria for event interactions within a component. Section 4.2 presents coverage criteria for events among components. Section 4.3 presents algorithms to evaluate intra- and inter-component coverage of the GUI for a given test suite. In Section 4.4, the results of experiments conducted on a version of the WordPad software are presented.

[1] Note that `Paste` will be available if the ClipBoard is not empty, perhaps because of an external software. External software is ignored in this simplified example.

4.1 Intra-component Coverage

In this section, several coverage criteria for events and their interactions within a component are defined. Recall from Section 3.6 that each GUI component is represented as an event-flow graph in which $\mathbf{V}$ is a set of vertices representing all the events in the component and $\mathbf{E} \subseteq \mathbf{V} \times \mathbf{V}$ is a set of directed edges between vertices. The intra-component coverage criteria are based on legal event sequences. In the remainder of this chapter, the term event sequence will be used to mean a legal event sequence.

4.1.1 Event Coverage

Intuitively, event coverage requires each event in the component to be performed at least once. Such a requirement is necessary to check whether each event executes as expected.

Definition: A set P of event-sequences satisfies the *event coverage criterion* if and only if for all events $v \in \mathbf{V}$, there is at least one event-sequence $p \in P$ such that event v is in p. □

For example, in the event-flow graph of Figure 3.10, event-coverage would require that all the events in the event-flow graph be executed by a test case at least once. Since there are 56 events in the event-flow graph, 56 test cases of length 1 would suffice.

4.1.2 Event-interaction Coverage

Another important aspect of GUI testing is to check the interactions among all possible pairs of events in the component. However, these checks should be restricted to pairs of events that may be performed in a sequence.

Definition: The *event-interactions* for an event e is the set $\{e_j \mid (e, e_j) \in \mathbf{E}\}$. □

This criterion requires that after an event e has been performed, all the events that can interact with e should be executed at least once. Note that this requirement is equivalent to requiring that each element in $\mathbf{E}$ be covered by at least one test case.

Definition: A set P of event-sequences satisfies the *event-interaction coverage criterion* if and only if for all elements $(e_x, e_y) \in \mathbf{E}$, there is at least one event-sequence $p \in P$ such that p contains (e_x, e_y). □

For example, the event-flow graph of Figure 3.10 contains 791 edges. All length 2 test cases that cover these 791 edges would satisfy event-interaction coverage.

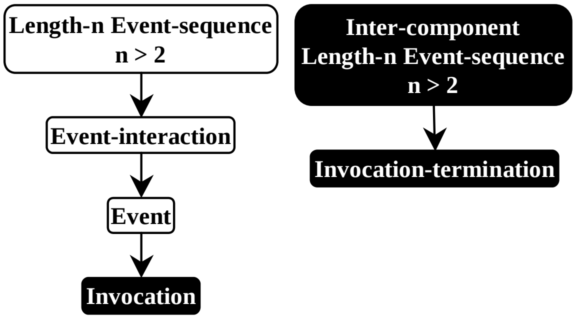


Figure 4.1: The **Subsume** Relation between Event-based Coverage Criteria.

4.1.3 Length-n Event-sequence Coverage

In certain cases, the behavior of events may change when performed in different contexts. In such cases, event coverage and event-interaction coverage on their own are weak requirements for sufficient testing. A criterion that captures the contextual impact is defined next. Intuitively, the context for an event e is the sequence of events performed before e. Formally, context is defined as:

Definition: The *context* of an event e_n in the event-sequence $< e_1, e_2, e_3, \ldots, e_n, \ldots >$ is $< e_1, e_2, e_3, ..., e_{n-1} >$. □

An event may be performed in an infinite number of contexts. For finite applicability, a limit is imposed on the length of the event-sequence. Hence, the *length-n event-sequence criterion* is defined as:

Definition: A set P of event-sequences satisfies the *length-n event-sequence coverage criterion* if and only if P contains all event-sequences of length equal to n. □

This criterion to similar to the *length-n path coverage criterion* defined by Gourlay for conventional software [29], which requires coverage of all subpaths in the program's flow-graph of length less than or equal to n. As the length of the event-sequence increases, the number of possible contexts also increases.

4.1.4 Subsumption

A coverage criterion $\mathcal{C}_\infty$ **subsumes** criterion $\mathcal{C}_\in$ if every test suite that satisfies $\mathcal{C}_\infty$ also satisfies $\mathcal{C}_\in$ [68]. Since event coverage and event-interaction coverage are special cases of length-n event-sequence coverage, i.e., length 1 event-sequence and length 2 event-sequence coverage respectively, it follows that length-n event-sequence coverage **subsumes** event and event-interaction coverage. Moreover, if a test suite satisfies event-interaction coverage, it must also satisfy event coverage. Hence, event-interaction **subsumes** event coverage. The **subsume** relationship between the coverage criteria is summarized in Figure 4.1. The nodes represent the criteria whereas the edges represent the subsume relation. Note that the figure also shows inter-component coverage criteria (in reverse color). The relationships among these criteria is presented in the next section.

4.2 Inter-component Criteria

The goal of inter-component coverage criteria is to ensure that all interactions among components are tested. In GUIs, the interactions take the form of invocation of components, termination of components, and more generally, event-sequences that start with an event in one component and end with an event in another component.

4.2.1 Invocation Coverage

Intuitively, invocation coverage requires that each restricted-focus event in the GUI be performed at least once. Such a requirement is necessary to check whether each component can be invoked.

Definition: A set P of event-sequences satisfies the *invocation coverage criterion* if and only if for all restricted-focus events $i \in \mathcal{I}$, where $\mathcal{I}$ is the set of all restricted-focus events in the GUI, there is at least one event-sequence $p \in P$ such that event i is in p. □

Note that event coverage subsumes invocation coverage (Figure 4.1) since it requires that *all* events be performed at least once, including restricted-focus events.

4.2.2 Invocation-termination Coverage

It is important to check whether a component can be invoked and terminated.

Definition: The invocation-termination set $\mathcal{IT}$ of a GUI is the set of all possible length 2 event sequences $< e_i, e_j >$, where e_i invokes component C_x and e_j terminates component C_x, for all components $C_x \in \mathcal{N}$. □

Intuitively, the invocation-termination coverage requires that all length 2 event sequences consisting of a restricted-focus event followed by the invoked component's termination events be tested.

Definition: A set P of event-sequences satisfies the *invocation-termination coverage criterion* if and only if for all $i \in \mathcal{IT}$, there is at least one event-sequence $p \in P$ such that i is in p. □

Satisfying the invocation-termination coverage criterion assures that each component is invoked at least once and then terminated immediately, if allowed by the GUI's specifications. For example, in WordPad, the component `FileOpen` is invoked by the event `Open` and terminated by either `Open` or `Cancel`. Note that WordPad's specification do not allow `Open` to terminate the component unless a file has been selected. On the other hand, `Cancel` can always be used to terminate the component.

4.2.3 Inter-component Length-n Event-sequence Coverage

Finally, the inter-component length-n event-sequence coverage criterion requires testing all event-sequences that start with an event in one component and end with an event in another component. Note that such an event-sequence may use events from a number of components. A criterion is defined to cover all such interactions.

Definition: A set P of event-sequences satisfies the *inter-component length-n event-sequence coverage criterion* for components C_1 and C_2 if and only if P contains all length-n event-sequences $v_1; v_2; v_3; \ldots; v_n$ such that $v_1 \in Vertices(C_1)$ and $v_n \in Vertices(C_2)$. Events $v_2, v_3, ..., v_{n-1}$ may belong to C_1 or C_2 or any other component C_i. □

Note that the inter-component length-n event-sequence coverage subsumes invocation-termination coverage (Figure 4.1) since length-n event sequences also include length 2 sequences.

4.3 Evaluating Coverage

Now that intra- and inter-component coverage criteria have been formally defined, the remaining question is how to evaluate the coverage of a test suite using these criteria. In this section, algorithms to evaluate the coverage of the GUI for a given test suite are presented.

4.3.1 Evaluating Intra-component Coverage

Given an event-flow graph for a component, the intra-component coverage of a given test suite may be evaluated using the elements of this graph. Figure 4.2 shows a dynamic programming algorithm to compute the percentage of length-n event-sequences

```
$\mathbf{ALGORITHM}$ : ComputePercentageTested(
$\mathbf{S}$: $\textit{Set of Components;}$
$\mathbf{T}$: $\textit{Test Suite;}$
$\mathbf{M}$: $\textit{Maximum Event-sequence Length}$)
{$\mathbf{count} \longleftarrow$ ComputeCounts($\mathbf{T}, \mathbf{S}, \mathbf{M}$);
/* $count_{i,j}$ is the tested number
of length-j event-sequences in component i */
$\mathbf{total} \longleftarrow$ ComputeTotals($\mathbf{S}, \mathbf{M}$);
/* $total_{i,j}$ is the total number
of length-j event-sequences in component i */
FOREACH $\mathbf{i} \in \mathbf{S}$ DO
  FOR $\mathbf{j} \longleftarrow 1$ TO $\mathbf{M}$ DO
    $\mathbf{Matrix}_{i,j} \longleftarrow (\mathbf{count}_{i,j}/\mathbf{total}_{i,j}) \times 100$;
return($\mathbf{Matrix}$)}
$\mathbf{SUBROUTINE}$ : ComputeCounts(
$\mathbf{T}$: $\textit{Test Suite;}$ $\mathbf{S}$: $\textit{Set of Components;}$
$\mathbf{M}$: $\textit{Maximum Event-sequence Length}$)
{
FOREACH $\mathbf{i} \in \mathbf{S}$ DO
  $\mathbf{A} \longleftarrow \{\}$; /* Empty Set */
  FOREACH $\mathbf{t} \in \mathbf{T}$ DO
    FOR $\mathbf{k} \longleftarrow 1$ TO $|\mathbf{t}|$ DO
      FOR $\mathbf{j} \longleftarrow \mathbf{k}$ TO $|\mathbf{t}|$ DO
        $\mathbf{A} \longleftarrow \mathbf{A} \cup \{< t_k ... t_j >\}$
  FOR $\mathbf{j} \longleftarrow 1$ TO $\mathbf{M}$ DO
      /* count number of sets of length j */
    $\mathbf{count}_{i,j} \longleftarrow$ NumberOfSetsOfLength($\mathbf{S}, \mathbf{j}$);
return($\mathbf{count}$)}
$\mathbf{SUBROUTINE}$ : ComputeTotals(
$\mathbf{S}$: $\textit{Set of Components;}$
$\mathbf{M}$: $\textit{Maximum Event-sequence Length}$)
{FOREACH $\mathbf{j} \in \mathbf{S}$ DO
  $\mathbf{E} \longleftarrow$ Edges($\mathbf{j}$);
  $\mathbf{V} \longleftarrow$ Vertices($\mathbf{j}$);
  FOREACH $\mathbf{i} \in \mathbf{V}$ DO
    $\mathbf{freq}_i \longleftarrow 1$;
  $\mathbf{total}_{1,j} \longleftarrow |\mathbf{V}|$;
  FOREACH $\mathbf{i} \in \mathbf{V}$ DO
    $\mathbf{newfreq}_i \longleftarrow 0$;
  FOR $\mathbf{k} \longleftarrow 2$ TO $\mathbf{M}$ DO
    FOREACH $\mathbf{i} \in \mathbf{V}$ DO
      $\mathbf{x} \longleftarrow$ follows($\mathbf{i}$);
      $\mathbf{total}_{j,k} \longleftarrow \mathbf{total}_{j,k} + |\mathbf{x}| \times \mathbf{freq}_i$;
      FOREACH $\mathbf{l} \in \mathbf{x}$ DO
        $\mathbf{newfreq}_j ++$;
    $\mathbf{freq} \longleftarrow \mathbf{newfreq}$;
    FOREACH $\mathbf{i} \in \mathbf{V}$ DO
      $\mathbf{newfreq}_i \longleftarrow 0$;
return($\mathbf{total}$)}
```

Figure 4.2: Computing Percentage of Tested Length-n Event-sequences of All Components.

tested. The final result of the above algorithm is **Matrix**, where $\mathbf{Matrix}_{i,j}$ is the percentage of length-**j** event-sequences tested on component **i**. Intuitively, the algorithm breaks a test case of length-n into all possible test cases of length $n-1$, $n-2$, $n-3$, and so on, and counts them. It stores this result in a matrix **count**, where $\mathbf{count}_{i,j}$ is the tested number of length-j event-sequences in component i. The algorithm also computes the total number of length-j event-sequences in component i and stores it in a matrix $total_{i,j}$. It uses `follows` to count the paths in the event-flow graph starting from each vertex.

The main algorithm is `ComputePercentageTested`. In this algorithm, two matrices are computed (`line 6,7`). $\mathbf{Count}_{i,j}$ is the number of length-**j** event-sequences in component **i** that have been covered by the test suite **T** (`line 6`). $\mathbf{Total}_{i,j}$ is the total number of all possible length-**j** event-sequences in component **i** (`line 7`). The subroutine `ComputeCounts` calculates the elements in **count** matrix. For each test case in **T**, `ComputeCounts` finds all possible event-sequences of different lengths (`line 19..21`). The number of event-sequences of each length are counted (`lines 22, 23`). Note that since `ComputeCounts` takes a union of the event sequences, there is no danger of counting the same event sequence twice. Intuitively, the `ComputeTotals` subroutine starts with single-length event-sequences, i.e., individual events in the GUI (`lines 31..33`). Using `follows` (`line 38`), the event-sequences are lengthened one event at each step. A counter keeps track of the number of event-sequences created (`line 39`). For every element in the `follow_set` of **i**, the frequency counter **newfreq** is incremented (`lines 40..41`), hence counting the total number of outgoing edges in the event-flow graph.

The result of the algorithm is **Matrix**, the entries of which can be interpreted as follows:

Event Coverage requires that individual events in the GUI be exercised. These individual events correspond to length 1 event-sequences in the GUI. $\mathbf{Matrix}_{j,1}$ $j \in S$ represents the percentage of individual events covered in each component.

Event-interaction Coverage requires that all the edges of the event-flow graph be covered by at least one test case. Each edge is effectively captured as a length 2 event-sequence. $\mathbf{Matrix}_{j,2}$ $j \in S$ represents the percentage of branches covered in each component **j**.

Length-n Event-sequence Coverage is available directly from **Matrix**. Each column **i** of **Matrix** represents the number of length-**i** event-sequence in the GUI.

4.3.2 Evaluating Inter-component Coverage

The integration tree may be used in several ways to identify interactions among components. For example, in Figure 3.12 a subset of all possible pairs of components that interact would be { (Main, FileNew), (Main, FileOpen), (Main, Print), (Main, FormatFont), (Print, Properties) }. To identify sequences such as the ones from Main to Properties, the integration tree is traversed in a bottom-up manner, identifying interactions among Print and Properties. Then Print and Properties are merged to form a *super-component* called PrintProperties. Then interactions among Main and PrintProperties are checked. This process continues until all components have been merged into a single super-component. Evaluating the inter-component coverage of a given test suite requires computing the (1) invocation coverage, (2) invocation-termination coverage, and (3) length-n event sequence coverage.

The total number of length 1 event sequences required to satisfy the invocation coverage criterion is equal to the number of restricted-focus events available in the GUI. The percentage of restricted-focus events actually covered by the test cases is $(x/\mathcal{I}) \times 100$, where x is the number of restricted-focus events in the test cases, and $\mathcal{I}$ is the total number of restricted-focus events available in the GUI. Similarly, the total number of length 2 event sequences required to satisfy the invocation-termination criterion is $\sum(I_i \times T_i)$, where I_i and T_i are the number of restricted-focus and termination events that invoke and terminate component C_i respectively. The percentage of invocation-termination pairs actually covered by the test cases is $(x/\sum(I_i \times T_i)) \times 100$, where x is the number of invocation-termination pairs in the test cases.

Computing the percentage of length-n event sequences is slightly more complex. The algorithm shown in Figure 4.3 computes the percentage of length-n event sequences tested among GUI components. Intuitively, the algorithm obtains the number of event sequences that end at a certain restricted-focus event. It then counts the number of event sequences that can be extended from these sequences into the invoked component. The main algorithm called Integrate is recursive and performs a bottom-up traversal of the integration tree **T** (line 2). Other than the recursive call (line 8), Integrate makes a call to ComputeTotalInteractions that takes two components as parameters (lines 13,14). It initializes the vector **Total** for all path lengths **i** ($1 \leq \mathbf{i} \leq \mathbf{M}$) (line 16,17). Assuming that a **freq** matrix has been stored for each component from the **freq** vector of the algorithm in Figure 4.2, i.e., $\mathbf{freq}_{i,j}$ is the number of event-sequences that start with event i and end with event j. After obtaining both frequency matrices for both $\mathbf{C}_1$ and $\mathbf{C}_2$, for all path lengths (lines 21,26), the new vector **Total** is obtained by adding the

```
ALGORITHM : Integrate(                                          1
T: Integration Tree)                                            2
{                                                               3
IF Leaf(T)                                                      4
  return(T);                                                    5
newT ⟵ T;                                                       6
FORALL c ∈ Children(T) DO                                       7
  Integrate(c);                                                 8
  ComputeTotalInteractions(newT, c);                            9
  $\mathbf{Matrix}_{newT+c}$ ⟵ $\mathbf{TestedEventSeq}_{newT+c}$/Total;   10
}                                                               11
SUBROUTINE : ComputeTotalInteractions(                          12
$C_1$: Component 1;                                             13
$C_2$: Component 2)                                             14
{                                                               15
FOR i ⟵ 1 TO M DO                                               16
  $\mathbf{Total}_i$ ⟵ 0;                                       17
x ⟵ GetCallingEvent($C_1$, $C_2$);                              18
FOR i ⟵ 1 TO M DO                                               19
   /* get freq table of $C_1$ for event-seq of length i */      20
  $F_1$ ⟵ GetFreqTable($C_1$, i);                               21
   /* Add all values in column x */                             22
  p ⟵ addColumn(x, $F_1$);                                      23
  FOR j ⟵ 1 TO M DO                                             24
   /* get freq table of $C_2$ for event-seq of length j */      25
    $F_2$ ⟵ GetFreqTable($C_2$, j);                             26
    q ⟵ 0;                                                      27
    FOREACH k ∈ B of $C_2$ DO                                   28
      q ⟵ q + addRow(k, $F_2$);                                 29
    $\mathbf{Total}_{i+j}$ ⟵ $\mathbf{Total}_{i+j}$ + p × q;    30
 ComputeFreqMatrix($C_1$, $C_2$);                               31
return(Total);                                                  32
}
```

Figure 4.3: Computing Percentage of Tested Length-n Event-sequences of All Components.

frequency entries from $\mathbf{F}_1$ and $\mathbf{F}_2$ (`lines 28..30`). A new frequency matrix is computed for the super-component "$\mathbf{C}_1\mathbf{C}_2$" (`line 31`). This new frequency matrix will be utilized by the same algorithm to integrate "$\mathbf{C}_1\mathbf{C}_2$" to other components.

The results of the above algorithm are summarized in **Matrix**. $\mathbf{Matrix}_{i,j}$ is the percentage of length-**j** event-sequences that have been tested in the super-component represented by the label **i**.

4.4 Implementation and Experiments

Two experiments were performed on the example WordPad to determine the (1) total number of event sequences required to test the GUI and hence enable a test designer to compute the percentage of event sequences tested, and (2) correlation between event-based coverage of the GUI and statement coverage of the underlying code.

The coverage evaluation algorithms were implemented in C. They were executed on a 300MHz Pentium-based computer with 256MB of RAM. In this experiment, specifications and a new implementation of the WordPad software was used. The software consists of 36 modal windows, and 362 events (not counting short-cuts).

4.4.1 Computing Total Number of Event-sequences for WordPad

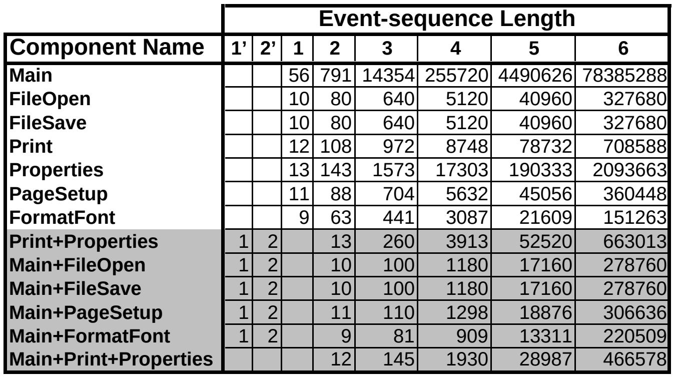
The purpose of the first experiment was to determine the total number of event sequences required to test WordPad with respect to the new coverage criteria. The following steps were performed:

Identifying Components and Events: Individual WordPad components and events within each component were identified. Table 3.1 shown earlier lists some of the components of WordPad that were used in this experiment.

Creating Event-flow Graphs: The next step was to construct event-flow graphs for the GUI. Figure 3.10 shows the event-flow graph of the `Main` component of WordPad. Recall that each node in the event-flow graph represents an event.

Computing Event-sequences: Once the event-flow graphs were available, the total number of possible event-sequences of different lengths in each component were computed using the `computeTotals` subroutine in Figure 4.2. Note that these event-sequences may also include infeasible event-sequences. The total number of event-sequences is shown in Table 4.1. The rows represent the components and the shaded rows represent the inter-component interactions. The columns represent different event-sequence lengths. Recall that an event-sequence of length 1 represents event coverage whereas an event-sequence of length 2 represents event-interaction coverage. The columns 1' and 2' represent invocation and invocation-termination coverage respectively.

The results of the first experiment show that, not surprisingly, the total number of event sequences grows with increasing length. Note that longer sequences subsume shorter sequences; e.g., if all event sequences of length 5 are tested, then so are all sequences of length-i, where $i \leq 4$. It is difficult to determine the maximum length of event sequences needed to test a GUI. The large number of event sequences show that it is impractical to test a GUI for all possible event sequences. Rather, depending on the resources, a subset of "important" event sequences should be identified, generated and executed. Identifying such important sequences requires that they be ordered by assigning a *priority* to each event sequence. For example, event sequences that are performed in the `Main` component



	Event-sequence Length							
Component Name	**1’**	**2’**	**1**	**2**	**3**	**4**	**5**	**6**
Main			56	791	14354	255720	4490626	78385288
FileOpen			10	80	640	5120	40960	327680
FileSave			10	80	640	5120	40960	327680
Print			12	108	972	8748	78732	708588
Properties			13	143	1573	17303	190333	2093663
PageSetup			11	88	704	5632	45056	360448
FormatFont			9	63	441	3087	21609	151263
Print+Properties	1	2		13	260	3913	52520	663013
Main+FileOpen	1	2		10	100	1180	17160	278760
Main+FileSave	1	2		10	100	1180	17160	278760
Main+PageSetup	1	2		11	110	1298	18876	306636
Main+FormatFont	1	2		9	81	909	13311	220509
Main+Print+Properties				12	145	1930	28987	466578

Table 4.1: Total Number of Event-sequences for Selected Components of WordPad. Shaded Rows Show Number of Interactions Among Components.

may be given higher priority since they will be used more frequently; all the users start interacting with the GUI using the `Main` component. The components that are deepest in the integration tree may be used the least. This observation leads to a heuristic for ordering the testing of event sequences within components of the GUI. The structure of the integration tree may be used to assign priorities to components; `Main` will have the highest priority, decreasing for components at the second level, with the deepest components having the lowest priority. A large number of event sequences in the high priority components may be tested first; the number will decrease for low priority components.

4.4.2 Correlation Between Event-based Coverage and Statement Coverage

The second experiment was performed to determine exactly what percentage of the underlying code is executed when event-sequences of increasing length are executed on the GUI, and how code coverage relates to event coverage. The following steps were performed:

Code Instrumentation: The underlying code of WordPad was instrumented to produce a *statement trace*, i.e., a sequence of statements in the order in which they are executed. Examining such a trace allowed determining which statements are executed by a test case.

Event-sequence Generation: All event-sequences up to a specific length were generated. `ComputeTotals` in Figure 4.2 was modified to output the event sequences as they were obtained. This change resulted in an event-sequence generation algorithm that constructs event sequences of increasing length. The dynamic programming algorithm constructs all event sequences of length 1. It then uses `follows` to extend each event sequence by one event, hence creating all length 2 event-sequences. All event-sequences up to length 3 were obtained; in all, 21659 event-sequences were obtained.

Controlling GUI's State: As mentioned earlier in Section 1.2, the controllability problem also occurs in GUIs, and for each test case, appropriate events may need to be performed on the GUI to bring it to a desired state S_i. This sequence of events is called the *prefix*, P_i, of the test case. Although generating the prefix in general may require the development of expensive solutions, a heuristic was used for this experiment. Each test case was executed in a fixed state S_0 in which WordPad contains text, part of the text was highlighted, the clipboard contains a text object, and the file system contains two text files. The event-flow graphs and the integration tree were traversed to produce the prefix of each test case. Note that using this heuristic may render some of the event sequences non-executable because of infeasibility. However, the results of this experiment will show that although infeasible sequences do exist, they are of no consequence to the results of this experiment. WordPad was modified so that no statement trace was produced for P_i.

Test-case Execution: After all the event-sequences up to length 3 were obtained, they were executed on the GUI using the automated test executor. Execution traces were collected during the test runs. The test case executor executed without any intervention for 30 hours. Note that 4189 (or 19.3%) of the test cases could not be executed because of infeasibility. These infeasible sequences were detected during test case execution.

Analysis: The traces were analyzed to determine the number of statements that were executed by event-sequences of length 1, 2, and 3. The graph in Figure 4.4 shows that almost 92% of the statements were executed by just individual events. As the length of the event sequences increased, very few new statements were executed (5%). Hence, a high statement coverage of the underlying code may be obtained by executing short event sequences.

The relationship between event sequences and code, obtained from this experiment, can be explained in terms of the design of the WordPad GUI. Since the GUI is an event-driven software, a method called an *event handler* is implemented for each event. Executing an event caused the execution of its corresponding event handler. Code inspection of the WordPad implementation revealed that there were few or no branch statements in the code of the event handler. Consequently, when an event was performed, most of the statements in the event-handler were executed. Hence high statement coverage was obtained by just

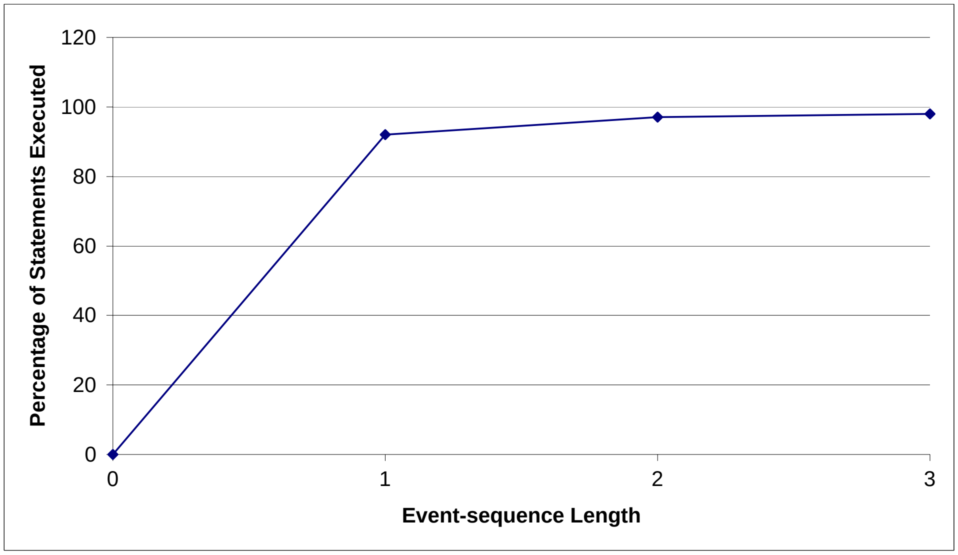


Figure 4.4: The Correlation Between Event-based Coverage and Statement Coverage of WordPad.

performing individual events. Whether other GUIs exhibit similar behavior requires a detailed analysis of a number of GUIs and their underlying code.

The result shows that statement coverage of the underlying code can be a misleading coverage criterion for GUI testing. A test designer who relies on statement coverage of the underlying code for GUI testing may test only short event sequences. However, testing only short sequences is not enough. Longer event sequences lead to different states of the GUI and that testing these sequences may help detect a larger number of faults than short event sequences. For example, in WordPad, the event `Find Next` (obtained by clicking on the `Edit` menu) can only be executed after at least 6 events have been performed; the shortest sequence of events needed to execute `Find Next` is `Edit; Find; TypeInText; FindNext2; OK; Edit; Find Next`, which has 7 events. If only short sequences (< 3) are executed on the GUI, a bug in `Find next` may not be detected. Extensive studies of the fault-detection capabilities of executing short and long event sequences for GUI testing are needed and are targeted for future work. Another possible extension to this experiment is to determine the correlation between event-based coverage and other code-based coverage, e.g., branch coverage.

4.5 Conclusions

In this chapter, new coverage criteria for GUI testing based on GUI events and their interactions were presented. Three new coverage criteria for events within a component were defined: event coverage, event-interaction coverage, and length-n event-sequence coverage. Invocation coverage, invocation-termination coverage, and inter-component length-n event-sequence coverage were defined for events among components. Algorithms were provided to evaluate the coverage of a given test suite. Experiments were performed on the example Wordpad showing the number of event sequences required to test a part of Wordpad and to demonstrate the correlation between event coverage and code coverage.

Chapter 5
Test Case Generator

The test case generator provides input to test the GUI. As described in Section 3.8, the input is in the form of test cases consisting of a legal sequence of events $e_1; e_2; e_3; \ldots; e_n$ executed on the GUI starting in a specific reachable state S_0, called the *initial state for the test case*. This chapter presents the design of a test case generator.

Designing the test case generator requires that it should exploit the component hierarchy of the GUI to generate test cases so that the test case generation process is scalable and that the test cases generated for a specific component are usable across multiple GUIs that employ the same component. Moreover, it should employ the high-level representation of events so that the generated test cases are free from platform-specific details, making them portable across platforms.

In principle, an infinite number of event sequences may be performed on a GUI. Depending on the resources available, a manageable number of these event sequences should be generated as test cases and tested on the GUI. There are various possible approaches to automatically generate test cases for GUIs, including the following:

1. *Random:*
   This approach randomly generates sequences of GUI events. Although straightforward to implement, this approach may yield a large number of event sequences that are not legal and hence not executable, wasting valuable resources. Moreover, since the test designer has no control over choice of event sequences, they may not have acceptable test coverage.
2. *Structural:*
   This approach generates legal event sequences by employing the structure of the GUI, represented by event-flow graphs and an integration tree. Recall that this approach was used in an experiment in Section 4.4.2 to generate short event sequences. Even in this controlled experiment, almost 20% of the event-sequences were not executable

because of infeasibility. As the length of the event sequences increases, the number of infeasible event sequences may become unacceptably large.

3. *Commonly-used Tasks:*

   In this approach, the test designer identifies commonly used tasks for the GUI; these are then input to the test case generator. The generator employs the GUI representation 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. Note that a similar approach is used to manually perform usability testing of the GUI [94]. 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 chapter uses an automated technique to generate GUI test cases for commonly used tasks.

   Note that test cases generated for commonly used tasks may not satisfy any of the structural coverage criteria defined in Chapter 4. In fact, the underlying philosophies of testing software using its structure vs. commonly used tasks are fundamentally different. The former tests software for event sequences as dictated by the software's structure whereas the latter determines whether the software executes correctly for commonly used tasks. Both testing methods are valuable and may be used to uncover different types of errors. The structural coverage criteria may be used to determine the structural coverage of test cases generated for commonly used tasks; missing event sequences may then be generated using a structural test case generation technique.

This chapter presents details of an approach that uses AI planning to generate test cases for GUIs. The test designer provides a specification of initial and goal states for commonly used tasks. An automated planning system generates plans for each specified task. Each generated plan represents a test case that is a reasonable candidate for helping test the GUI, because it reflects an intended use of the system.

This technique of using planning for test case generation is called Planning Assisted Testing (PAT). The test case generator is called Planning Assisted Tester for grapHical user interface Systems (PATHS). The test case generation process is partitioned into two

Phase	Step	Test Designer	PATHS
`Setup`	1		Derive Planning Operators from the GUI representation
	2	Define Preconditions and Effects of Operators	
`Plan Generation`	3	Identify a Task $\mathcal{T}$	
	4		Generate Test Cases for $\mathcal{T}$

Iterate 3 and 4 for Multiple Scenarios

Table 5.1: Roles of the Test Designer and PATHS During Test Case Generation.

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, PATHS 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. Table 5.1 summarizes the tasks assigned to the test designer and those performed by PATHS.

The remainder of this chapter presents the design of PATHS. In particular, the derivation of planning operators and how AI planning techniques are used to generate test cases is described. An algorithm that performs a restricted form of hierarchical planning is presented that employs new hierarchical operators and leads to an improvement in planning efficiency and to the generation of multiple alternative test cases. The algorithm has been implemented in PATHS, and Section 5.4 presents the results of experiments in which test cases for the example WordPad system were generated.

5.1 Setting up the Planning Problem

As described in Section 2.6, setting up a planning problem requires performing two related activities: (1) defining planning operators in terms of preconditions and effects, and (2) describing tasks in the form of initial and goal states. This section provides details of these two activities in the context of using planning for test case generation.

5.1.1 Modeling Planning Operators

For a given GUI, the simplest approach to obtain planning operators would be to identify one operator for each GUI event (`Open`, `File`, `Cut`, `Paste`, etc.) directly from the GUI representation, ignoring the GUI's component hierarchy. For the remainder of this chapter, these operators, presented earlier in Section 3.3, are called *primitive operators*. When developing the GUI representation, the test designer defines the preconditions and effects for all these operators. Although conceptually simple, this approach is inefficient for generating test cases for GUIs as it results in a large number of operators.

An alternative modeling scheme, and the one used in this test case generator, uses the component hierarchy and creates high-level operators that are decomposable into sequences of lower level ones. These high-level operators are called *system-interaction operators* and *component operators*. The goal of creating these high-level operators is to control the size of the planning problem by dividing it into several smaller planning problems. Intuitively, the system-interaction operators fold a sequence of menu-open or unrestricted-focus events and a system-interaction event into a single operator, whereas component operators encapsulate the events of the component by treating the interaction within that component as a separate planning problem. Component operators need to be decomposed into low-level plans by an explicit call to the planner. Details of these operators are presented next.

The first type of high-level operators are called system-interaction operators.

Definition: A *system-interaction operator* is a single operator that represents a sequence of zero or more menu-open and unrestricted-focus events followed by a system-interaction event. □

Consider a small part of the WordPad GUI: one pull-down menu with one option (`Edit`) which can be opened to give more options, i.e., `Cut` and `Paste`. The events available to the user are `Edit`, `Cut` and `Paste`. `Edit` is a menu-open event, and `Cut` and `Paste` are system-interaction events. Using this information the following two system-interaction operators are obtained.

```
EDIT_CUT   = <Edit, Cut>
EDIT_PASTE = <Edit, Paste>
```

The above is an example of an *operator-event mapping* that relates system-interaction operators to GUI events. The operator-event mappings fold the menu-open and unrestricted focus events into the system-interaction operator, thereby reducing the total number of operators made available to the planner, resulting in planning efficiency. These mappings are used to replace the system-interaction operators by their corresponding GUI events when generating the final test case.

In the above example, the events `Edit`, `Cut` and `Paste` are hidden from the planner, and only the system-interaction operators, namely, `EDIT_CUT` and `EDIT_PASTE`, are made available to the planner. This abstraction prevents generation of test cases in which `Edit` is used in isolation, i.e., the model forces the use of `Edit` either with `Cut` or with `Paste`, thereby restricting attention to meaningful interactions with the underlying software.[1]

The second type of high-level operators are called *component operators*.

Definition: A *component operator* encapsulates the events of the underlying component by creating a new planning problem and its solution represents the events a user might generate during the focused interaction. □

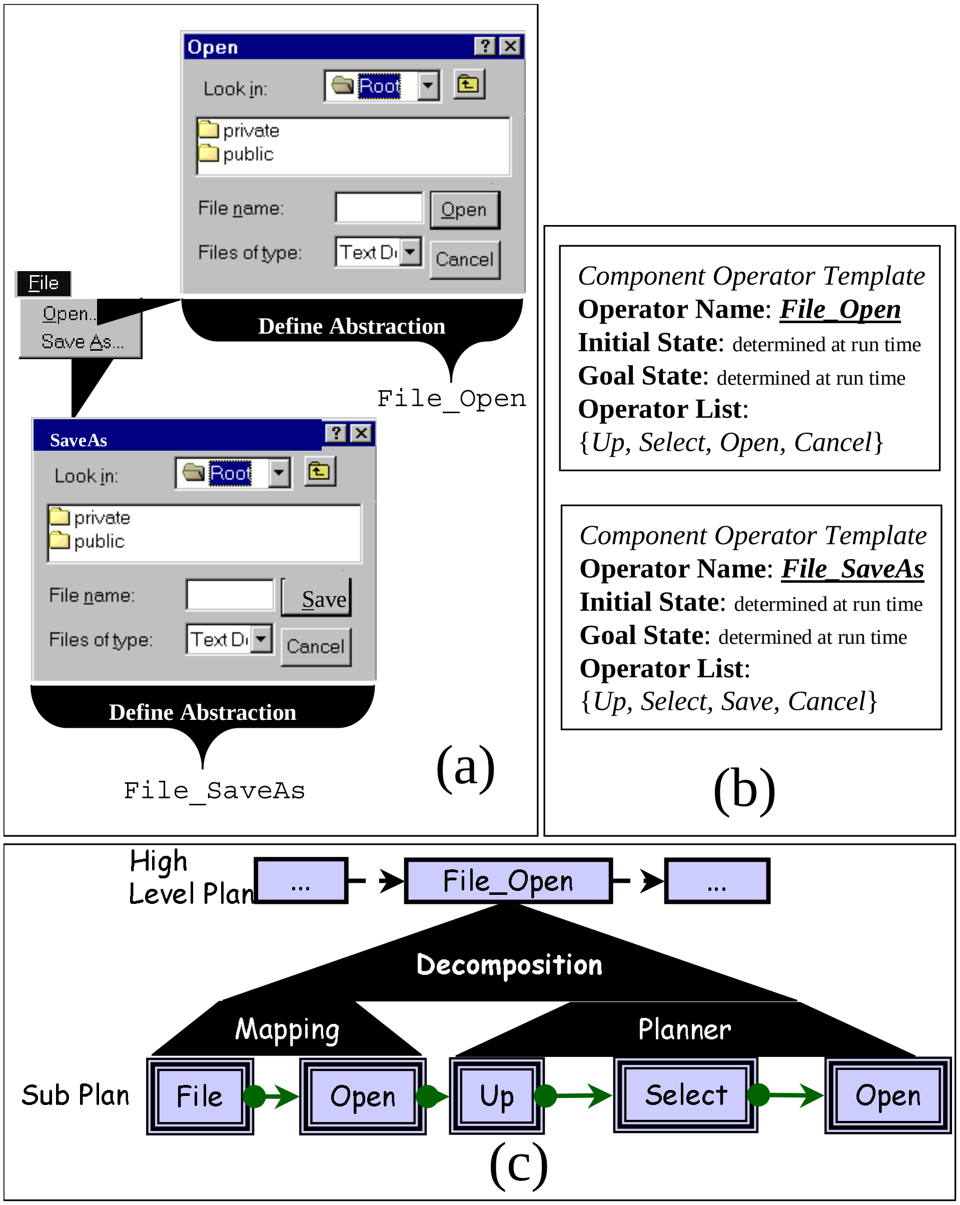
The component operators employ the component hierarchy of the GUI so that test cases can be generated for each component, thereby resulting in greater efficiency. For example, consider a small part of the WordPad's GUI shown in Figure 5.1(a): a `File` menu with two restricted-focus events, namely `Open` and `SaveAs`. Both these events invoke two components called `Open` and `SaveAs` respectively. The events in both windows are quite similar. For `Open` the user can exit after pressing `Open` or `Cancel`; for `SaveAs` the user can exit after pressing `Save` or `Cancel`. For simplicity, assume that the complete set of events available is `Open`, `SaveAs`, `Open.Select`, `Open.Up`, `Open.Cancel`, `Open.Open`, `SaveAs.Select`, `SaveAs.Up`, `SaveAs.Cancel` and `SaveAs.Save`. (Note that the component name is used to disambiguate events.) Once the user selects `Open`, the focus is restricted to `Open.Select`, `Open.Up`, `Open.Cancel` and `Open.Open`. Similarly, when the user selects `SaveAs`, the focus is restricted to `SaveAs.Select`, `SaveAs.Up`, `SaveAs.Cancel` and `SaveAs.Save`. Two component operators called `File_Open` and `File_SaveAs` are obtained.

The component operator is a complex structure since it contains all the necessary elements of a planning problem, including the initial and goal states, the set of objects, and the set of operators. The *prefix* of the component operator is the sequence of menu-open and unrestricted-focus events that lead to the restricted-focus event, which invokes the component in question. This sequence of events is stored in the operator-event mappings. For the example of Figure 5.1(a), the following two operator-event mappings are obtained, one for each component operator:

`File_Open = <File, Open>`, and

`File_SaveAs = <File, SaveAs>`.

[1] Test cases in which `Edit` stands in isolation can be created by (1) testing `Edit` separately, or (2) inserting `Edit` at random places in the generated test cases.



Component Operator Template
Operator Name: ***File_Open***
Initial State: determined at run time
Goal State: determined at run time
Operator List:
{*Up, Select, Open, Cancel*}

Component Operator Template
Operator Name: ***File_SaveAs***
Initial State: determined at run time
Goal State: determined at run time
Operator List:
{*Up, Select, Save, Cancel*}

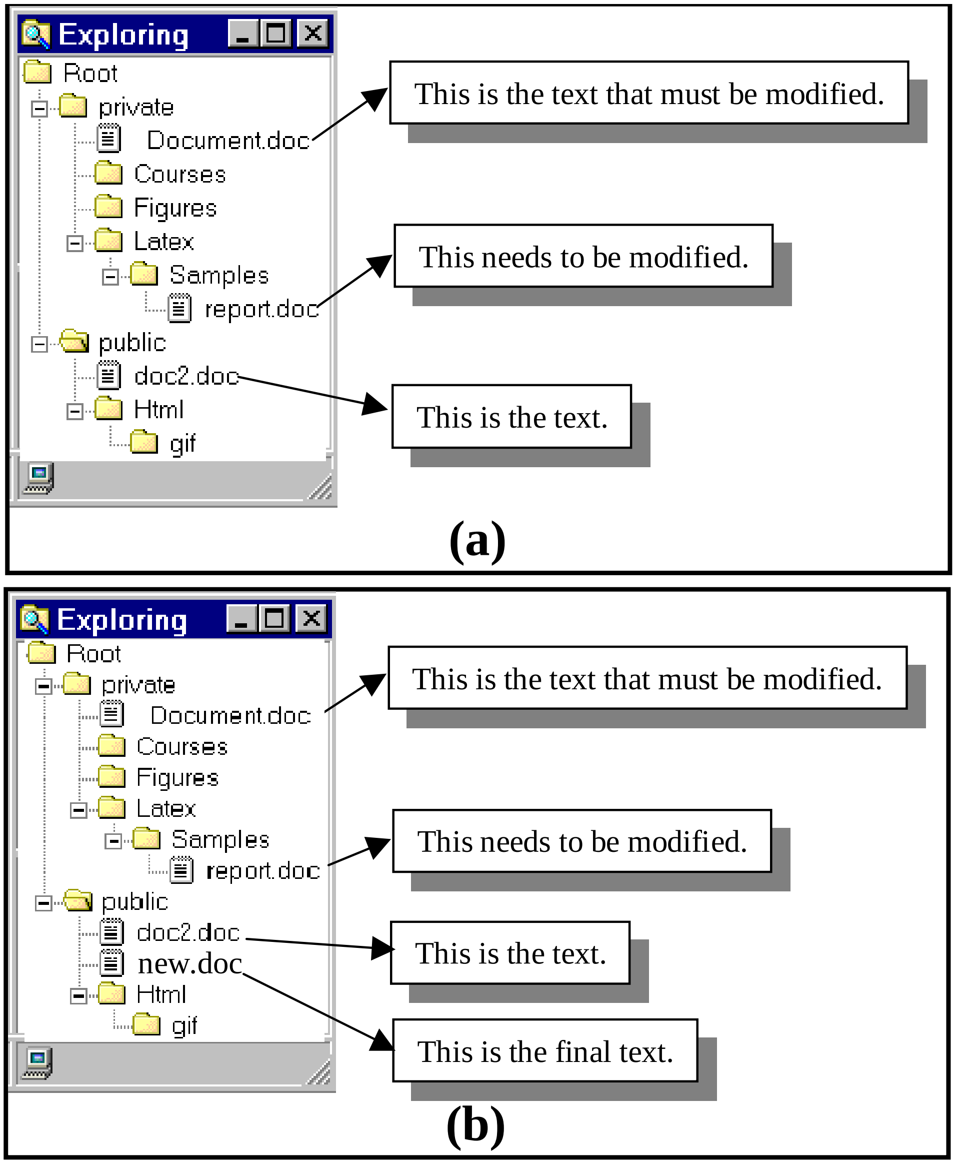
Figure 5.1: (a) `Open` and `SaveAs` Windows as Component Operators, (b) Component Operator Templates, and (c) Decomposition of the Component Operator Using Operator-event Mappings and Making a Separate Call to the Planner to Yield a Sub-plan.

The *suffix* of the component operator represents the modal dialog. A `component operator definition` template is created for each component operator. This template contains all the essential elements of the planning problem, i.e., the set of operators that are available during the interaction with the component and initial and goal states, both determined dynamically at the point before the call. The component operator definition template created for each operator is shown in Figure 5.1(b).

The component operator is decomposed in two steps: (1) using the operator-events mappings to obtain the component operator prefix, and (2) explicitly calling the planner to obtain the component operator suffix. Both the prefix and suffix are then substituted back into the high-level plan. At the highest level of abstraction, the planner will use the component operators, i.e., `File_Open` and `File_SaveAs`, to construct plans. For example, in Figure 5.1(c), the high-level plan contains `File_Open`. Decomposing `File_Open` requires (1) retrieving the corresponding GUI events from the stored operator-event mappings (`File`, `Open`), and (2) invoking the planner, which returns the sub-plan (`Up`, `Select`, `Open`). `File_Open` is then replaced by the sequence (`File`, `Open`, `Up`, `Select`, `Open`). Since the higher-level planning problem has already been solved before invoking the planner for the component operator, the preconditions and effects of the high-level component operator are used to determine the initial and goal states of the sub-plan.

5.1.2 Modeling the Initial and Goal State and Generating Test Cases

Once all the operators have been identified and defined, the test designer begins the generation of particular test cases by identifying a task, consisting of an initial state and a goal state. The test designer then codes these initial and goal states. Recall that GUI states are represented by a set of properties of GUI objects. Figure 5.2 shows an example of a task for WordPad. Figure 5.2(a) shows the `initial state`: a collection of files stored in a directory hierarchy. The contents of the files are shown in boxes, and the directory structure is shown in an `Exploring` window. Assume that the initial state contains a description of the directory structure, the location of the files, and the contents of each file. Using these files and WordPad's GUI, a goal of creating the new document shown in Figure 5.2(b) and then storing it in file *new.doc* in the `/root/public` directory is defined. Figure 5.2(b) shows this `goal state` that contains, in addition to the old files, a new file stored in `/root/public` directory. Note that *new.doc* can be obtained in numerous ways, e.g., by loading file `Document.doc`, deleting the extra text and typing in the word `final`, by loading file `doc2.doc` and inserting text, or by creating the document from scratch by typing in the text. The code for the initial state and the changes needed to achieve the goal states is shown in Figure 5.3. Once the task has been specified, the system automatically generates a set of test cases that achieve the goal.



(a)

This is the text that must be modified.

This needs to be modified.

This is the text.

(b)

This is the text that must be modified.

This needs to be modified.

This is the text.

This is the final text.

Figure 5.2: A Task for the Planning System; (a) the Initial State, and (b) the Goal State.

5.2 Generating Plans

The test designer begins the generation of particular test cases by inputing the defined operators into PATHS and then identifying a task, such as the one shown in Figure 5.2, that is defined in terms of an initial state and a goal state. PATHS automatically generates a set of test cases that achieve the goal. An example of a plan is shown in Figure 5.4. (Note that `TypeInText()` is a keyboard event.) This plan is a high-level plan that must be translated into primitive GUI events. The translation process makes use of the

```
Initial State:
isCurrent(root)
contains(root private)
contains(private Figures)
contains(private Latex)
contains(Latex Samples)
contains(private Courses)
contains(private Thesis)
contains(root public)
contains(public html)
contains(html gif)
containsfile(gif doc2.doc)
containsfile(private
            Document.doc)
containsfile(Samples
report.doc)
currentFont(Times Normal
            12pt)
in(doc2.doc This)
in(doc2.doc is)
in(doc2.doc the)
in(doc2.doc text.)
isText(This)
isText(is)
isText(the)
isText(text)
after(This is)
after(is the)
after(the text.)
font(This Times Normal 12pt)
font(is Times Normal 12pt)
font(the Times Normal 12pt)
font(text. Times Normal
            12pt)
…………….
Similar descriptions for
Document.doc and report.doc

Goal State:
containsfile(public new.doc)
in(new.doc This)
in(new.doc is)
in(new.doc the)
in(new.doc final)
in(new.doc text.)
after(This is)
after(is the)
after(the final)
after(final text.)
font(This Times Normal 12pt)
font(is Times Normal 12pt)
font(the Times Normal 12pt)
font(final Times Normal
            12pt)
font(text. Times Normal
            12pt)
……………….
```

Figure 5.3: Initial State and the changes needed to reach the Goal State.

operator-event mappings stored during the modeling process. One such translation is shown in Figure 5.5. This figure shows the component operators contained in the high-level plan are decomposed by (1) inserting the expansion from the operator-event mappings, and (2) making an additional call to the planner. Since the maximum time is spent in generating the high-level plan, it is desirable to generate a family of test cases from this single plan. This goal is achieved by generating alternative sub-plans at lower levels. One of the main advantages of using the planner in this application is to automatically generate alternative plans (or sub-plans) for the same goal (or sub-goal). Generating alternative plans is important to model the various ways in which different users might interact with the GUI, even if they are all trying to achieve the same goal. AI planning systems typically generate only a single plan; the assumption made there is that the heuristic search control rules will ensure

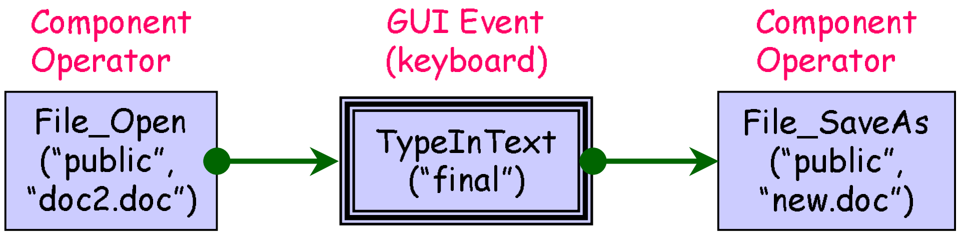


Figure 5.4: A Plan Consisting of Component Operators and a GUI Event.

that the first plan found is a high quality plan. PATHS generates alternative plans in the following two ways.

1. Generating multiple linearizations of the partial-order plans. Recall from an earlier discussion (Section 2.6) that the ordering constraints O only induce a partial ordering, so the set of solutions are all linearizations of S (plan steps) consistent with O. Any linear order consistent with the partial order is a test case. All possible linear orders of a partial-order plan result in a family of test cases. Multiple linearizations for a partial-order plan were shown earlier in Figure 2.3.

2. Repeating the planning process, forcing the planner to generate a different test case at each iteration.

The sub-plans are generated much faster than generating the high-level plan and can be substituted into the high-level plan to obtain alternative test cases. One such alternative low-level test case generated for the same task is shown in Figure 5.6. Note the use of nested invocations to the planner during component-operator decomposition.

5.3 Algorithm for Generating Test Cases

The test case generation algorithm is shown in Figure 5.7. The operators are assumed to be available before making a call to this algorithm, i.e., steps 1-3 of the test case generation process shown in Table 5.1 must be completed before making a call to this algorithm. The parameters (`lines 1..5`) include all the components of a planning problem and a threshold (**T**) that controls the looping in the algorithm. The loop (`lines 8..12`) contains the explicit call to the planner (Φ). The returned plan **p** is recorded with the operator set, so that the planner can return an alternative plan in the next iteration (`line 11`). At the end of this loop, **planList** contains all the partial-order plans. Each partial-order plan is then linearized (`lines 13..16`), leading to multiple linear plans. Initially the

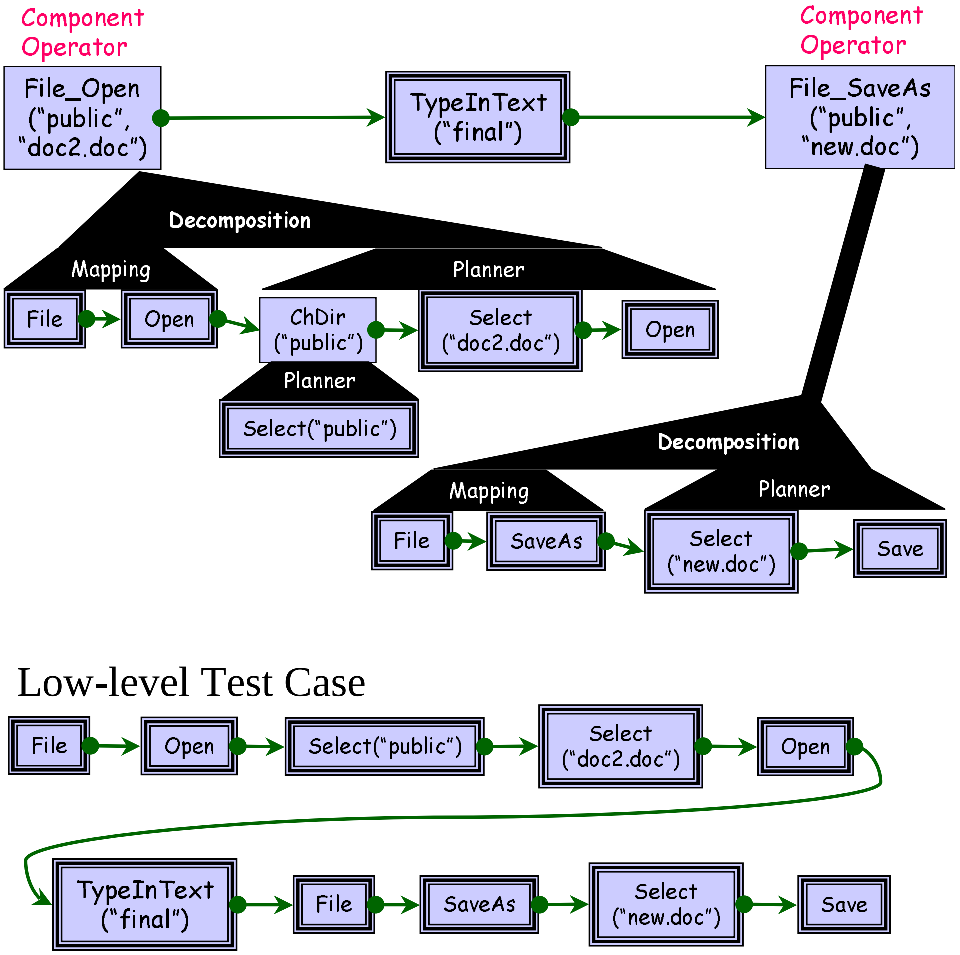


Figure 5.5: Expanding the Higher Level Plan.

test cases are high-level linear plans (`line 17`). The decomposition process leads to lower level test cases. The high-level operators in the plan need to be expanded/decomposed to get lower level test cases. If the step is a system-interaction operator, then the operator-event mappings are used to expand it (`lines 20..22`). However, if the step is a component operator, then it is decomposed to a lower level test case by (1) obtaining the GUI events from the operator-event mappings, (2) calling the planner to obtain the sub-plan, and (3) substituting both these results into the higher level plan. Extraction functions are used to access the planning problem's components (`lines 24..27`). The lowest level test cases, consisting of GUI events, are returned as a result of the algorithm (`line 33`).

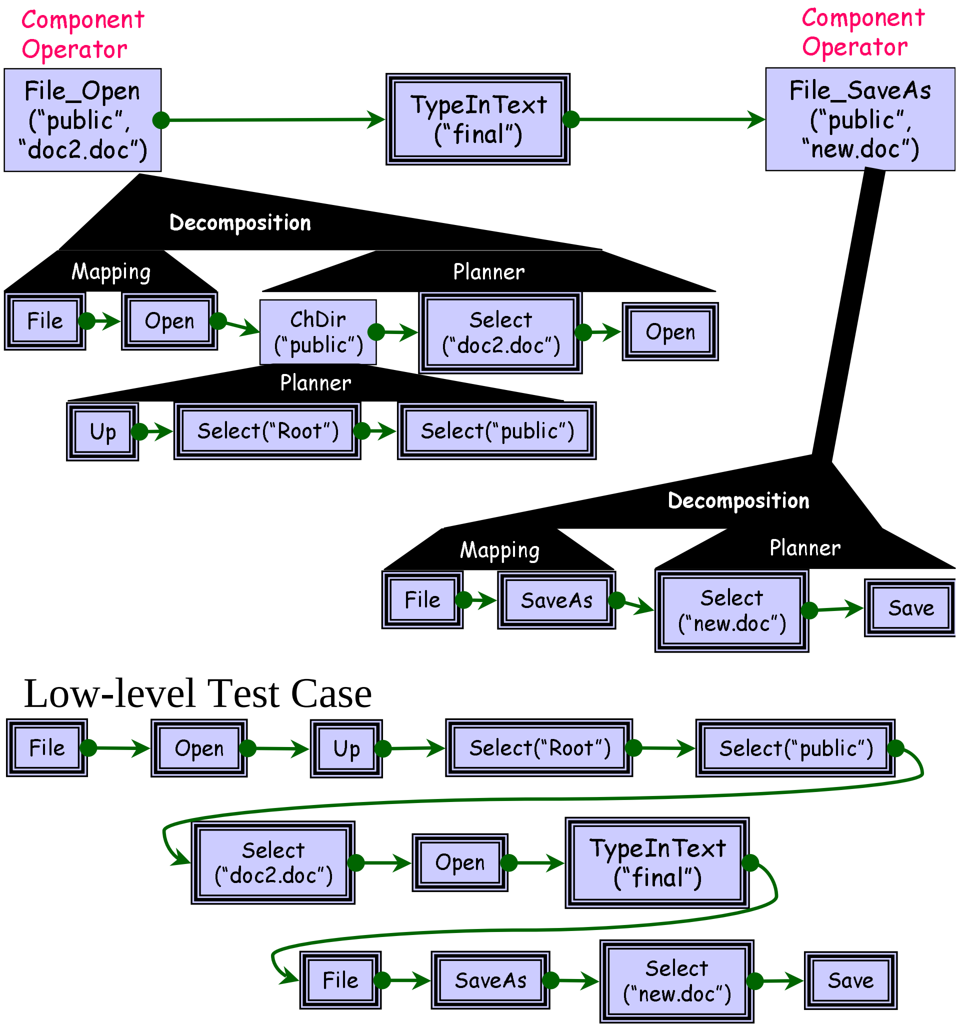


Figure 5.6: An Alternative Expansion Leads to a New Test Case.

5.4 Experiments

A prototype of PATHS was developed and several sets of experiments were conducted to determine whether PATHS is practical and useful. A summary of the results of these experiments is given in the following sections.

5.4.1 Generating Test Cases for Multiple Tasks

In this first experiment, PATHS was used to generate test cases for WordPad. This experiment was executed on a Pentium-based computer with 200MB RAM running Linux OS. Examples of the generated high-level test cases are shown in Table 5.2. The total number of GUI events in WordPad was determined to be approximately 362. Since

```
Algorithm :: GenTestCases(
  $\Lambda$ = Operator Set;
  D = Set of Objects;
  I = Initial State;
  G = Goal State;
  T = Threshold) {

  planList ← {};
  c ← 0;
   /* Successive calls to the planner ($\Phi$),
   modifying the operators before each call */
  WHILE (($p == \Phi(\Lambda, D, I, G)$) != NO_PLAN)
     && ($c < T$) DO {
    InsertInList(p, planList);
    $\Lambda$ ← RecordPlan($\Lambda$, p);
    c++}

  linearPlans ← {};/* No linear Plans yet */
  /* Linearize all partial order plans */
  FORALL e ∈ planList DO {
    L ← Linearize(e);
    InsertInList(L, linearPlans)}

  testCases ← linearPlans;
  /* decomposing the testCases */
  FORALL tc ∈ testCases DO {
    FORALL C ∈ Steps(tc) DO {
      IF (C == systemInteractionOperator) THEN {
        newC ← lookup(Mappings, C);
        REPLACE C WITH newC IN tc}
      ELSEIF (C == componentOperator) THEN {
        $\Lambda C$ ← OperatorSet(C);
        GC ← Goal(C);
        IC ← Initial(C);
        DC ← ObjectSet(C);
        /* Generate the lower level test cases */
        newC ← APPEND(lookup(Mappings, C),
          GenTestCases($\Lambda C, DC, IC, GC$, T));

        FORALL nc ∈ newC DO {
          copyOftc ← tc;
          REPLACE C WITH nc IN copyOftc;
          APPEND copyOftc TO testCases}}}}
  RETURN(testCases)}
```

Figure 5.7: The Complete Algorithm for Generating Test Cases

mouse and keyboard events are part of the GUI, three operators for mouse and keyboard events were defined in addition to the primitive and high-level operators. After analysis

Plan No.	**Plan Step**	**Plan Action**
1	1	FILE-OPEN(“private”, “Document.doc”)
	2	DELETE-TEXT(“that”)
	2	DELETE-TEXT(“must”)
	2	DELETE-TEXT(“be”)
	2	DELETE-TEXT(“modified”)
	2	TYPE-IN-TEXT(“final”, Times, Italics, 12pt)
	3	FILE-SAVEAS(“public”, “new.doc”)
2	1	FILE-OPEN(“public”, “doc2.doc”)
	2	TYPE-IN-TEXT(“is”, Times, Italics, 12pt)
	2	TYPE-IN-TEXT(“the”, Times, Italics, 12pt)
	2	DELETE-TEXT(“needs”)
	2	DELETE-TEXT(“to”)
	2	DELETE-TEXT(“be”)
	2	DELETE-TEXT(“modified”)
	2	TYPE-IN-TEXT(“final”, Times, Italics, 12pt)
	2	TYPE-IN-TEXT(“text”, Times, Italics, 12pt)
	3	FILE-SAVEAS(“public”, “new.doc”)
3	1	FILE-OPEN(“public”, “doc2.doc”)
	2	TYPE-IN-TEXT(“is”, Times, Italics, 12pt)
	2	TYPE-IN-TEXT(“the”, Times, Italics, 12pt)
	2	DELETE-TEXT(“to”)
	2	DELETE-TEXT(“be”)
	2	DELETE-TEXT(“modified”)
	2	TYPE-IN-TEXT(“final”, Times, Italics, 12pt)
	2	TYPE-IN-TEXT(“text”, Times, Italics, 12pt)
	2	SELECT-TEXT(“needs”)
	3	EDIT-CUT(“needs”)
	4	FILE-SAVEAS(“public”, “new.doc”)
4	1	FILE-NEW(“public”, “new.doc”)
	2	TYPE-IN-TEXT(“This”, Times, Italics, 12pt)
	2	TYPE-IN-TEXT(“is”, Times, Italics, 12pt)
	2	TYPE-IN-TEXT(“the”, Times, Italics, 12pt)
	2	TYPE-IN-TEXT(“final”, Times, Italics, 12pt)
	2	TYPE-IN-TEXT(“text”, Times, Italics, 12pt)
	3	FILE-SAVEAS(“public”, “new.doc”)

Table 5.2: Some WordPad Plans Generated for the Task of Figure 5.2.

of the hierarchical structure of WordPad, 36 system-interaction and component operators were obtained, i.e., roughly a ratio of 10 : 1. This reduction in the number of operators is impressive and helps speed up the plan generation process, as will be shown in Section 5.4.2.

Task No.	Plan Time (sec)	Sub Plan Time	Total Time (sec)
1	0.40	0.04	0.44
2	3.16	0.00	3.16
3	3.17	0.00	3.17
4	3.20	0.01	3.21
5	3.38	0.01	3.39
6	3.44	0.02	3.46
7	4.09	0.04	4.13
8	8.88	0.02	8.90
9	40.47	0.04	40.51

Table 5.3: Time Taken to Generate Test Cases for WordPad.

Defining preconditions and effects for the 36 operators was fairly straightforward. The average operator definition required 5 preconditions and effects, with the most complex operator requiring 10 preconditions and effects. Although operator definition is currently done by the test designer, this task may be simplified by maintaining definitions of commonly used operators in libraries, allowing operator reuse. It is anticipated that the primitive operators will be widely reusable, whereas the GUI dependent system-interaction operators may not be reusable because they are based on the structure of a specific GUI. However, component operators that are associated with a GUI component may be reused to test GUIs that employ the component. Another technique to obtain these operators is to automatically generate the preconditions and effects of the operators from formal GUI specifications.

Table 5.3 presents the CPU time taken to generate test cases for WordPad. Each row in the table represents a different planning task. The first column shows the task number; the second column shows the time needed to generate the highest-level plan; the third column shows the average time spent to decompose *all* sub-plans; the fourth column shows the *total* time needed to generate the test case (i.e., the sum of the two previous columns). These results demonstrate that the maximum time is spent in generating the high-level plan (column 2). This high-level plan is then used to generate a family of test cases by substituting alternative low-level sub-plans. These sub-plans are generated relatively faster (average shown in column 3), amortizing the cost of plan generation over multiple test cases. Plan 9, which took the longest time to generate, was linearized to obtain 2 high-level plans, each of which was decomposed to give several low-level test cases, the shortest of which consisted of 25 GUI events.

An automated test execution system was implemented, so that all the test cases could be automatically executed without human intervention. Automatically executing the test cases involved generating the physical mouse/keyboard events. Since the test cases are represented at a high level of abstraction, the high-level events were translated into physical events. The actual screen coordinates of the buttons, menus, etc. were derived from the layout information.

5.4.2 Hierarchical vs. Single-level Test Case Generation

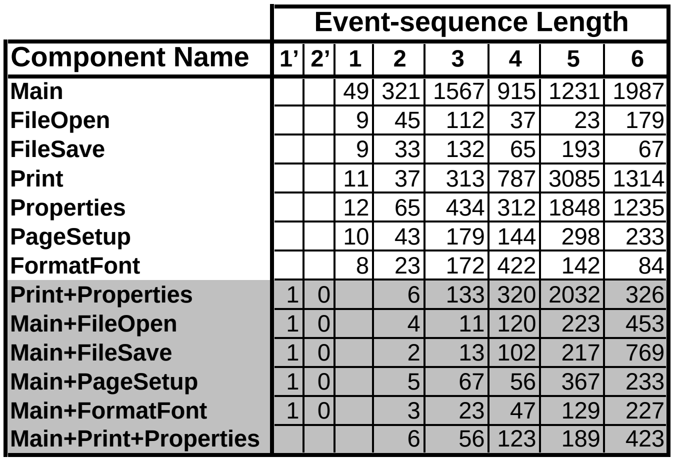
In the second experiment, the single-level test case generation was compared to the hierarchical test case generation technique. Recall that in the single-level test case generation technique, planning is done at a single level of abstraction, without using any component hierarchy. The primitive operators are used, which have a one-to-one correspondence with the GUI events. On the other hand, in the hierarchical test case generation approach, the hierarchical model of the GUI is used.

Results of this experiment are summarized in Table 5.4. The table shows CPU times for 6 different tasks. Column 1 shows the task number; Column 2 shows the length of the test case generated by using the single-level approach and Column 3 gives its corresponding CPU time ('-' indicates that no plan was found in 1 hour.). The same task was then used to generate another test case but this time using the system-interaction and component operators. Column 4 shows the length of the high-level plans and Column 5 displays the time needed to generate this high-level plan and then decompose it. The timing results show the hierarchical approach is more efficient than the single-level approach. For example, plan 1 obtained from the hierarchical algorithm expands to give a plan of length 18, i.e., exactly the same plan obtained by running its corresponding single-level algorithm. The efficiency results from the smaller number of operators used in the planning problem.

This experiment demonstrates the importance of the hierarchical modeling process. The key to efficient test case generation is to have a small number of planning operators at each level of planning. As GUIs become more complex, the modeling algorithm is able to obtain increasing number of levels of abstraction. Exploratory analysis for the much larger GUI of Microsoft Word was also performed. The automatic modeling process reduced the number of operators by a ratio of 20 : 1. The results of this analysis show that even though Microsoft Word has a larger GUI, it can be decomposed to obtain a small number of operators at each level of planning, a key to efficient test case generation.

	Single level		Hierarchical	
Task No.	**Plan Length**	**Time (sec.)**	**Plan Length**	**Time (sec.)**
1	18	8.93	3	0.11
2	20	47.62	4	0.18
3	24	189.87	5	0.14
4	26	3312.72	6	7.18
5	-	-	3	0.1
6	-	-	4	13.01

Table 5.4: Comparing the single level with the hierarchical approach. '-' indicates that no plan was found in 1 hour.



	Event-sequence Length							
Component Name	**1'**	**2'**	**1**	**2**	**3**	**4**	**5**	**6**
Main			49	321	1567	915	1231	1987
FileOpen			9	45	112	37	23	179
FileSave			9	33	132	65	193	67
Print			11	37	313	787	3085	1314
Properties			12	65	434	312	1848	1235
PageSetup			10	43	179	144	298	233
FormatFont			8	23	172	422	142	84
Print+Properties	1	0		6	133	320	2032	326
Main+FileOpen	1	0		4	11	120	223	453
Main+FileSave	1	0		2	13	102	217	769
Main+PageSetup	1	0		5	67	56	367	233
Main+FormatFont	1	0		3	23	47	129	227
Main+Print+Properties				6	56	123	189	423

Table 5.5: The Number of Event-sequences for Selected Components of WordPad Covered by the Test Cases.

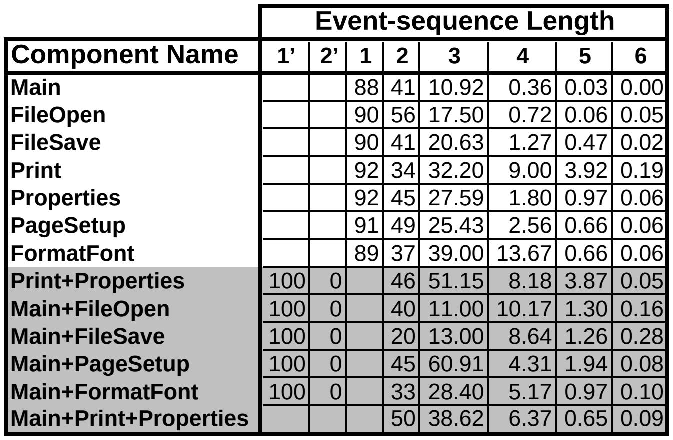
5.4.3 Evaluating the Coverage of a Test Suite

The third experiment was performed to determine the time taken to evaluate the coverage of a given test suite and how the resulting coverage report could guide further testing. The following steps were performed:

Identifying Tasks: 72 different tasks were carefully identified, making sure that each task exercised at least one unique feature of WordPad. For example, one task modified the font of text, and another printed the document on A4 size paper.

Generating Test Cases: Test cases were generated to achieve these 72 tasks. In all, 500 test cases were generated (multiple test cases for each task).

Coverage Evaluation: After the 500 test cases were available, the coverage evaluation algorithms of Figures 4.2 and 4.3 were executed. The coverage evaluation algorithms



	Event-sequence Length							
Component Name	**1'**	**2'**	**1**	**2**	**3**	**4**	**5**	**6**
Main			88	41	10.92	0.36	0.03	0.00
FileOpen			90	56	17.50	0.72	0.06	0.05
FileSave			90	41	20.63	1.27	0.47	0.02
Print			92	34	32.20	9.00	3.92	0.19
Properties			92	45	27.59	1.80	0.97	0.06
PageSetup			91	49	25.43	2.56	0.66	0.06
FormatFont			89	37	39.00	13.67	0.66	0.06
Print+Properties	100	0		46	51.15	8.18	3.87	0.05
Main+FileOpen	100	0		40	11.00	10.17	1.30	0.16
Main+FileSave	100	0		20	13.00	8.64	1.26	0.28
Main+PageSetup	100	0		45	60.91	4.31	1.94	0.08
Main+FormatFont	100	0		33	28.40	5.17	0.97	0.10
Main+Print+Properties				50	38.62	6.37	0.65	0.09

Table 5.6: The Percentage of Total Event-sequences for Selected Components of WordPad Covered by the Test Cases.

were implemented using Perl and *Mathematica* [93] and were executed on a Sun Ultra SPARC workstation (SPARC Ultra 4) running Sun OS 5.5.1. Even with the inefficiencies inherent in the Perl and Mathematica implementation, the algorithms could process the 500 test cases in 47 minutes (clock time). The results of applying the algorithms are summarized as coverage reports in Tables 5.5 and 5.6. Table 5.5 shows the actual number of event-sequences that the test cases covered. Table 5.6 presents the same data, but as a percentage of the total number of event sequences. Column 1 in Table 5.6 shows close to 90% coverage for single events. The remaining 10% of the events (such as `Cancel`) were never used by the planner since they did not contribute to a goal. Column 2 shows that the test cases achieved 40-55% event-interaction coverage. Note that since all the components were invoked at least once, 100% invocation coverage (column 1') was obtained. However, none of the components were terminated immediately after being invoked. Hence, no invocation-termination coverage (column 2') was obtained.

This result shows that the coverage of a large test suite can be evaluated in a reasonable amount of time. Columns 4, 5, and 6 of Table 5.6 show that only a small percentage of length 4, 5, and 6 event sequences were tested. The test designer can evaluate the importance of testing these longer sequences and perform additional testing. Also, the two-dimensional structure of Table 5.6 helps target specific components and component-interactions. For example, 60% of length 2 interactions among `Main` and `PageSetup` have been tested whereas only 11% of the interactions among `Main` and `FileOpen` have been tested. Depending on the relative importance of these components and their interactions, the test designer can focus on testing these specific parts of the GUI.

The coverage report produced from this experiment shows two important weaknesses of PATHS. First, PATHS did not use events such as `Cancel` since they did not contribute to the planning goal, resulting in loss of coverage as seen in column 1 of Table 5.6. Second, PATHS did not generate event sequences that invoke a component and terminate it immediately since such preemptive termination did not contribute to the final goal. This behavior of the planning-based test-case generator resulted in loss of coverage as seen in column 2' of Table 5.6. Note that, in practice, GUI users can, and do terminate components without interacting with other events in the component. It is important to test the GUI for such event sequences, perhaps by employing other testing techniques. The important lesson learned from this experiment is that it is necessary to combine several techniques to test a software, so that weaknesses of one technique do not have too much impact on the overall testing results. Rather, the combined strengths of several testing techniques will result in better testing of the software.

5.5 Conclusions

This chapter presented the design of the test case generator, an essential component of the GUI testing framework. The test case generator employs tasks, consisting of initial and goal states, to generate test cases. The key idea of using tasks to guide test case generation is that the test designer is likely to have a good idea of the possible goals of a GUI user, and it is simpler and more effective to specify these goals than to specify sequences of events that achieve them. This test case generation technique is unique in that it employs an automatic planning system to generate test cases from GUI events and their interactions.

Experiments have demonstrated that the planning technique is both practical and useful by generating test cases for the WordPad software's GUI. The experiments showed that the planning approach was successful in generating test cases for different scenarios. The GUI representation was used extensively during the test case generation process. Experiments showed that the hierarchical component model of the GUI was necessary to efficiently generate test cases. Representing the test cases at a high level of abstraction makes it possible to fine-tune the test cases to each implementation platform, making the test suite more portable. A mapping is used to translate the low-level test cases to sequences of physical actions. Such platform-dependent mappings can be maintained in libraries to customize the generated test cases to low-level, platform-specific test cases.

Chapter 6
Test Oracles

Once test cases have been generated by the test case generator, they are executed on the GUI by the test executor. The question now is to *automatically* determine whether a GUI behaves correctly when a test case is executed on it. This question is answered by using a *test oracle.*

The characteristics of GUIs present special challenges when designing a test oracle. These challenges stem from the fact that GUIs are event-based systems. The GUI test case consists of an event sequence, where the effect of each event may depend upon the effects of its previous events. There is no specific output: rather, each event affects the state of the GUI. Moreover, comparison of the expected and actual GUI states cannot wait until the entire event sequence has been executed. Instead, it is necessary to verify the state of the GUI after the execution of each event; otherwise, incorrect GUI behavior for one event may result in a state in which future events in the sequence cannot be executed at all.

The above challenges suggest the need to develop an automated oracle that answers the question of whether a GUI executing under a test case behaves as expected. The automation should occur both in the derivation of the expected state and the comparison of the expected and actual states. Developing an automated test oracle for GUIs has a number of requirements. First, the GUI representation should be used to model the GUI's intended behavior so that its *expected state* can be automatically derived for each test case. Second, the *actual state* of the executing GUI needs to be captured and represented in a form that is suitable for comparison with the expected state. Finally, a mechanism to automatically compare the expected state with the actual state of the executing GUI needs to be developed.

This chapter presents techniques for an automated GUI test oracle. An overview of the oracle is shown in Figure 6.1. An expected-state generator uses the GUI representation presented in Chapter 3 to automatically derive the GUI's expected state for each test case. The oracle obtains the GUI's actual state from an *execution monitor.* A *verifier* in the

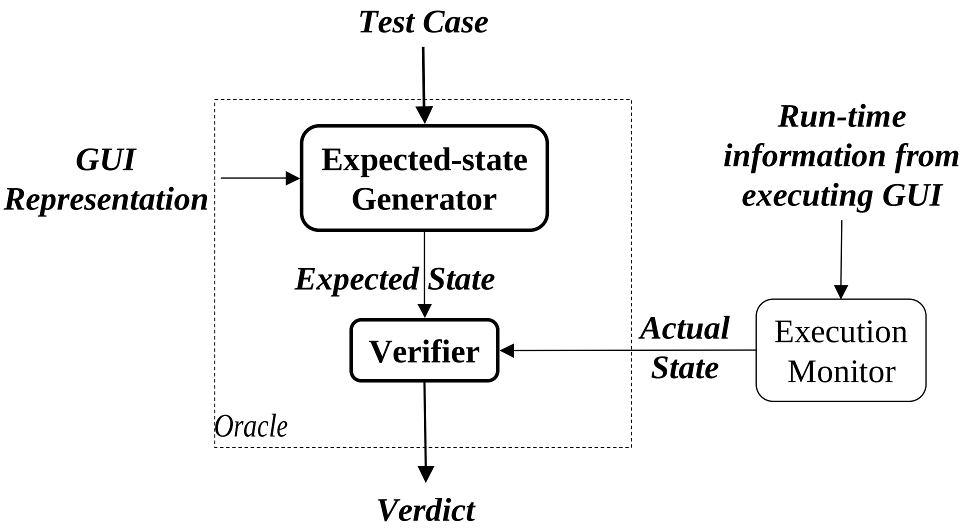


Figure 6.1: An Overview of the GUI Oracle.

oracle then automatically compares the two states and determines if the GUI is executing as expected. The oracle was implemented as part of the GUI testing framework. Experiments evaluated the oracle on WordPad and provide timing results that establish the feasibility of this approach.

The remainder of this chapter presents the components of the test oracle and their functionality. Section 6.1 presents the design of the expected-state generator. Section 6.2 describes techniques to design the execution monitor. Details of the verifier is discussed in Section 6.3. The algorithm for the complete oracle is described in Section 6.4. Finally, experimental results are presented in Section 6.5.

6.1 Expected State Generator

The expected-state generator uses the GUI representation to determine the expected state of a GUI after the complete or partial execution of any test case. Recall that events are modeled as state transducers. For any test case $< S_0, e_1; e_2; \ldots; e_n, S_1; S_2; \ldots S_n >$, the legal sequence of states $S_1; S_2; \ldots S_n$ such that $S_i = e_i(S_{i-1})$ for $i = 1, \ldots, n$ represent the expected state of the GUI after each event is executed, starting in S_0. The question is how, in practice, to compute these expected states.

The next state is obtained from the current state S_c and the event e's operator's effects, represented by $\mathit{Eff}(e)$ (see Section 3.3), as follows:

1. Delete all literals in S_c that unify with a negated literal in $\mathit{Eff}(e)$, and

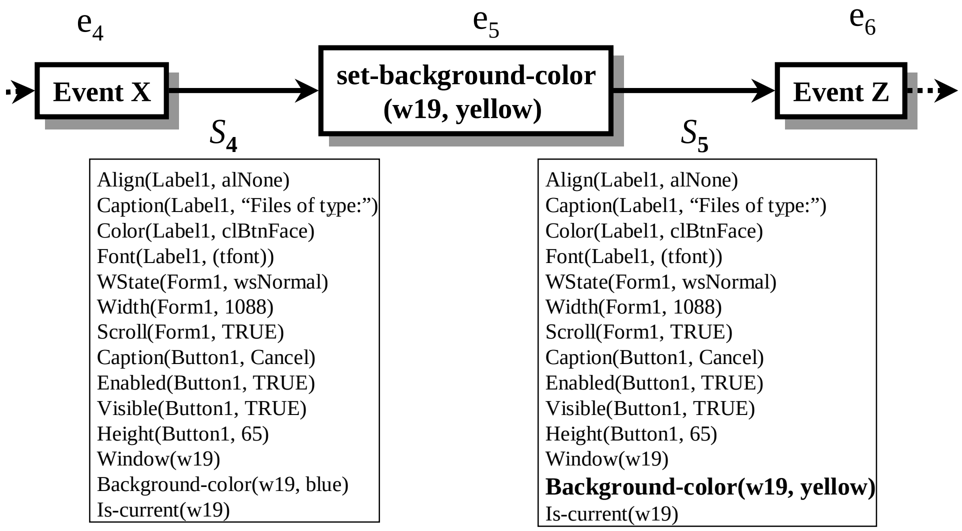


Figure 6.2: A Few Test-Case Events with Expected State Information.

2. add all positive literals in $\mathit{Eff}(e)$.

Thus, using the GUI representation, the expected state can be derived from the initial state and the sequence of events in the test case. The expected state S_1 is derived from S_0 by using the effects of e_1's operator, i.e., $S_1 = e_1(S_0)$. The process is repeated until the entire expected state sequence has been derived. For example, consider the expected state shown in terms of properties for events e_4 and e_5 in Figure 6.2. The expected state of the GUI after e_4 is performed is represented as S_4. The GUI's state changes after event e_5 (`set-background-color(w19, yellow)`) is executed. The new state obtained is S_5. The changes are highlighted using bold font. As mentioned earlier in the description of the `set-background-color` operator (Section 3.3), the `background-color` of the window changes.

The test case and expected state sequence shown in Figure 6.2 have all the necessary components to carry out a successful test run and can be used for manual testing. One manually executes a test case, and after each step, manually compares the appearance of the GUI with the expected state at that time. Manual verification has at least two problems: (1) it is labor intensive, and (2) often the GUI state includes “hidden” properties that are not visually accessible. Hence, test execution and the oracle have been fully automated by implementing the execution monitor and the verifier, which are described next.

6.2 Execution Monitor

The *execution monitor* is a process that, given an executing GUI, returns the current values of all the properties in the complete set for the GUI. There are several different approaches that can be used to automate the process of extracting actual GUI state information in a form that is suitable for comparison with the expected state description. Two possible approaches are as follows:

1. *Screen scraping*

   Screen scraping is a technique used to selectively remove information from an application's screen/terminal interface for reuse. Typically, the information is accessed by using low-level, terminal-specific system calls. The bitmaps/text obtained are analyzed to determine the correctness of the executing GUI. Although useful for determining exactly what is visible to the user, non-visible properties cannot be verified using screen scraping.

2. *Querying*

   Querying the GUI's software is a technique to determine the values of all the properties present in the GUI, including non-visible and visible properties. Although the results of the querying technique are more complete than screen scraping, querying requires access to the GUI's code, possibly modifying the code to access the values of properties.

In a typical testing scenario, both the above techniques may be used to obtain the values of properties. Once the actual values of properties for an element or elements are known, the verifier can compare them against the expected values, to determine if they are equal. The details of the verifier are presented next.

6.3 Verifier

The verifier is a process that compares the expected state of the GUI with the actual state and returns a *verdict* of equal or not equal. The question, then, is what properties should be compared during the verification process. Several possible approaches can be used to select the properties to be compared. The differences among these approaches establishes the *level of testing* performed:

Changed-Properties Verification: Here, comparison is made only for those properties that were expected to change as a result of the immediately preceding event. That is, if event e was just executed, only the properties that are included in $\mathit{Eff}(e)$ are compared against their expected values. Although efficient, this level of testing will fail to detect changes to properties that change when they are not expected to change. For example, if the background color of a window changes, but it was not expected to change, the error would go unnoticed.

Relevant-Properties Verification: Here, all the properties in the reduced property set are checked. Recall from Section 3.2 that the reduced property set includes all the properties that the current GUI can have. This is a more extensive level of testing than changed-properties verification, but it may still fail when some GUI property P changed in the executing GUI, but P was not a part of the GUI specification. For example, consider a GUI for a plain-text editor, e.g., MS NotePad in which users cannot change the text color. If some event in the test case has the unintended effect of changing the text color, then this error would go unnoticed, since the color information was not encoded in the expected state.

Complete-Properties Verification: Here, a check is made for all the properties that a language or toolkit provides for a GUI. Recall that the verifier has access to the complete set of properties. The only problem is the absence of an expected state to compare against all these additional properties. The currently available expected state encodes only the reduced property set. To address this problem, before the test case is executed, a baseline *complete expected state* of the GUI is created. During test-case execution, the comparisons are done between the GUI's actual state and the updated complete expected state.

In practice, the test designer can choose a combination of the above levels of testing. For example, the verifier can perform changed-properties verification after each test event and complete-properties verification after every 10 events.

At each step in the test case, the verifier uses the values of all these properties to check them for correctness. Thus, in the example in Figure 6.2, the expected state shown in S_4 and S_5 will be automatically compared with the actual GUI state when the test case is executed. In case the properties in the actual state do not match with those in the expected state, an error is reported to the test designer. In addition, the mismatched property, the complete set of expected properties, and the actual properties are also returned to the test designer to help pinpoint the source of the error during debugging. An error detected during testing may be due to a problem in the (1) implementation or (2) operator definition. If the test designer determines that the error occurred because of an incorrect operator definition, then the operator is debugged and fixed. Testing is then resumed. If, however, the implementation is found to be faulty, then the problem is reported to the GUI development team.

6.4 GUI Testing Algorithm

In this section, an algorithm is presented that shows how the components of the test oracle are used when testing the GUI. It also shows the details of how the expected state is derived from the current state.

Figure 6.3 gives a high-level view of the main testing algorithm (`TestGUI`) and a procedure `ExpStateGen`, invoked by `TestGUI`. The algorithm `TestGUI` executes a test case automatically on the GUI, examining its actual state and comparing it with the expected state. The algorithm takes three parameters: (1) the `levelOfTesting`, which determines what properties are to be compared by the verifier, (2) the test case `T` to be executed on the GUI (`T` contains the expected initial state and a sequence of events), and (3) the operators (`GUI_Operators`) representing the abstract model of the GUI. Note that each event in the test case has a corresponding definition in `GUI_Operators`. The algorithm returns a verdict, depending on the outcome of the test case execution. For each event in the test case, `TestGUI` calls the procedure `ExpStateGen` (`line 9`) to determine the expected state of the GUI. If `ExpStateGen` is successful, then the event in the test case is automatically executed (`line 12`) on the GUI and its actual state is determined by invoking the execution monitor `ExecMonitor` (`line 13`). Both the expected and actual state are compared by the verifier (`line 15`) that performs comparisons based on the current level of testing. `TestGUI` returns the verdict (`line 30`), i.e., the outcome of the execution of the test case.

The procedure `ExpStateGen` takes three inputs: (1) the current state of the GUI (`currentState`), (2) the `event` to be executed on the GUI, and (3) the GUI operators (`operators`). Every event in the test case has a corresponding operator definition (`line 35`). The event contains the actual parameters of the operator definition, which are substituted for the formal parameters (`line 36`). `ExpStateGen` performs an extra check to determine if the preconditions of the operator are satisfied in the current state (`lines 37..39`). If they are not satisfied, then there is an error in the test case, and this result is propagated to the calling procedure. If the preconditions are satisfied, the new state is computed by applying the effects of the operator. If the effects contain a negated property, then it is deleted from the new state (`lines 42..43`) and if it contains a positive property, it is inserted (`lines 44..45`) in the new state. The result `newState` is returned to the calling algorithm.

```
ALGORITHM: TestGUI(
 levelOfTesting, /* changed, relevant, or complete property
          verification */
 T, /* test case $S_0, e_1; e_2; e_3; \ldots; e_n$ */
 GUI_operators /* $\{Op_1, Op_2, Op_3, ..., Op_n\}$. Each $Op_i$ =
       <Name, Preconditions, Effects>*/ ) {
   State ⟵ $S_0$;
   FOREACH event e ∈ $< e_1; e_2; e_3; \ldots; e_n >$ {
     expState ⟵ ExpStateGen(State, e, GUI_operators);
     IF (expState == TEST_CASE_INVALID)
        BREAK;
     ExecuteEvent(e, GUI); /* Automatically execute event on GUI */
     actualState ⟵ ExecMonitor(GUI);
     /* check actual State and expected for this LEVEL_OF_TESTING. */
     IF (Verifier(expState, actualState,
          levelOfTesting) == FALSE)
       BREAK;
     State ⟵ expState;}
   IF (TEST_CASE_INVALID) {
     error("Invalid Test Case");
     debugInfo("Actual GUI State = ", actualState);
     debugInfo("Expected GUI State = ", expState);
     Verdict ⟵ INVALID;}
   IF (FALSE) { /* if verifier reported FALSE, then GUI is incorrect*/
     report("GUI failed the test case");
     debugInfo("Actual GUI State = ", actualState);
     debugInfo("Expected GUI State = ", expState);
     Verdict ⟵ INCORRECT;}
   ELSE Verdict ⟵ CORRECT;
   RETURN(Verdict);}
PROCEDURE: ExpStateGen(
 currentState, /* properties, $\{p_1, p_2, p_3, ..., p_n\}$ - the State of the GUI*/
 event, /* step of the test case – eventName(parameters)*/
 operators /* $\{Op_1, Op_2, Op_3, ..., Op_n\}$. */) {
   opDef ⟵ Lookup(event, operators); /* get operator for event */
   op ⟵ Bind(opDef, event); /* bind all variables in op def. */
   p ⟵ preconditions(op); /* extract the preconditions of the operator */
   IF(Satisfied(p, currentState) == FAILED)
     RETURN(TEST_CASE_INVALID);
   eff ⟵ effects(op); /*extract the effects of the operator*/
   newState ⟵ currentState;
   FOREACH (f ∈ eff) {/*delete all properties that are negated in effects*/
     IF (negated(f)) delete f from newState;
   FOREACH (f ∈ eff) {/*insert all properties that are positive in effects*/
     IF (positive(f)) insert f in newState;
   RETURN(newState);}
```

Figure 6.3: The GUI Testing Algorithm.

6.5 Experiments

To explore the practicality of this approach, the performance of the oracle was evaluated on the example WordPad GUI. More specifically, the goals of the experiment

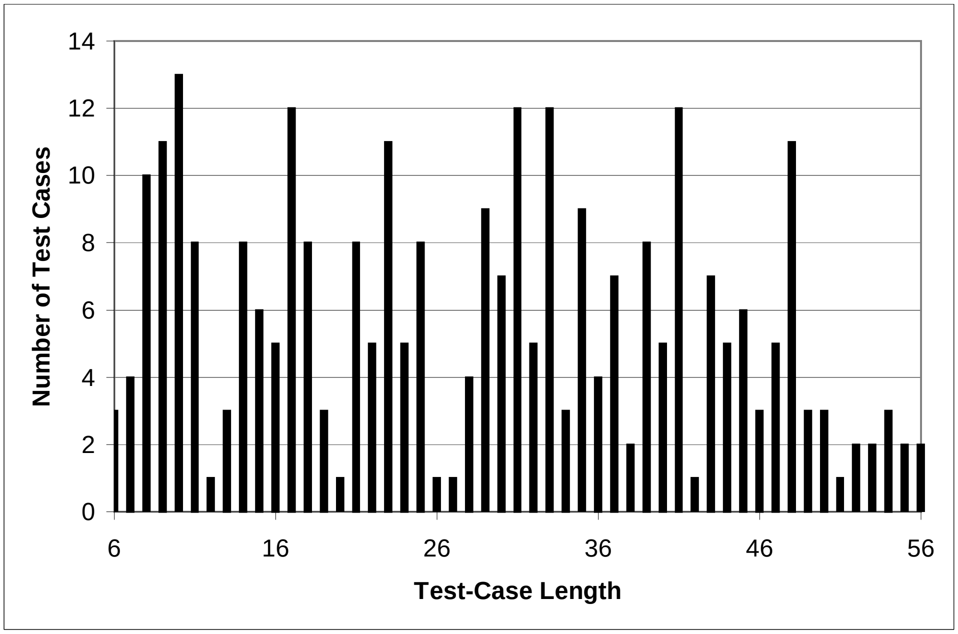


Figure 6.4: Number of Test Cases Generated and their Lengths.

were to determine (1) the execution time to derive the expected state information, and (2) the time to execute the verifier and the execution monitor. In both cases, the times were compared with test case generation and execution time to determine the extra time needed to derive the expected state and execute the verifier and the execution monitor.

These experiments were designed to help determine the scalability of the expected-state generator and test-oracle executor. In all, 290 test cases of lengths varying from 6 to 56 events were generated. Figure 6.4 shows the number of test cases generated for each length.

For the first experiment, the expected-state generator was implemented in C and executed on a Pentium-based computer (350MHz, 256MB RAM) running Linux. The expected-state generator produced the expected states of all the test cases off-line, during test case generation. As each test case was generated, the expected state generator used the operators to produce the corresponding expected state.

The results of this experiment are summarized in Figure 6.5. The x-axis shows the test case length, and the y-axis shows the average time (in seconds) to generate a test case. Note that the time shown is the average of multiple test cases. As the graph shows, the significant portion of the time was spent in generating the test cases. The expected state was derived much faster. Note that the total time needed to generate the test cases

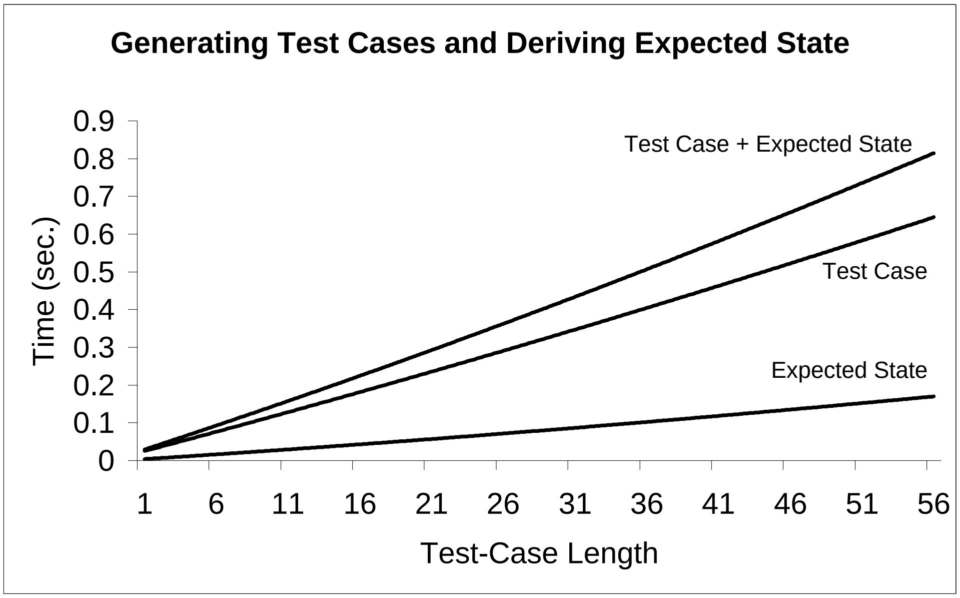


Figure 6.5: Time needed to Generate the Test Cases and Expected-State Information.

and expected state was very small. In fact, all of the 290 test cases and their corresponding expected states were generated in a total of 75.84 sec. CPU time.

For the second experiment, to determine the time to execute the verifier and the execution monitor, the execution monitor and verifier were implemented in Borland's C++ Builder, running under Windows NT. The execution monitor maintained a list of all the properties of the executing GUI and extracted the values after each event. Some properties were visible, e.g., open menus, that could be retrieved directly from the screen by using screen scraping whereas other properties required getting values from the executing GUI by using queries, implemented through a socket connection.

Implementing the verifier was straightforward. The *relevant properties verification* approach was performed. Note that this more expensive level of testing was deliberately chosen to determine the worst-case time for oracle execution. During comparison, the expected and actual states were compared for equivalence.

As seen in Figure 6.6, the total time needed to execute the verifier and the execution monitor was very small. All 290 test cases required less than a total of 10 minutes clock time to execute without any intervention.

These experiments demonstrate that the GUI representation can be used to develop an oracle that is both efficient and useful for GUI testing.

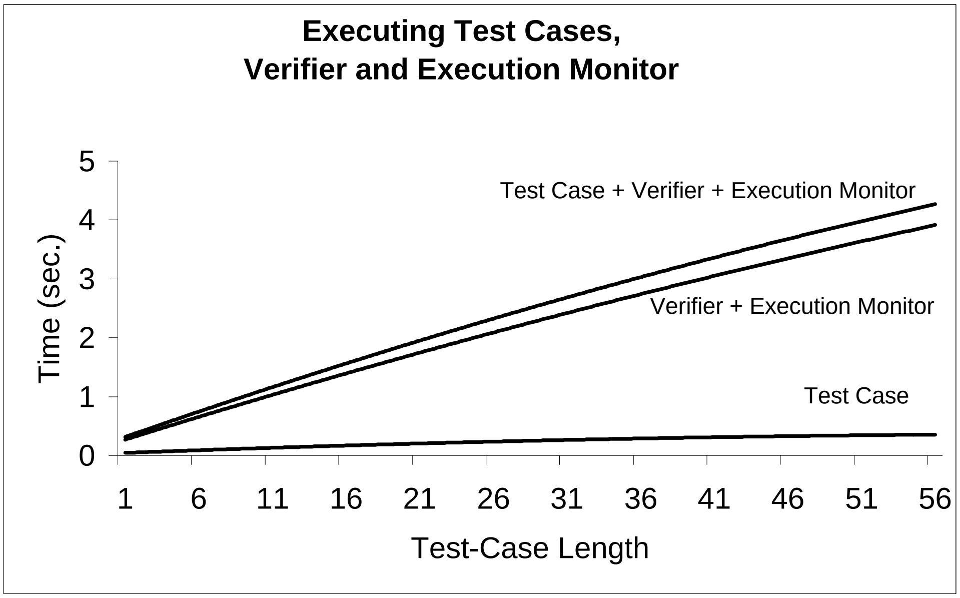


Figure 6.6: Time needed to Execute the Test Cases and Verifier.

6.6 Conclusions

This chapter presented the design of the automated GUI test oracle. The test oracle automatically derives the expected state sequences and compares the actual and expected states after each event in the test case. The oracle generates the expected state from the GUI representation. The oracle obtains the actual state from an execution monitor. The actual state is represented as a set of objects and properties. The oracle then compares the two states and determines if the GUI is performing as expected.

Two experiments have demonstrated that the oracle is both practical and useful by deriving expected state sequences for the example WordPad software's GUI and using them to test the software's GUI. The experiments have also demonstrated that a large number of test cases can be executed and the GUI's execution behavior verified automatically in very little time.

Chapter 7
Regression Tester

The regression tester is the only component of the GUI testing framework that is not used during first-time testing of a GUI; it is invoked by the test designer to retest a modified GUI. Instead of re-testing the modified GUI in its entirety, the regression tester reuses results from previous test runs to conserve resources and speed up the re-testing process while still maintaining the same quality of testing. The goal of regression testing is to help ensure the correctness of the new/modified parts of the GUI as well as to establish confidence that the modifications have not adversely affected previously tested parts.

Regression testing of conventional software typically involves performing three tasks. First, parts of the original software that may have been affected by the modifications are identified. Then, a subset of the original test cases is selected to retest these parts. Third, new test cases are generated to test affected parts of the software, not tested by the selected test cases. This model of regression testing of conventional software can be extended for regression testing of GUIs.

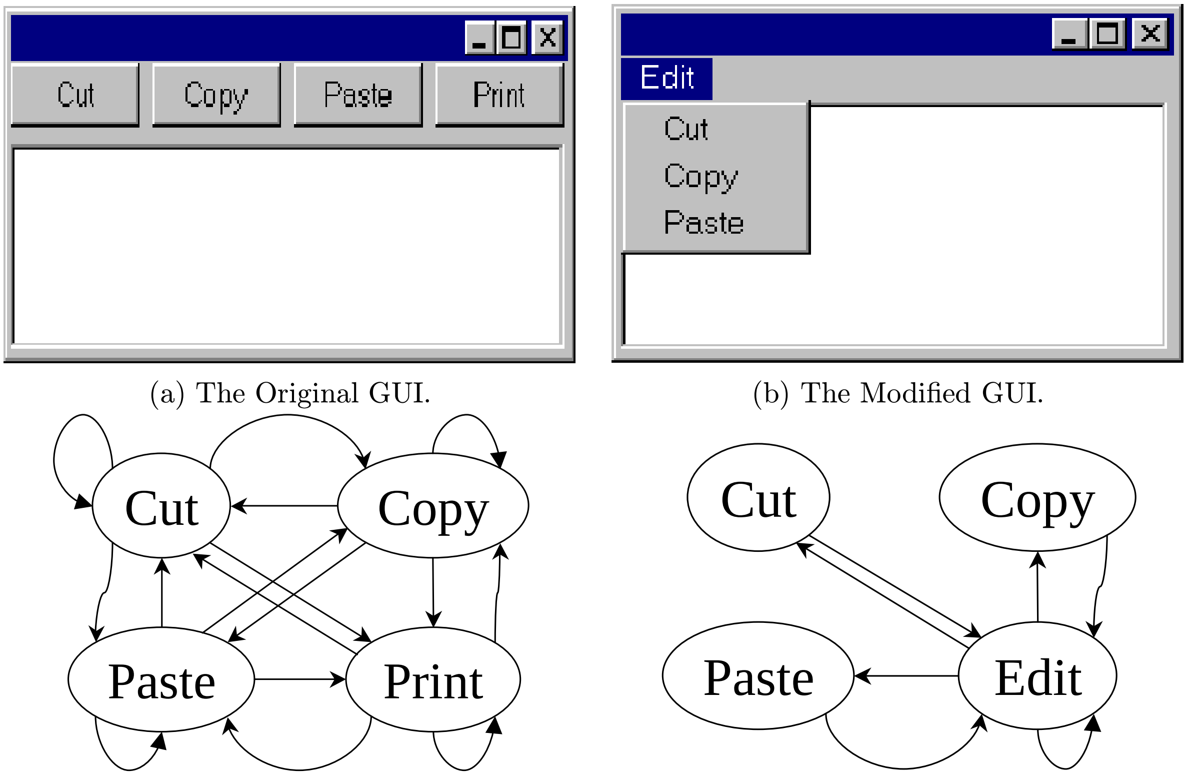
Recall (from Section 3.8) that a GUI test case consists of three parts – a reachable initial state S_0, a legal event sequence $e_1; e_2; \ldots; e_n$ for S_0, and expected states $S_1; S_2; \ldots; S_n$. A modification in the GUI may affect any of these parts of a test case. For example, a modification to the event-flow of the GUI may cause a test case's event sequence to become illegal. Another modification, such as a change to the background color of a window in a GUI in which no event can modify the background color, may make the initial state of a test case unreachable. Such test cases cannot be executed on the modified GUI. Table 7.1 shows all the possible ways in which modifications made to the GUI may affect the three parts of a test case. The columns show the effects of the modifications to the initial state, event sequence, expected state, and test case respectively. The first row shows the case where a test case was not affected by the GUI modifications, since its initial state is reachable, it has a legal event sequence and a corresponding correct expected state. Such a test case is called a *valid* test case. A valid test case need not be run on the modified GUI since

Initial State S_0	**Event Sequence** $e_1; e_2; \ldots; e_n$	**Expected State** $S_1; S_2; \ldots; S_n$	**Test Case** **Status**
reachable	legal	correct	valid
reachable	legal	incorrect	invalid
reachable	illegal	×	invalid
unreachable	×	×	invalid

Table 7.1: All Possible Effects of GUI Modifications on the Parts of a Test Case.

it will re-execute a sequence of unmodified events that have already been tested on the original GUI. The second row shows that a modification caused the expected state part of the test case to become incorrect, perhaps because the effects of one of the events in the test case's event sequence changed. Although the event sequence of this test case is legal and can be executed on the GUI, its corresponding expected state cannot be used to verify the correctness of the GUI. Executing such a test case is not useful since the tester cannot determine whether or not the GUI executed correctly. The third row shows that the GUI modification altered the event-flow of the GUI, causing the event sequence part of the test case to become illegal. The fourth row shows that the initial state of the test case became unreachable. Test cases of rows 3 and 4 cannot be executed on the GUI. Entries marked with "×" indicate "don't care" conditions, i.e., if the initial state of a test case is unreachable, it does not matter if the event sequence is legal and expected state is incorrect – the test case cannot be executed. As the table shows, test cases represented by rows 2, 3, and 4 are called *invalid* test cases. Note that a test case may become invalid because of a number of modifications made to the GUI. Although an invalid test case cannot be executed on the GUI, it contains valuable information about how the modifications have affected the execution behavior of the GUI and hence can be used to produce test cases that target these modifications.

The regression tester developed in this research is based on a new approach for regression testing that *repairs* some of the invalid test cases. The technique is targeted to GUI regression testing. Compared to new test cases generated from scratch, the repaired test cases are more likely to reveal faults introduced by modifications made to the GUI since they target sequences of events that were modified in the GUI. The next section presents a GUI regression testing example and shows test cases that may be repaired and executed on a modified GUI.



(a) The Original GUI.

(b) The Modified GUI.

(c) The Original GUI's Event-flow Graph.

(d) The Modified GUI's Event-flow Graph.

Figure 7.1: A Regression Testing Example.

7.1 A GUI Regression Testing Example

This section presents a GUI regression testing example by showing (1) an example of a GUI modification, (2) examples of test cases that have become invalid for the modified GUI, (3) an intuitive idea of how analysis of the GUI can help identify the invalid test cases, and (4) how invalid test cases may be repaired to obtain valid test cases.

Figure 7.1 presents a GUI, its modified version, and their corresponding event-flow graphs. The original GUI consists of 4 events, `Cut`, `Copy`, `Paste`, and `Print`, all directly accessible when the GUI is invoked. The modified GUI contains 3 of the 4 original events; `Print` has been deleted and the remaining 3 events have been grouped into a pull-down menu, which is opened by clicking on `Edit`. Figures 7.1(c) and (d) show the event-flow graphs of the original and modified GUIs respectively. The original GUI's event-flow graph is fully connected with 4 vertices representing the 4 events. The modified GUI's event-flow graph is quite different from that of the original GUI; it is no longer fully connected and

#	Event Sequence	Events Used	Edges Covered
1	Copy; Print; Cut	{Copy, Cut, Print}	{(Copy, Print), (Print, Cut)}
2	Cut	{Cut}	{}
3	Cut; Paste	{Cut, Paste}	{(Cut, Paste)}
4	Copy; Cut; Paste	{Cut, Copy, Paste}	{(Copy, Cut), (Cut, Paste)}

Table 7.2: Four Event Sequences for the Original GUI.

Edit must be performed before any other event can be performed. The following four sets of changes may be obtained, summarizing the differences between the two event-flow graphs:

1. **events_deleted** = {Print}.
2. **events_added** = {Edit}.
3. **efg_edges_deleted** = {(Cut, Cut), (Copy, Copy), (Paste, Paste), (Print, Print), (Cut, Copy), (Cut, Paste), (Cut, Print), (Copy, Cut), (Copy, Paste), (Copy, Print), (Print, Cut), (Print, Copy), (Print, Paste), (Paste, Cut), (Paste, Copy), (Paste, Print)}.
4. **efg_edges_added** = {(Edit, Edit), (Edit, Cut), (Edit, Copy), (Edit, Paste), (Cut, Edit), (Copy, Edit), (Paste, Edit)}.

Four event sequences used to test the original GUI are shown in Table 7.2. Column 1 shows the test case number, column 2 shows the event sequence of the test case, column 3 shows the events in the event-flow graph used by the test case, and column 4 shows the edges of the event-flow graph covered by the test case. The following observations can be made by examining these test cases and the 4 sets above:

1. Since Print was deleted from the GUI (**events_deleted**), event sequence 1 is invalid.
2. Since (Cut, Paste) and (Copy, Cut) have been deleted from the GUI (**efg_edges_deleted**), event sequences 3 and 4 have become invalid.
3. Event sequence 2 is still valid since Cut is available in the modified GUI (starting in an initial state in which Edit has been performed).

Intuitively, looking at the original and modified GUIs, event sequences 3 and 4 may be modified (or *repaired*) to obtain legal event sequences. Repairing event sequence 3 yields <Cut; Edit; Paste> and event sequence 4 yields <Copy; Edit; Cut; Edit; Paste>. These two repaired event sequences are legal and may be used to test the modified GUI. It is not obvious how event sequence 1 may be repaired since it contains an event, namely Print, that is no longer available in the modified GUI. In this example, this event sequence may be discarded and not used for regression testing. This example shows that some invalid test cases may not be repairable. After repairing, the test designer can choose from a total of three event sequences and use them for regression testing. Note that since event sequence 2 has already been executed on the original GUI, and none of the events in this event sequence have been modified, it need not be rerun. The remaining two event sequences, 3 and 4, can be used for regression testing. Since these event sequences were repaired from the original test suite, they are able to test whether modifications have adversely affected the previously tested parts of the GUI.

Note that a test case may become invalid because of several modifications made to the GUI. Consequently, such a test case may need to be repaired several times before it becomes valid. This example did not present details of modification of the initial state and expected states of the four test cases. As shown in Table 7.1, the initial state and the expected states also play important roles in determining the validity of the test case. For example, if the specifications of the `Cut` event were modified, then the expected state corresponding to event sequence 2 would become incorrect, making test case 2 also invalid. The expected state can also be repaired as will be described in Section 7.5. Note that new events and edges added to an event-flow graph cannot result in illegal event sequences. The event sequences from the original test suite neither use any of the new events nor do they cover any of the new edges.

The remainder of this chapter presents the design of the regression tester that repairs invalid test cases for regression testing. In performing regression testing, the regression tester partitions the original test suite into valid and invalid test cases. Of the invalid test cases, the repaired test cases form a part of the regression test suite whereas the non-repairable ones are discarded. The new GUI testing method is summarized in Figure 7.2. Note that new test cases, generated to test affected parts of the GUI not tested by the repaired test cases, are also a part of the regression test suite. The next section presents an overview of the design of the regression tester.

7.2 Overview of Regression Tester

The regression tester, based on the new repairing method, contains the following components.

- **Test case checker** partitions the original test suite into (1) valid test cases, (2) test cases that are invalid because they specify incorrect expected state for the modified GUI, (3) test cases that are invalid because they specify an illegal event sequence for

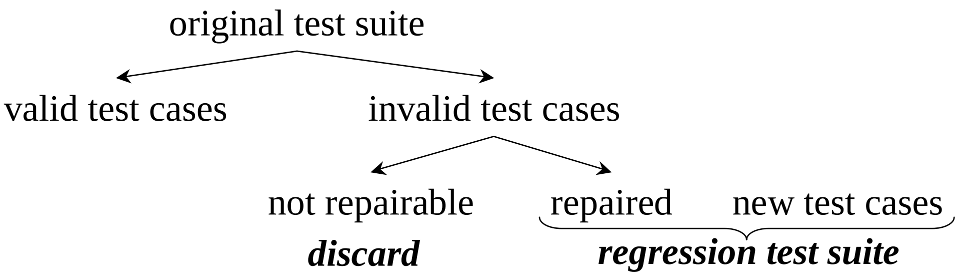


Figure 7.2: The New Regression Testing Method.

the modified GUI, and (4) test cases that contain an unreachable initial state and hence cannot be repaired.

- **Test case repairer** repairs the invalid test cases. The test case repairer consists of two parts – an **event-sequence repairer** that repairs illegal event sequences, and an **expected-state repairer** that repairs incorrect expected states.

Figure 7.3 shows the components of the regression tester and their interactions with other components of the GUI testing framework. The figure shows, in addition to the components discussed above, the **test case generator** that interacts with the **coverage evaluator** to generate new test cases to test the new parts of the GUI. Together, the repaired and new test cases form the regression test suite. The remainder of this chapter presents techniques to repair invalid test cases. The next section describes how GUI modifications are identified by analyzing the GUI's model. Section 7.4 and 7.5 present details and algorithms for the test case checker and repairer respectively. Finally, Section 7.6 presents results of an experiment performed to determine whether the test case repairing technique could be used to produce valid test cases and the time taken to make the repairs.

7.3 Analyzing GUI Modifications

The first step to performing automated regression testing is to identify the modifications made to the GUI and their effects. Since the GUI is composed of components, these modifications are classified as either *event-level* or *component-level*, and intra- and inter-component analyses are used respectively to identify them. The key idea is to compute the additions and deletions made to the event-flow graphs and integration tree of the original GUI to obtain the modified GUI. The assumption made here is that events and components have unique names. Moreover, they are not renamed across versions of the GUI unless they are modified. For example, if the event `File` is not modified, then it is called `File` in the

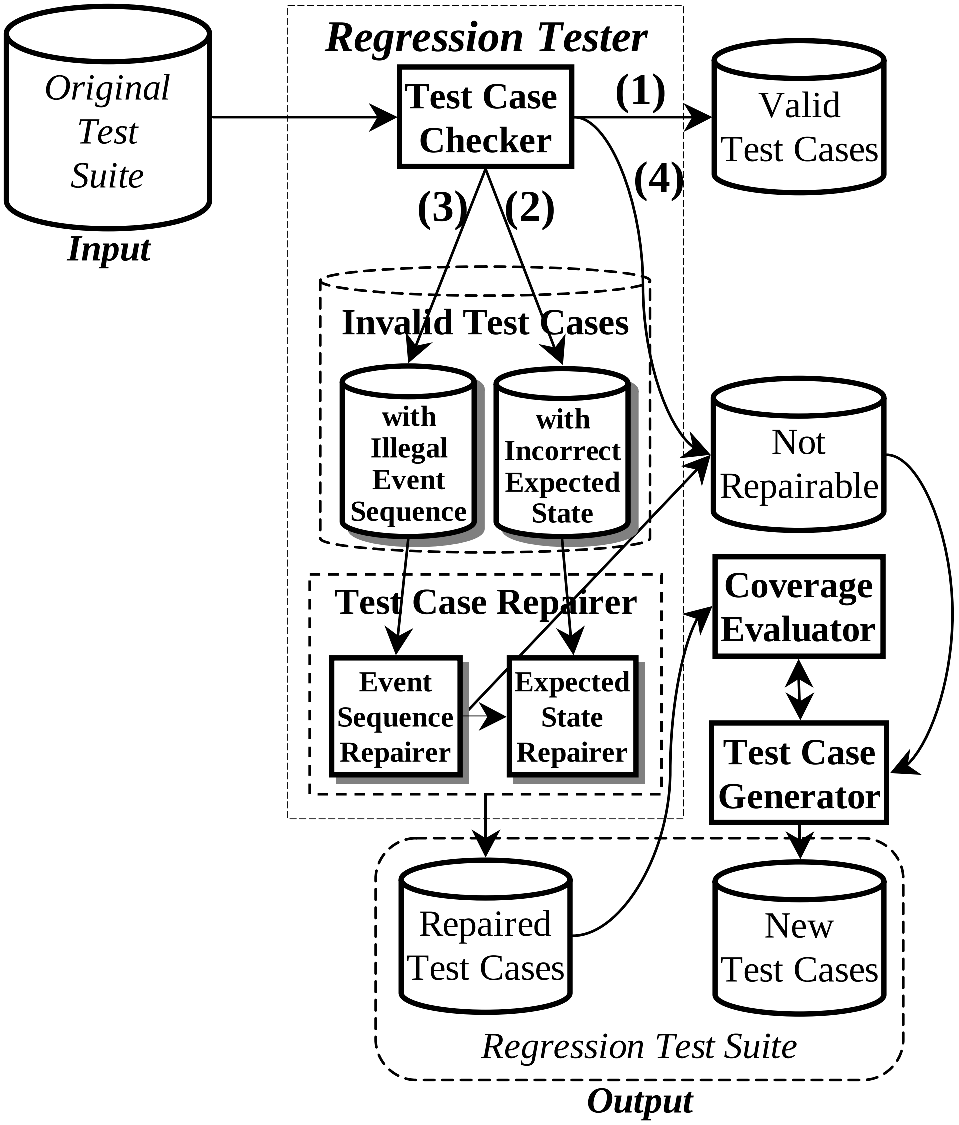


Figure 7.3: The Regression Tester's Components and their Interactions with other Components of the GUI Testing Framework.

modified GUI. In case some events or components are renamed, then the test designer is made aware of these changes by the GUI developer who must maintain a log of all such changes.

The analysis used to identify GUI modifications is straightforward and efficient, involving the computation of simple additions and deletions to the event-flow graphs and integration trees. Because of the simplicity, there are restrictions on the types of GUI modifications that may be detected. For example, if an event e is moved from one component C_x to another component C_y, then it will be analyzed as a deletion of e from component

C_x and an addition of e to component C_y. Consequently, the test case repairer is unable to detect the movement of e, and hence repair the test cases made invalid by the modification by invoking C_y instead of C_x and executing e. However, as will be seen in subsequent sections, not being able to analyze such modifications is a small price to pay for the simplicity of the analysis and the efficiency with which a number of invalid test cases can be repaired.

7.3.1 Intra-component Analysis

The goal of intra-component analysis is to determine changes made to events within a component. The results of this analysis are used by the test case checker to identify invalid test cases. The following modifications may be made to events within a component, represented by an event-flow graph:

1. a vertex may be deleted,
2. a vertex may be added,
3. an edge may be deleted, and
4. an edge may be added.

If EFG_o and EFG_m are the event-flow graphs of a component that exists in both the original GUI and the modified GUI respectively, then the following sets of modifications are obtained by performing set subtraction. Note that the functions $Vertices$ and $Edges$ return the sets $\mathbf{V}$ (the set of vertices) and $\mathbf{E}$ (the set of edges) for the event-flow graph in question.

1. The set of all new vertices in the event-flow graph:
   vertices_added $\leftarrow Vertices(EFG_m) - Vertices(EFG_o)$;
2. The set of all vertices deleted from the original event-flow graph:
   vertices_deleted $\leftarrow Vertices(EFG_o) - Vertices(EFG_m)$;
3. The set of all new edges added to the event-flow graph:
   efg_edges_added $\leftarrow Edges(EFG_m) - Edges(EFG_o)$;
4. The set of edges deleted from the original event-flow graph:
   efg_edges_deleted $\leftarrow Edges(EFG_o) - Edges(EFG_m)$;

As illustrated earlier in Section 7.1, the above sets can be used to identify invalid test cases. Details of how these sets of modifications are used by the *event-sequence checker* to identify invalid test cases are presented in Section 7.4.

7.3.2 Inter-component Analysis

Intra-component analysis is used to detect changes made to events within components. Similarly, changes may also be made at the component level in the GUI. Such modifications are reflected by a change in the structure of the GUI's integration tree. The following changes may be made to an integration tree:

1. a component may be added,
2. a component may be deleted,
3. an edge may be added, and
4. an edge may be deleted.

Let T_o and T_m be the integration trees of the original and modified GUI respectively. The following sets of modifications may be obtained from these two integration trees. Note that $Nodes$ and $CompEdges$ return the sets $\mathcal{N}$ and $\mathcal{B}$ for the integration tree respectively.

1. The set of components added to the integration tree:
   components_added $\leftarrow Nodes(T_m) - Nodes(T_o)$;
2. The set of components deleted from the integration tree:
   components_deleted $\leftarrow Nodes(T_o) - Nodes(T_m)$;
3. The set of edges added to the integration tree:
   comp_edges_added $\leftarrow CompEdges(T_m) - CompEdges(T_o)$;
4. The set of edges deleted from the integration tree:
   comp_edges_deleted $\leftarrow CompEdges(T_o) - CompEdges(T_m)$;

Note the difference between the edges of an event-flow graph and integration tree. Edges of an event-flow graph are ordered pairs of the form (e_x, e_y), where e_x and e_y are events, whereas edges of the integration tree are ordered pairs of the form C_x, C_y, where C_x and C_y are components. Each edge of the integration tree represents a set of edges with events. An edge (C_x, C_y) represents the set of all edges (e_y, e_z), where e_y is a restricted-focus event in component C_x that invokes C_y, and $e_z \in \texttt{follows}(e_y)$ (computed in Figure 3.11, `Lines 13, 14`). Assume the existence of a new function `EventEdges` that takes a set of integration-tree edges and returns its corresponding set of edges in terms of events.

The set of modifications obtained by the intra- and inter-component analyses are used to classify modifications made to the GUI. Such a classification helps the test case checker identify invalid test cases. Its operation is described next.

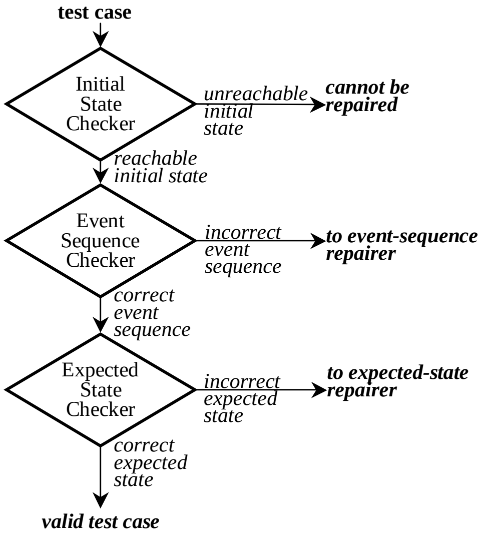


Figure 7.4: Parts of the Test Case Checker.

7.4 Determining Affected Test Cases

The test case checker's primary function is to identify invalid test cases. In addition, it performs preliminary identification of non-repairable test cases. The logical functionality of the test case checker is summarized as a graph in Figure 7.4. The nodes in the graph correspond to three parts of the test case checker that check the validity of a test case. The components are:

Initial State Checker determines whether the initial state S_0 associated with the test case is reachable. A test case with an unreachable initial state is useless since the GUI cannot be brought into the state to execute the test case. If $S_0 \in S_I$ (the set of valid initial states of the GUI), then it is reachable; otherwise the checker reduces the problem of checking the initial state to one of plan generation. Following are the elements of the planning problem:

Initial State for the planning problem is a state $S_x \in S_I$,

Goal State is S_0, and

Operators are the planning operators of the GUI.

If, for at least one $S_x \in S_I$, a plan is found, i.e., a sequence of events exists in the GUI to transform S_x to S_0 then S_0 is a reachable initial state; otherwise it is unreachable. In case the initial state is unreachable, the test case is not repairable.

Event-Sequence Checker determines whether the event sequence in the test case is a legal event sequence for S_0. It uses the sets of modifications obtained from the GUI modification analysis to identify test cases that were made invalid. Specifically, the following two sets are used to identify invalid test cases:

1. **vertices_deleted**, and
2. **edges_deleted** $\longleftarrow$ **efg_edges_deleted** $\cup$ `EventEdges`(**comp_edges_deleted**).

As noted earlier, new vertices and edges cannot make test cases invalid. To aid in the identification of invalid test cases, the event-sequence checker uses bit vectors associated with each test case. These bit vectors contain information about the events and edges used by each test case. If a test case uses an event (or edge), then the event's (or edge's) bit is set in the bit vector for that test case. The following bit vectors are associated with each test case T:

EVENTS-USED represent the events used by T. Its length is $|E|$, where E is the set of events in the GUI.

EDGES-USED represent the edges covered by T. Its length is $|D|$, where D is the set of all the edges in the event-flow graphs and integration tree of the GUI.

Examining the above bit vectors for each modification, the event-sequence checker identifies test cases that were made invalid by each modification. For example, if an event e is deleted from the GUI, then all test cases whose EVENTS-USED bit vector's e^{th} bit is set are invalid. Note that one GUI modification may be reflected in more that one set of modifications, and a test case may be marked as invalid several times because of the same modification. As will be seen later, being marked as invalid several times has no effect on the repairability of the test case.

Expected State Checker determines whether the expected state sequence associated with each test case is valid. If the initial state and event sequence of a test case are valid, then the test case can be executed on the GUI. However, if the preconditions/effects of an event have been modified in the GUI then the expected state sequence associated with this test case may be incorrect. Such modifications are detected statically by comparing the modified and the original operator for each event.

Once the invalid test cases have been identified, they are repaired by the test case repairer, which is described next.

7.5 Test Case Repairer

The test case repairer consists of two parts: the *expected-state repairer* and the *event-sequence repairer*. The expected-state repairer employs the expected-state generator

```
ALGORITHM : EventSeqRepairer(
 S: Invalid event sequence; /* The event sequence to be repaired */
 vertices_deleted: Set of vertices; /* The set of all the deleted events */
 edges_deleted: Set of edges; /* The set of all the deleted edges */
 EVENTS: Set of events; /* All the events in the modified GUI */
 EVENTS-USED: Bit vector; /* The events in the sequence */
 EDGES-USED: Bit vector) /* The edges in the sequence */
{
 FOREACH ($e_i \in$ vertices_deleted) DO /* Examine each event that was deleted */
  WHILE ($e_i^{th}$ bit of EVENTS-USED == 1) DO /* As long as S uses this event */
   repairability ⟵ repair_del_event(t, $e_i$); /* repair S */
   IF (! repairability) THEN RETURN(FALSE); /* If S is not repairable, then terminate */
   UPDATE(EVENTS-USED, S); /* Update the changes */
 UPDATE(EDGES-USED, S); /* Update the edges */
 FOREACH (($e_i, e_j$) $\in$ edges_deleted) DO /* Examine each edge that was deleted */
  IF (($e_i \in$ EVENTS && $e_j \in$ EVENTS) THEN /* Events are still available? */
   WHILE (($e_i, e_j$)$^{th}$ bit of EDGES-USED == 1) DO /* As long as S uses the edge */
    repairability ⟵ repair_del_edge(S, ($e_i, e_j$)); /* repair S */
    IF (! repairability) THEN RETURN(FALSE); /* If S is not repairable, then terminate */
    UPDATE(EDGES-USED, S); /* Update the changes */
 RETURN(TRUE); /* Success!! */
}
PROCEDURE : repair_del_event(
 S: Event Sequence; /* The event sequence */
 e: Event) /* The event that was deleted. At position p in the event sequence */
{
 FOR k ⟵ p+1 TO n DO /* start scanning the event sequence */
  IF $e_k \in$ follows($e_{p+1}$) THEN /* if Case 1 is solved */
   S ⟵ $< e_1; \ldots; e_{p-1}; e_k; \ldots; e_n >$; /* then update the event sequence */
   done1 ⟵ TRUE; break; /* event sequence repaired */
  ELSE IF $\exists\ e_x$ (($e_x \in$ follows($e_{p-1}$)) && ($e_k \in$ follows($e_x$))) THEN /* if Case2 is solved */
   S ⟵ $< e_1; \ldots; e_{p-1}; e_x; e_k; \ldots; e_n >$; /* then update the event sequence */
   done2 ⟵ TRUE; break; /* event sequence repaired */
  RETURN (done1 || done2); /* In either case's success, return success */
}
PROCEDURE : repair_del_edge(
 S: Event Sequence; /* The event sequence */
 ($e_a, e_b$): Edge) /* The edge that was deleted. $e_b$ is at position b in the event sequence */
{
 FOR k ⟵ b TO n DO
  IF $e_k \in$ follows($e_a$) THEN
   S ⟵ $< e_1; \ldots; e_a; e_k; \ldots; e_n >$;
   done1 ⟵ TRUE; break;
  ELSE IF $\exists\ e_x$ (($e_x \in$ follows($e_a$)) && ($e_k \in$ follows($e_x$))) THEN
   S ⟵ $< e_1; \ldots; e_a; e_x; e_k; \ldots; e_n >$;
   done2 ⟵ TRUE; break;
  RETURN (done1 || done2);
}
```

Figure 7.5: Algorithm for the Event-sequence Repairer.

(Section 6.1) to repair the incorrect expected states. Knowing which definition of event e_i was changed, already determined by the expected-state checker, the expected-state repairer uses the expected state S_{i-1} of the test case to generate all successive expected states $S_i; S_{i+1}; S_{i+2}; \ldots$ by applying the operators corresponding to the events in the test case iteratively until it reaches a correct expected state or the end of the event sequence. The repaired test case is valid and may be used for regression testing.

The event-sequence repairer repairs illegal event sequences. The illegal event sequences use either a deleted event or a deleted edge. Intuitively, if an event e_i, at position i in an event sequence, is deleted from the GUI, then the event-sequence repairer must remove e_i from the event sequence. However, to obtain a legal resulting event sequence, the event-sequence repairer scans the event sequence from left to right, starting at position $i + 1$, until it finds an event e_j such that either: (1) $< e_{i-1}; e_j >$ is a legal event sequence for the modified GUI, or (2) there is another event e_x, from the set of all the events in the modified GUI, such that $< e_{i-1}; e_x; e_j >$ is a legal event sequence for the modified GUI.[1] Once such an e_j is found, then the sub-sequence $< e_i; \ldots; e_{j-1} >$ is deleted from the event sequence and in case 2, e_x is inserted. Figure 7.6(a) shows these two cases. In case 1, the event-sequence repairer searches for an event e_j from e_{i+1} to e_n, such that e_{i-1} `follows` e_j, and in case 2, it searches for an event e_x, from the set of all the events in the modified GUI, such that e_{i-1} `follows` e_x and for some e_j in the event sequence, e_j `follows` e_x.

Similarly, Figure 7.6(b) shows the repairing technique for the deleted edge (e_i, e_j). In this technique, the event sequence is scanned from left to right, starting with the event e_j, the second element in the deleted edge. Case 1 tries to find an event e_a from the subsequence $< e_j; \ldots; e_n >$ such that e_a `follows` e_i. Case 2 tries to find an event e_x, from the set of all the events in the modified GUI, such that e_x `follows` e_i and e_j `follows` e_x.

As noted earlier, an event sequence may have become illegal because of several changes made to the GUI. Each event sequence is checked for all instances of deleted events and edges that made the event sequence illegal.

The algorithm for the event-sequence repairer is shown in Figure 7.5. The main algorithm is called `EventSeqRepairer` that takes a number of parameters: (1) the invalid event sequence **S**, (2) the set vertices_deleted, (3) the set edges_deleted, (4) the set of all the events available in the modified GUI, (5) the bit vector **EVENTS-USED** associated with the event sequence, and (5) the bit vector **EDGES-USED**. `EventSeqRepairer` returns TRUE if the event sequence was repaired successfully, and FALSE otherwise. The algorithm

[1] In general, this technique may be extended to finding a sequence of events $< e_p; \ldots; e_q >$ such that $< e_{i-1}; e_p; \ldots; e_q; e_j >$ is a legal event sequence for the modified GUI. However, computing such a sequence is expensive.

starts by examining each event e_i that was deleted from the GUI (Line 9). If **S** uses this event (`Line 10`), then it is illegal. The procedure `repair_del_event` is invoked to repair **S** (`Line 11`). If **S** is repairable, then `repair_del_event` returns TRUE, otherwise `EventSeqRepairer` terminates with a FALSE result (`Line 12`). Since `repair_del_event` may have changed the events used by **S**, the bit vector **EVENTS-USED** is updated to reflect the changes (`Line 13`). Note that the WHILE loop continues examining the event sequence for the deleted event e_i. After **S** has been repaired for all deleted events, its **EDGES-USED** is updated to reflect all the changes made so far (`Line 14`). `EventSeqRepairer` continues by examining each edge (e_i, e_j) that was deleted (`Line 15`). It makes sure that both events e_i and e_j are available in the GUI (`Line 16`). If **S** uses this edge (`Line 17`), then it is illegal. The procedure `repair_del_edge` is invoked to repair **S** (`Line 18`). If **S** is repairable, then `repair_del_edge` returns TRUE, otherwise `EventSeqRepairer` terminates with a FALSE result (`Line 19`). **EDGES-USED** is updated to reflect the changes made to **S** (`Line 20`). If `EventSeqRepairer` has not terminated using any of the RETURN statements (`Lines 12, 19`), then the event sequence has been successfully repaired (`Line 21`).

The procedure `repair_del_event` tries to repair the illegal event sequence caused by deleting an event. It takes two parameters: (1) the event sequence **S**, and (2) the deleted event e. It starts scanning the subsequence $< e_{p+1}; \ldots; e_n >$ from left to right (`Line 27`) until one of the cases shown in Figure 7.6(a) is found or the sequence terminates. If case 1 is solved (`Line 28`), then the sequence is updated (`Line 29`) and success reported (`Line 30`). Otherwise if case 2 is solved (`Line 31`), then the sequence is updated (`Line 32, 33`). The procedure `repair_del_edge` is similar to `repair_del_event`. It scans the subsequence $< e_b; \ldots; e_n >$ from left to right until one of the cases of Figure 7.6(b) is found.

Note that since the event-sequence repairer employs information from the event-flow graphs and integration tree (represented by `follows`), the event sequence repairer is guaranteed to produce legal event sequences. Once these sequences have been repaired, their expected states are repaired by the expected-state repairer.

7.6 Experiments

To explore the practicality of the test case repairing technique, the regression tester was implemented and its performance evaluated on an example GUI. The experiment consisted of the following steps:

1. *Choice of GUI:* The experiment was performed on the same version of the WordPad software used throughout the dissertation as a running example.

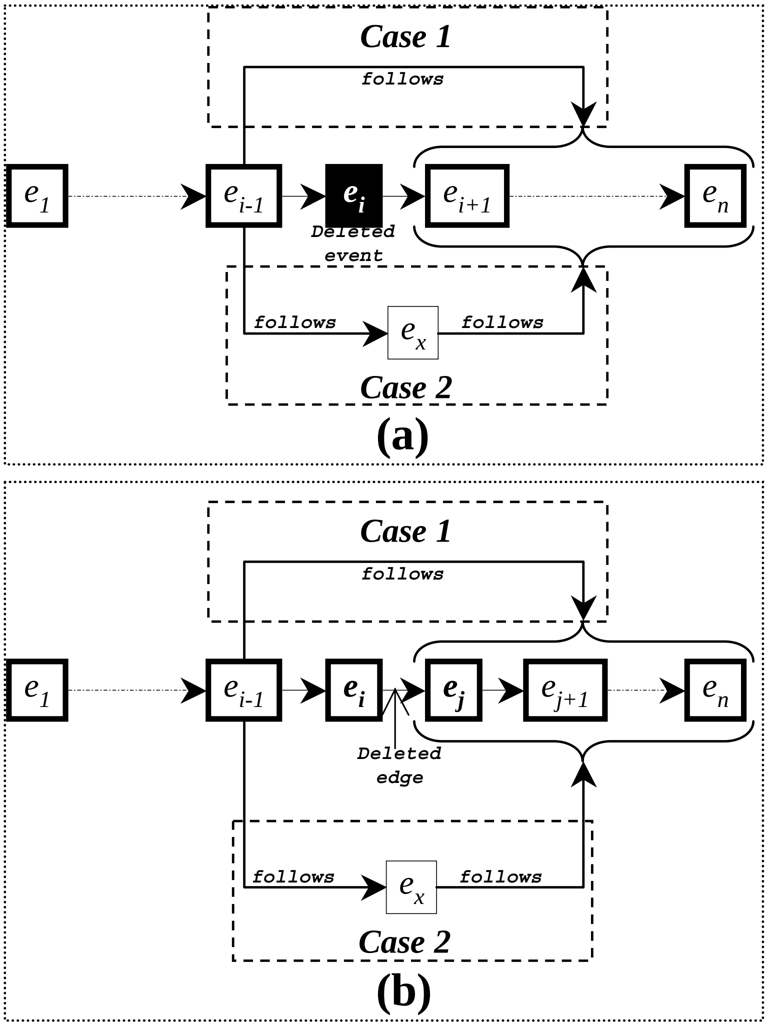


Figure 7.6: Repairing an Event Sequence that Uses a (a) Deleted Event e_i, and (b) Deleted Edge (e_i, e_j).

2. *Generating test cases:* All event sequences of length < 4 were generated for the WordPad's `Main` component in 120.83 seconds CPU time. In all, 270921 event sequences were generated.
3. *Modifying the GUI:* A modified version of the WordPad GUI was created by (1) replacing the `File` event by a new event called `NewFile`, and (2) modifying the `follows` of `Cancel` to the events of the `Find` window (see Figure 3.10).
4. *Identifying invalid test cases:* The test case checker was implemented in Perl. Of the 270921 original test cases, 57100 were found to be invalid.
5. *Repairing test cases:* The test case repairer was also implemented in Perl. The total time to repair all invalid test cases was 7.83 seconds CPU time.

This preliminary experiment showed that the repairing technique is practical. In future work, more experiments need to be conducted using a real-world regression testing example.

7.7 Conclusions

This chapter presented the design of the regression tester, which is based on a new technique that reuses some *invalid test cases* by *repairing* them. These test cases are repaired by employing the specifications of the GUI to make the repairs. Differences between the event-flow graphs and integration trees of the original and modified GUIs are obtained to identify invalid test cases. Feasibility experiments show that the regression testing technique is efficient, in that it is cheaper to repair existing invalid test cases than to generate new ones.

The modifications discussed in this chapter were complex event-level and component-level modifications. Other low-level modifications may also be made to a GUI. For example, new keyboard shortcuts may be introduced in the modified GUI or the physical locations of buttons/menus may be changed. Such changes do not affect the test cases since all the events in the test case are represented by logical symbols rather than low-level physical locations on the screen or keyboard shortcuts used to generate them. A mapping between logical events and the corresponding physical actions used to generate them is maintained. At test case execution time, the mapping is used to generate physical actions for each logical event. When these events/shortcuts are changed from one GUI version to the next, the mappings are modified without affecting the test cases.

New test cases may be required to test parts of the GUI for which the original test cases could not be repaired. This problem can be easily solved in the context of PATHS. The initial and goal states for non-repairable test cases may be reused to generate new test cases by rerunning PATHS. Note that repairing two different test cases may yield test cases that are the same. Analysis to remove repeated test cases must be done before they are executed.

Chapter 8

Testing Web User Interfaces

The recent popularity of the Internet has led to the widespread use of web user interfaces (WUIs). WUIs present an integrated front-end to software typically consisting of multiple programs, possibly implemented in different languages, concurrently executing on several platforms, and connected by the Internet. The user interacts with the WUI, through a *web-browser's* window, without knowledge of the underlying software, topology of the Internet, or the implementation platforms. The WUI user expects the entire system to work as if it was executing on the local client.

Similar to GUIs, the input to the WUI is in the form of events and the output is graphical. In fact, WUIs have all the characteristics of GUIs, including event-driven input that changes the WUI's state, graphical output, hierarchical structure, and graphical objects with properties. Hence, testing WUIs has all the complexities of testing GUIs discussed in Chapter 1. In addition, WUIs have special characteristics, such as timing and synchronization constraints and very high portability requirements that makes testing them even more complex than GUIs.

The important characteristics of WUIs include their graphical orientation, connectivity to the Internet, event-driven input, frames, pages and the constraints among pages, the objects they contain, constraints among objects, and properties (attributes) of those objects. Formally, a WUI may be defined as follows:

Definition: A *Web User Interface (WUI)* is a GUI in which the hierarchical structure consists of frames and pages, with geometric and temporal constraints among pages. Each page contains objects and constraints among the objects. The WUI provides a graphical front-end to a software consisting of multiple programs, possibly implemented in different languages, concurrently executing on several platforms, all connected by the Internet. □

This chapter presents a cursory exploration of extending the GUI testing framework to include WUIs. Because of the additional complexities of WUIs, testing them has certain requirements. A representation of the WUI and its operations should include a representation of the multiple programs that determine the state of the WUI. These programs may execute on the server and produce static output (such as HTML) or dynamic output (such as DHTML) generated on the fly to be displayed in the browser's window. Other programs such as Java Applets may execute on the local client and their interface (usually a GUI) may be displayed as part of the WUI. Checking the correctness of the WUI should include checking the correctness of the GUIs of these individual programs. Synchronization relations may exist between these programs, which should also be checked.

A typical user, interacting with the WUI performs at least three types of events: (1) those available in the browser (such as `cut` and `copy`), (2) those available in the browser's window (such as clicking on links, selecting an item from a drop-down list, and clicking on buttons), and (3) those provided by the multiple programs' GUIs executing in the browser (such as Java Applets and plug-ins). Testing the WUI should include performing all these interleaved events.

The WUI's state depends largely on the *environmental conditions* in which it is executed. These environmental conditions include the state of the server, client and network. Examples of server-specific environmental conditions include its speed and the state of its file system. Client-specific environmental conditions include display size, security settings, installed components, geographic location, and installed hardware. Network-specific environmental conditions include its speed and connectivity. When testing the WUI, these environmental conditions also form a part of the test input. Moreover, coverage evaluation should also determine the adequacy of the different environmental conditions in which the WUI was tested.

Currently, there are three different approaches to WUI testing. The *automated approach* simulates a web-browser by generating requests, e.g., HTTP requests by using one of several *HTTP torture machines* [5]. The response to each request is then analyzed and its correctness in the context of the single request determined. The disadvantage of this approach is that the tester lacks a global perspective of a typical users' interactions and the collective effect of a sequence of events as seen on a browser's window. Because of this limitation, this testing is restricted to *load testing* [5] of the servers to determine the number of requests they can handle simultaneously. Another approach, which is *semi-automated* and the most popular, is to employ *capture/replay* tools similar to those used for GUIs [81]. The test designer captures an interaction with the WUI, edits the captured script to create slightly different test cases and executes them automatically on the WUI. However, the capture/reply tools provide limited support for checking the output. Moreover, the overall coverage of the test cases depends largely on the test designer's first interaction with the WUI. The last and most expensive is the *manual approach*, which produces the most realistic test cases. Human testers interact with the WUI, trying to find errors to help test the WUI. Since this approach is resource intensive, it is usually performed by a large number of users on beta-releases of the WUI. For example when Janus (www.janus.com) was upgrading its WUI in July 2000, they invited customers to use the new WUI and report any problems before they actually installed it on their web-site.

Subsequent sections present preliminary ideas to explore how to extend the framework to test WUIs. The goal is to combine the benefits of the above three approaches (automated, semi-automated, and manual) by automatically generating and executing test cases on the WUI. In particular, the GUI representation may be extended to incorporate geometric and temporal constraints among WUI objects. Instead of a hierarchy based on modal windows, a new hierarchy of WUI objects is presented in terms of pages and frames. Timing information is incorporated into WUI test cases. The test oracle is extended to include a new component called a timing monitor that checks the correctness of the temporal and synchronization constraints. An approach that uses the *category-partition method* [59] to select environmental conditions is described. Test cases are executed using "important" combinations of environmental conditions by assigning priorities to them. Finally, a technique that employs user profiles for regression testing of WUIs is presented.

8.1 Pages, Frames, and Constraints

A WUI contains *objects* designed to accept input from a WUI user and present output to be displayed in the browser. Examples of objects include text items, text boxes, images, Java Applets, buttons, and links. These WUI objects are logically grouped together into pages and pages into frames. Note that these groupings increase the usability of the WUI by displaying related objects together.

Intuitively, a page creates a layout of WUI objects for the browser and establishes timing and synchronization relationships among them. Formally, a page is defined as follows:

Definition: A *page* is a pair (O, C), where each $o \in O$ is a WUI object and each $c \in C$ is a constraint on the elements of O. □

Common examples of constraints are geometric constraints that define the layout of the objects in the WUI and temporal/synchronization constraints. Note that additional levels of grouping can be similarly represented, e.g., *frames* can be represented by constraints on a set of pages. Frames in WUIs force a dialog similar to a modal dialog in GUIs. Events

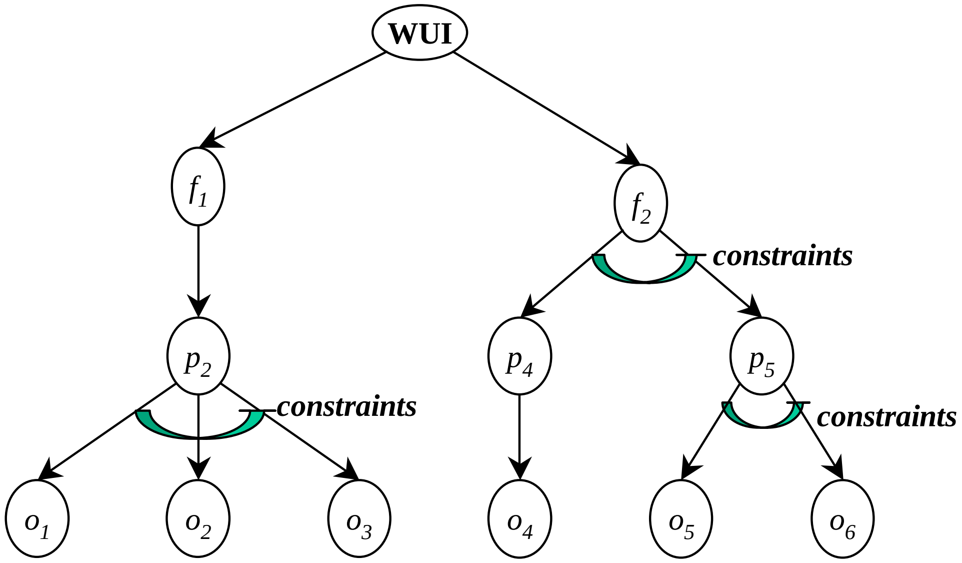


Figure 8.1: A WUI as a Hierarchy of Pages, Frames and Objects with Constraints.

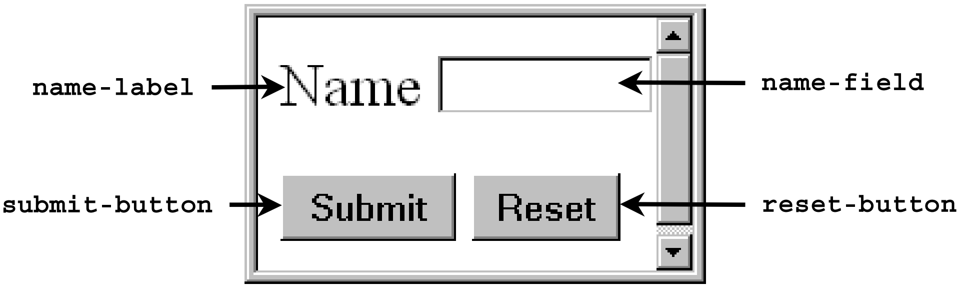


Figure 8.2: A WUI Example.

in two different frames cannot be interleaved. Their respective frames must be invoked or terminated. For example, Figure 8.1 shows a WUI decomposed into frames, pages and objects. Each frame (f_1 and f_2) contains pages (p_1, p_2, and p_3) with several objects (o_1, o_2, …, o_6). Events on o_1, o_2, and o_3 cannot be interleaved with events on o_4, o_5, and o_6. Note that events performed on o_4, o_5, and o_6 can be interleaved since pages p_2 and p_3 are displayed in the same frame (f_2) and, hence, are simultaneously visible to the user. These characteristics of pages and frames may be used to identify *WUI components* similar to the ones developed for GUIs.

The simple WUI shown in Figure 8.2 may be modeled in terms of its objects with properties and the constraints among the objects. The WUI contains four objects, `name-label`, `name-field`, `submit-button`, and `reset-button`. The contents of the WUI are summarized as follows:

Frames: f_1 */* A single frame*/*

Pages: p_1 */* A single page */*

Objects of p_1:

- `name-label`: set of properties = {type("`label`"), value("`Name`"), color("`Black`"), font("`Type Roman`")}.
- `name-field`: set of properties = {type("`text-field`"), value(""), editable("`TRUE`")}.
- `submit-button`: set of properties = {type("`button`"), caption("`Submit`"), action("`POST`")}.
- `reset-button`: set of properties = {type("`button`"), caption("`Reset`"), action("`RESET`")}.

Constraints: */* geometric constraints imposed by the HTML code */*

{`first-object(name-label)`, `after(name-label, name-field)`,
`new-line(submit-button)`, `after(submit-button, reset-button)`}

The properties for each WUI object describe the characteristics of that object. The property "type" describes the type of the object, hence determining its behavior and the interpretation of its remaining properties. The property "action" associates an executable program with the object in question. For example, `submit-button` and `reset-button` have the actions `POST` and `RESET` associated with them.

Note that the WUI representation is more complex than that of GUIs. In WUIs, timing and position constraints play important roles in its execution. The next section shows how timing and synchronization information are incorporated into WUI test cases.

8.2 Representing Timing Information in WUI Test Cases

Temporal and synchronization constraints are an important part of a WUI's behavior. A common example of a temporal constraint on a WUI event is the maximum time allowed for that event to execute. Other constraints, such as synchronization constraints may require that an object be downloaded completely before the next event is executed. Such temporal constraints may be defined for each event in the test case by a *timing/synchronization sequence.*

Definition: A *timing/synchronization sequence* $T_1; T_2; T_3; \ldots; T_n$ is associated with each WUI test case, where each T_i is a set of temporal/synchronization constraints on event e_i. □

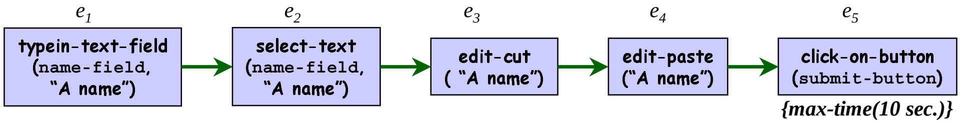


Figure 8.3: A WUI Event Sequence.

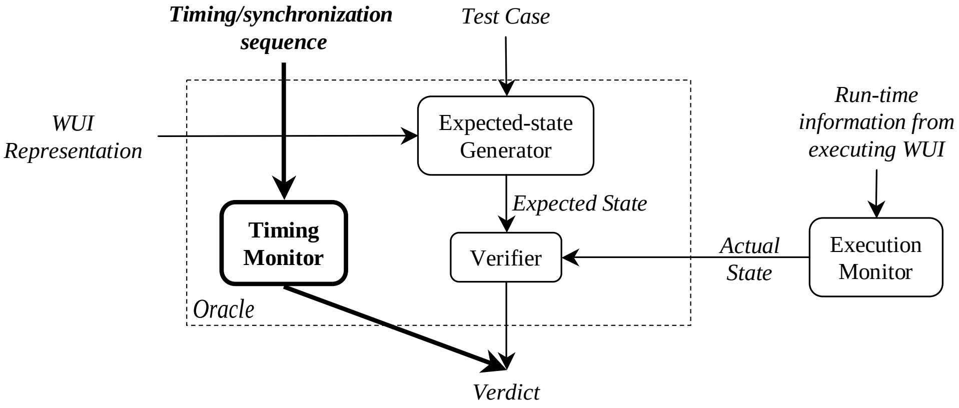


Figure 8.4: Extending the Oracle to Handle Temporal Constraints.

For example, consider the event sequence shown in Figure 8.3 for the WUI of Figure 8.2. The test case consists of 5 events – `typein-text-field` (e_1) and `click-on-button` (e_5) are events available in the browser window whereas `select-text` (e_2), `edit-cut` (e_3), and `edit-paste`(e_4) are events available in the browser. Event e_5 has a temporal constraint that imposes a limit on the time elapsed between its execution and the display of results. If this time is longer than 10 seconds, then an error must be reported.

The test designer can define any type of temporal constraint. These constraints are used by the test executor to control the execution of each event and by the test oracle to determine the correctness of the timing of the test case. Hence, for each temporal constraint defined by the test designer, appropriate routines must be developed in the test executor and test oracle to handle that constraint. Note that some synchronization constraints are automatically handled by the test executor. For example, the test executor waits for an object to be loaded before performing an event on that object. The test oracle developed in Chapter 6 is extended to handle WUIs (see Figure 8.4). A new component, called a *timing monitor* uses the timing/synchronization sequence and verifies the correctness of the timing as defined in the sequence.

8.3 Environmental Conditions

As mentioned earlier, *environmental conditions* may affect a WUI's execution behavior. Common examples are the client's security settings, the browser used, and the speed of the network. The WUI must be tested on a sufficient number of variations of the environmental conditions. The test executor in the testing framework is extended to initialize the WUI's execution environment to the environmental conditions on which it is tested. The behavior of each event may change depending on environmental conditions chosen for testing the WUI.

There are several possible approaches to handle the effect of environmental conditions on events.

1. *Ignore them:* Each event's behavior would be non-deterministic, making it essentially impossible to validate test results or to re-execute test cases.
2. *Explicit encoding:* Encode the environmental conditions as parameters to each event's operator and modify the operator definition for each environmental condition.
3. *Demand driven:* Instead of encoding the environment conditions explicitly in each event's operator, take each condition's effect into consideration at test case execution time.

While approach 1 is clearly unacceptable, 2 and 3 provide similar results. However, in approach 2, the specification of each operator becomes bulky and non-intuitive. Moreover, as new environmental conditions are identified and old, less important ones discarded, the test designer may have to change the operators. On the other hand, approach 3 allows the test designer to specify and handle important environmental conditions whenever necessary.

An examination of the events of the WUI yields the characteristics of the client, server, and network's state that effects the event's execution behavior. As in the category-partition method, *categories* classify these characteristics. *Choices* are the different significant cases that can occur within each category.

Formally, the categories of the environmental conditions of a specific WUI are $C = \{c_1, c_2, \ldots, c_n\}$. For each c_i, the choices are $H_i = \{h_{i1}, h_{i2}, \ldots, h_{im}\}$.

Definition: The *category-choices* $\mathcal{CC}$ of a WUI is a set of ordered pairs $(c_i, \{h_{i1}, h_{i2}, \ldots, h_{im}\})$ where $c_i \in C$ is a category and each h_{ij} is a choice of category c_i. □

Note that each WUI has a unique $\mathcal{CC}$ since the categories and choices are obtained by examining the events of the WUI. However, once the category-choices have been identified for a WUI, they can be reused with very few alterations across WUIs since many categories and choices are common across WUIs. Web-browsers can also be used to obtain some of these categories and choices. For example, `Internet Options` in Microsoft's Internet Explorer gives a list of options to set the client's preferences (environmental conditions). The choices available in the web-browser can be used to construct the category-choices.

The experience of the test designer plays an important role in selecting the categories and choices. The test designer (1) examines all the events in the WUI and identifies the characteristics of the client, server, and network state that effects the event's execution behavior (2) classifies the characteristics into categories, and (3) determines the different significant cases that can occur within each category. These cases become the choices of the category.

During test case execution, values are chosen for each category from its corresponding choice list. Hence, the input at test execution time consists of $\mathcal{CC}$ as well as the test cases.

$$\textbf{Input} = \{\mathcal{CC}\} \times \{ \text{ Test Cases } \}$$

$\mathcal{CC}$ is used by the test executor to initialize the environment of the WUI before executing each test case.

It is impractical to test the WUI for all possible combinations of choices for each category. Important choices must be identified by the test designer. The test designer assigns priorities to each choice, creating *extended category-choices.*

Definition: The *extended category-choices* $\mathcal{CC}'$ is a set of ordered pairs of the form $(c_i, \{(h_{i1}, I_{i1}), (h_{i2}, I_{i2}), \ldots, (h_{im}, I_{im})\})$, where c_i is a category and I_{ij} is the priority assigned to the choice h_{ij}. □

An example of $\mathcal{CC}'$ is shown in Table 8.1. The table shows 4 categories: the browser, connection speed, operating system, and the level of security. The columns show the choices of each category and the priority assigned to each choice. For example, column 1 shows all the choices of the category browser. The priority of the choice "IE" is 0.6 and that of "Netscape" is 0.4. Using the extended category-choices, the test designer orders the setting of the environmental conditions by using the choices with highest priority first. For example, in Table 8.1, the maximum number of test cases will be executed on the WUI by using the IE browser, low security settings, Linux operating system, and connected by a 28.8Kbps modem. Then depending on the resources available, some test cases may be executed with lower priority choices.

c_1 **Browser**		c_2 **Cnx. Speed**		c_3 **Opr Sys**		c_4 **Security**	
$h_{1,1}$ IE	$I_{1,1}$ 0.6	$h_{1,2}$ T_1	$I_{1,2}$ 0.1	$h_{1,3}$ WinNT	$I_{1,3}$ 0.1	$h_{1,4}$ High	$I_{1,4}$ 0.1
$h_{2,1}$ Netscape	$I_{2,1}$ 0.4	$h_{2,2}$ 56 kbps	$I_{2,2}$ 0.3	$h_{2,3}$ Win2000	$I_{2,3}$ 0.3	$h_{2,4}$ Medium	$I_{2,4}$ 0.4
		$h_{3,2}$ 28.8 kbps	$I_{3,2}$ 0.6	$h_{3,3}$ Linux	$I_{3,3}$ 0.4	$h_{3,4}$ Low	$I_{3,4}$ 0.5
				$h_{4,3}$ IBM	$I_{4,3}$ 0.1		

Table 8.1: An Example of Extended Category-choices.

8.3.1 User Profiles for Regression Testing

Once the WUI has been deployed, valuable information may be collected about its usage. Although such information is not readily available for conventional software [60], web-based software already collects this information in log files. These log files may be data-mined [46] and used in the following ways for regression testing.

1. The log files may be used to identify event-sequences that users employ to interact with the WUI and extract common patterns. These patterns can then be used to generate test cases for the modified WUI.
2. The log files may also be used to identify new categories, their choices and assign priorities to the choices.

Using profile information, the test designer is better informed about the WUI's usage and is thus able to perform better regression testing of the WUI.

8.4 Conclusions

This chapter presented some of the important problems of WUI testing and presented possible extensions to the testing framework to solve them. The GUI representation was extended to incorporate constraints among WUI objects. A new hierarchy of WUI objects was presented in terms of pages and frames. Timing information was incorporated into WUI test cases. A new component called a timing monitor was added to the test oracle allowing it to check the correctness of the temporal and synchronization constraints. The category-partition method was used to select environmental conditions for the WUI. Finally, a technique that employs user profiles for regression testing of WUIs was presented.

Chapter 9

Conclusions and Future Work

The widespread recognition of the usefulness of graphical user interfaces (GUIs) has established their importance as critical components of today's software. Although the use of GUIs continues to grow, GUI testing has remained a neglected research area. Testing GUIs requires the development of (1) *coverage criteria* to determine what to test in the GUI, (2) *test cases* based on the coverage criteria, (3) *test oracles* to determine whether the GUI executed correctly during testing, (4) a *regression test suite* to test the modified and affected parts of the GUI by selective test case execution.

Because GUIs have characteristics different from conventional software, such as event-based input and graphical output, techniques developed to test conventional software cannot be directly applied to GUI testing. Currently, the most popular tool support for GUI testing is in the form of record/playback tools, which are largely manual, making GUI testing resource intensive. Although a few independent tools and techniques to automate some aspect of GUI testing have been proposed in the published literature, they are rarely used in practice because a test designer who makes use of these independent tools has to learn the idiosyncrasies of each tool. A practical solution to the GUI testing problem must develop automated tools and techniques that are integrated and employ a common representation so that results of one tool are compatible with the others.

9.1 Summary of Contributions

This thesis develops a unified solution to the GUI testing problem with particular emphasis on the integration of tools and techniques to be used in the various phases of GUI testing. The integration goal was accomplished by the development of a framework with a GUI representation useful for all phases of testing. As the first step of testing, the test designer creates a model of the GUI that is used as input to all the tools/techniques.

The main contribution of this thesis is a comprehensive framework for testing GUIs. The framework consists of several interacting components: a GUI representation, a test case generator, test coverage evaluator, test oracle, test executor, and regression tester. The individual contributions of developing each of these tools/techniques are outlined next.

1. **Representation:** The representation of a GUI is a fundamental component of the framework. A GUI is represented as a set of objects, (window, menu, button, text, etc.), a set of properties of those objects (background color, font, is-open, etc.), and a set of events that change the properties of certain objects (set-background-color, etc.). Each GUI uses certain types of objects with associated properties; at any specific point in time, the GUI is described in terms of the specific objects, or GUI elements that it currently contains, and the current values of their properties. Events that are performed on the GUI are modeled as state transducers or *operators*. These operators are defined in terms of the *preconditions* and *effects* of the events. For efficiency and scalability, events are classified in a hierarchy as *restricted-focus events*, *unrestricted-focus events*, *termination events*, *menu-open events*, and *system-interaction events*. This classification is used to create a hierarchy of *GUI components* that is used by the test case generator, coverage evaluator, test oracle, and regression tester. A *GUI component* is defined as the basic unit of testing. A new representation of a GUI component called the *event-flow graph* identifies events and their interactions. An *integration tree* represents the interactions among components.
2. **Coverage Evaluator:** The coverage evaluator employs a new class of coverage criteria called *event-based coverage criteria*. These criteria use events and event sequences to specify a measure of test adequacy. The coverage evaluator employs (1) intra-component criteria for events within a component and (2) inter-component criteria for events across components. Three types of intra-component coverage criteria are used: *event coverage*, *event-interaction coverage*, and *length-n event-sequence coverage*. The coverage evaluator employs *invocation coverage*, *invocation-termination coverage*, and *inter-component length-n event-sequence coverage* criteria for events across components.
3. **Test case generator:** The test case generator is based on a new technique that exploits planning, a well developed and used area of artificial intelligence. Given a set of operators, an initial state and a goal state, a planner produces a sequence of the operators that will transform the initial state to the goal state. The test case generator enables efficient application of planning by using the hierarchical model of the GUI. High-level planning operators are developed that represent the events in a component.

The test designer identifies typical tasks (scenarios) represented by initial and goal states. The planner then generates plans representing sequences of GUI interactions that a user might employ to reach the goal state from the initial state. These plans are used as test cases for the GUI.

4. **Test oracle:** A GUI test oracle determines whether a GUI behaves as expected for a given test case. The oracle uses the GUI representation and for every test case, automatically derives the expected state for every event in the test case. The actual state of an executing GUI is also represented in terms of objects and their properties derived from the GUI's execution. Using the actual state acquired from an *execution monitor*, the oracle automatically compares the expected and actual states after each event to verify the correctness of the GUI for the test case.
5. **Test executor:** Test cases, generated by the test case generator, are input to the test executor that executes each event in the test case, such as mouse and keyboard events, thereby mimicking a GUI user.
6. **Regression tester:** The regression tester partitions the original GUI test cases into valid test cases that represent correct input/output for the modified GUI and invalid test cases that no longer represent correct input/output. Valid test cases are not rerun on the modified GUI since they execute the same sequences of events already tested on the original GUI. On the other hand, invalid test cases cannot be rerun because they either specify incorrect input or incorrect expected output. The regression tester reuses some of the invalid test cases by repairing them. The key idea is that the repaired test cases are more likely to reveal faults in the modified GUI since they test specific sequences of events that were modified in the GUI. Invalid test cases are repaired by the application of repairing transformations that employ the specifications of the GUI to make the repairs. The regression tester employs the event-flow graphs and integration tree of the original and the modified GUI to determine the changes made to the GUI, identify invalid test cases, and repair them.

A cursory exploration of extending the framework to handle the new testing requirements of web-user interfaces (WUIs) was also done. The WUI is modeled in terms of its constituent *objects*, *properties* of these objects, and a set of *constraints* (geometric, temporal, etc.) among the objects. *Environment conditions* represent the characteristics of the states of the client, server, and network that effect the behavior of the WUI. A *WUI test case* is defined as a sequence of events with temporal/synchronization constraints associated with each event. Test cases can be generated in two phases: (1) a plan generation technique (similar to the one used for GUIs) generates the event sequences, and (2) the test designer annotates the event sequences with temporal/synchronization constraints. The environment conditions for the WUI are obtained by employing the *category-partition* method. The test designer partitions the characteristics of the states of the client, server, and network into *categories* (browser, operating system, etc.), which are further partitioned into *choices* (e.g., Netscape, Internet explorer for *browser* and Windows NT, Windows 2000, Linux for *operating system*). The test designer assigns a priority, a real number between 0 and 1, to each choice within each category. This priority is then used to order test case execution with appropriate environmental conditions.

9.2 Future Work

Several new questions were raised while conducting this research and performing experiments. New problem domains that could benefit from some of the developed techniques were also identified. These questions and identified domains are the basis for future research that can be conducted using the ideas developed in this dissertation. Some ideas are outlined below:

1. **Relationship between the interface and underlying code:** Software contains both the interface and the underlying code. Yet, different testing paradigms are used to test the interface and the underlying code. Test cases executed on the interface cause a path to be executed in the control-flow graph of the underlying code. It may be redundant and expensive to retest these paths when testing the underlying code. A unified theory between testing the interface and the underlying code may be useful in reducing testing costs.
2. **Separating the GUI logic from the underlying logic:** The running example used throughout the dissertation was a new implementation of the WordPad software. WordPad was chosen because it was possible to encode is underlying code's logic directly in GUI operators. However, encoding the underlying logic of a more complex software may make the operator definitions bulky. Techniques need to be developed that can separate the GUI's functionality from that of the underlying code.
3. **GUI specifications and testing:** As is the case with all software, a GUI's specifications are developed before it is implemented. The GUI implementer employs these specifications to implement the GUI. The test designer uses the same specifications to test the GUI. However, writing the specifications (usually written in natural language), realizing them as programs and using them to generate test cases is error-prone. If, however, the GUI specifications were executable, it might be possible for a

GUI designer to formally specify the design of the GUI as executable specifications, debug these specifications for logical correctness using automated tools (such as model checkers), and use a GUI generator to automatically generate the GUI implementation. The same specifications could then be used to test the GUI. Developing these GUI specifications remains an open research issue. One promising starting point is to specify GUIs in terms of operators. Preconditions and effects have been used in the past to specify GUIs [25].

The same design/implementation/testing paradigm can also be extended to other software. For example, the paradigm may be applied to developing device drivers. The device developer may provide formal specifications for the device. Device drivers may be automatically generated for different operating systems and then tested.

4. **Prioritizing GUI test cases:** Experiments showed that it is impractical to test the GUI for all event-sequences. A subset of "important" event sequences needs to be identified, generated and executed. Identifying such important sequences requires that they be ordered by assigning a *priority* to each event sequence. Detailed experiments need to be conducted to determine the error detection capability of these high-priority test cases.
5. **Non-deterministic GUIs and probabilistic input devices:** The output of several types of software (such as games) and input devices (such as virtual reality gloves) is non-deterministic. A probabilistic model of the software/hardware may be created to generate testing information.
6. **Repairing test cases for regression testing of conventional software:** This dissertation presented a new technique to perform regression testing by repairing invalid test cases. Modification of conventional software also results in invalid test cases that are simply discarded. Studies need to be conducted to determine whether the repairing technique developed in this dissertation can be extended to repair invalid test cases for conventional software.
7. **Exploring the correlation between event-based and code-based coverage criteria:** One experiment showed an interesting correlation between event-coverage and statement coverage of the underlying code. Additional experiments need to be conducted to determine whether such a correlation exists between event-coverage and other code-based coverage criteria.
8. **Object-oriented and component-based software:** Modern software development is an engineering effort where a software developer composes software by reusing classes, objects, and components. However, these development paradigms create new

challenges for testing. Source code from certain classes may not be available to the test designer. In such cases, code-based testing may not be applicable. An interface-based technique similar to the one used for GUI testing may be beneficial.

9. **Reactive software:** Reactive software is finding increasing importance in embedded and safety critical systems. To create an oracle for testing, the test designer manually specifies a set of conditions that must be met during software execution. This manual specification is prone to incompleteness. More comprehensive checking may be achieved if the software's reactive components are modeled in the form of preconditions and effects and the test oracle is automatically generated.
10. **Networks:** A network consists of a collection of heterogenous elements such as links and switches. Each element is responsible for routing traffic through the network. Testing a network is a complex process where each element of the network plays an important role in determining the correctness of its state. A network can be modeled in terms of its elements (as objects) and their state (as a set of properties). Messages passing through the network may be modeled as events that change the state of the network's elements. Such a model can then be used to test the network.
11. **Execution profiles and testing:** Conventional testing techniques focus on employing results of the software's static analyses and specifications to generate testing information. However, run-time (dynamic) information in the form of execution profiles of the software, may be especially valuable for testing the software. Techniques have been studied to collect this data [60]. It may be beneficial to use execution profiles to generate test cases that test frequently-used paths in the software's control-flow graph.

Bibliography

Bibliography

[1] Agrawal, H., Horgan, J. R., Krauser, E. W., and London, S. A. Incremental regression testing. In *Proceedings of the Conference on Software Maintenance* (Washington, Sept. 1993), D. Card, Ed., IEEE Computer Society Press, pp. 348–357.

[2] Arnold, K., Gosling, J., and Holmes, D. *The Java Programming Language Third Edition*, third ed. Addison-Wesley, Reading, MA, 2000.

[3] Avritzer, A., and Weyuker, E. J. The automatic generation of load test suites and the assessment of the resulting software. *IEEE Transactions on Software Engineering 21*, 9 (Sept. 1995), 705–716.

[4] Ball, T. On the limit of control flow analysis for regression test selection. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA-98)* (New York, Mar.2–5 1998), vol. 23,2 of *ACM Software Engineering Notes*, ACM Press, pp. 134–142.

[5] Baran, N. Load testing Web sites. *Dr. Dobb's Journal of Software Tools 26*, 3 (Mar. 2001), 112, 114, 116, 118–119.

[6] Beizer, B. *Software Testing Techniques*, 2nd ed. Van Nostrand Reinhold, New York, 1990.

[7] Benedusi, P., Cimitile, A., and DeCarlini, U. Post-maintenance testing based on path change analysis. In *Proceedings of the IEEE Conference on Software Maintenance* (1988), pp. 352–368.

[8] Bernhard, P. J. A reduced test suite for protocol conformance testing. *ACM Transactions on Software Engineering and Methodology 3*, 3 (July 1994), 201–220.

[9] Binkley, D. Reducing the cost of regression testing by semantics guided test case selection. In *Proceedings of the International Conference on Software Maintenance* (Washington, Oct.17–20 1995), G. Caldiera and K. Bennett, Eds., IEEE Computer Society Press, pp. 251–263.

[10] Binkley, D. Semantics guided regression test cost reduction. *IEEE Transactions on Software Engineering 23*, 8 (Aug. 1997), 498–516.

[11] Blum, A. L., and Furst, M. L. Fast planning through planning graph analysis. *Artificial Intelligence 90*, 1–2 (1997), 279–298.

[12] Chays, D., Dan, S., Frankl, P. G., Vokolos, F. I., and Weyuker, E. J. A framework for testing database applications. In *Proceedings of the 2000 International Symposium on Software Testing and Analysis (ISSTA)* (2000), pp. 147–157.

[13] Chow, T. S. Testing software design modeled by finite-state machines. *IEEE trans. on Software Engineering SE-4, 3* (1978), 178–187.

[14] Clarke, J. M. Automated test generation from a behavioral model. In *Proceedings of Pacific Northwest Software Quality Conference* (May 1998), IEEE Press.

[15] Dillon, L. K., and Ramakrishna, Y. S. Generating oracles from your favorite temporal logic specifications. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering* (New York, Oct.16–18 1996), vol. 21 of *ACM Software Engineering Notes*, ACM Press, pp. 106–117.

[16] Dillon, L. K., and Yu, Q. Oracles for checking temporal properties of concurrent systems. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering* (Dec. 1994), pp. 140–153.

[17] Donat, M. Automating Formal Specification Based Testing. In *Proc. Conf. on Theory and Practice of Software Development (TAPSOFT 97)* (Lille, France, 1997), M. Bidoit and M. Dauchet, Eds., vol. 1214 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, pp. 833–847.

[18] du Bousquet, L., Ouabdesselam, F., Richier, J.-L., and Zuanon, N. Lutess: a specification-driven testing environment for synchronous software. In *Proceedings of the 21st International Conference on Software Engineering* (May 1999), ACM Press, pp. 267–276.

[19] Erol, K., Hendler, J., and Nau, D. S. HTN planning: Complexity and expressivity. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)* (Seattle, Washington, USA, Aug. 1994), vol. 2, AAAI Press/MIT Press, pp. 1123–1128.

[20] Erol, K., Nau, D., and Hendler, J. Toward a general framework for hierarchical task-network planning. In *Foundations of Automatic Planning: The Classical Approach and Beyond: Papers from the 1993 AAAI Spring Symposium* (1993), AAAI Press, Menlo Park, California, pp. 20–23.

[21] Esmelioglu, S., and Apfelbaum, L. Automated test generation, execution, and reporting. In *Proceedings of Pacific Northwest Software Quality Conference* (Oct 1997), IEEE Press.

[22] Fikes, R., and Nilsson, N. strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence 2* (1971), 189–208.

[23] Fogel, L. J., Owens, A. J., and Walsh, M. J. Artificial intelligence through a simulation of evolution. In *Biophysics and Cybernetic Systems: Proc. of the 2nd Cybernetic Sciences Symposium* (Washington, D.C., 1965), M. Maxfield, A. Callahan, and L. J. Fogel, Eds., Spartan Books, pp. 131–155.

[24] Fogel, L. J., Owens, A. J., and Walsh, M. J. *Artificial Intelligence through Simulated Evolution.* John Wiley & Sons, New York, 1966.

[25] Gieskens, D. F., and Foley, J. D. Controlling user interface objects through pre- and postconditions. In *Proceedings of ACM CHI'92 Conference on Human Factors in Computing Systems* (1992), Tools and Techniques, pp. 189–194.

[26] Gomes, C. P., Selman, B., McAloon, K., and Tretkoff, C. Randomization in backtrack search: Exploiting heavy-tailed profiles for solving hard scheduling problems. In *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems* (Carnegie Mellon University, Pittsburgh, PA, June 1998), R. Simmons, M. Veloso, and S. Smith, Eds., AAAI Press, pp. 208–213.

[27] Goodenough, J. B., and Gerhart, S. L. Toward a theory of test data selection. *ACM SIGPLAN Notices 10*, 6 (June 1975), 493–493.

[28] Gosling, J., Joy, B., Steele, G., and Bracha, G. *The Java Language Specification Second Edition.* The Java Series. Addison-Wesley, Boston, Mass., 2000.

[29] Gourlay, J. S. A mathematical framework for the investigation of testing. *IEEE Transactions on Software Engineering 9*, 6 (Nov. 1983), 686–709.

[30] Gray, J. What next? a few remaining IT problems. Jim Gray received the 1998 ACM Turing Award at the ACM awards banquet in NYC on April 15. His Turing award lecture: What Next? A few remaining IT Problems was presented at the ACM Federated Research Computer Conference in Atlanta, Georgia, on 4 May 1999. A refined version of it will be presented at the SIGMOD conference in Philadelphia in June.

[31] H. Cho, G.D. Hachtel, and F. Somenzi. Redundancy identification/removal and test generation for sequential circuits using implicit state enumeration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 12*, 7 (July 1993), 935–945.

[32] Hammontree, M. L., Hendrickson, J. J., and Hensley, B. W. Integrated data capture and analysis tools for research and testing an graphical user interfaces. In *Proceedings of the Conference on Human Factors in Computing Systems* (New York, NY, USA, May 1992), ACM Press, pp. 431–432.

[33] Harrold, M. J., Gupta, R., and Soffa, M. L. A methodology for controlling the size of a test suite. *acm Transactions of Software Engineering and Methodology 2*, 3 (July 1993), 270–285.

[34] Harrold, M. J., and Soffa, M. L. Interprocedual data flow testing. In *Proceedings of the ACM SIGSOFT '89 Third Symposium on Testing, Analysis, and Verification (TAV3)* (1989), R. A. Kemmerer, Ed., pp. 158–167.

[35] Howe, A., von Mayrhauser, A., and Mraz, R. T. Test case generation as an AI planning problem. *Automated Software Engineering 4* (1997), 77–106.

[36] Jagadeesan, L. J., Porter, A., Puchol, C., Ramming, J. C., and Votta, L. G. Specification-based testing of reactive software: Tools and experiments. In *Proceedings of the 19th International Conference on Software Engineering (ICSE '97)* (Berlin - Heidelberg - New York, May 1997), Springer, pp. 525–537.

[37] Jónsson, A. K., and Ginsberg, M. L. Procedural reasoning in constraint satisfaction. In *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning* (San Francisco, Nov. 5–8 1996), L. C. Aiello, J. Doyle, and S. Shapiro, Eds., Morgan Kaufmann, pp. 160–173.

[38] Kasik, D. J., and George, H. G. Toward automatic generation of novice user test scripts. In *Proceedings of the Conference on Human Factors in Computing Systems : Common Ground* (New York, 13–18 Apr. 1996), ACM Press, pp. 244–251.

[39] Kautz, H., and Selman, B. Planning as satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence* (Vienna, Austria, Aug. 1992), B. Neumann, Ed., John Wiley & Sons, pp. 359–363.

[40] Kautz, H., and Selman, B. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)* (Portland, Oregon, USA, Aug. 1996), AAAI Press / The MIT Press, pp. 1202–1207.

[41] Kautz, H., and Selman, B. Blackbox: A new approach to the application of theorem proving to problem solving. In *AIPS-98 Workshop on Planning as Combinatorial Search* (Pittsburgh, PA, USA, June 1998), AAAI Press / The MIT Press.

[42] Kautz, H., and Selman, B. The role of domain-specific knowledge in the planning as satisfiability framework. In *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems* (Carnegie Mellon University, Pittsburgh, PA, June 1998), R. Simmons, M. Veloso, and S. Smith, Eds., AAAI Press, pp. 181–189.

[43] Kepple, L. R. The black art of GUI testing. *Dr. Dobb's Journal of Software Tools 19*, 2 (Feb. 1994), 40.

[44] Kirda, E. Web engineering device independent web sevices. In *Proceedings of the 23rd International Conference on Software Engineering, Doctoral Symposium* (Toronto, Canada, May 2001).

[45] Koehler, J., Nebel, B., Hoffman, J., and Dimopoulos, Y. Extending planning graphs to an ADL subset. *Lecture Notes in Computer Science 1348* (1997), 273.

[46] Kranakis, E., Krizanc, D., Pelc, A., and Peleg, D. The complexity of data mining on the web. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC '96)* (New York, USA, May 1996), ACM, pp. 153–153.

[47] Kung, D. C., Gao, J., Hsia, P., Toyoshima, Y., and Chen, C. On regression testing of object-oriented programs. *The Journal of Systems and Software 32*, 1 (Jan. 1996), 21–31.

[48] Lifschitz, V. On the semantics of STRIPS. In *Reasoning about Actions and Plans: Proceedings of the 1986 Workshop* (Timberline, Oregon, June-July 1986), M. P. Georgeff and A. L. Lansky, Eds., Morgan Kaufmann, pp. 1–9.

[49] Mahajan, R., and Shneiderman, B. Visual & textual consistency checking tools for graphical user interfaces. Technical Report CS-TR-3639, University of Maryland, College Park, May 1996.

[50] McCarthy, J. Situations, actions, and causal laws. Memo 2, Stanford University Artificial Intelligence Project, Stanford, California, 1963.

[51] Memon, A. M., Pollack, M., and Soffa, M. L. Comparing causal-link and propositional planners: Tradeoffs between plan length and domain size. Technical Report 99-06, University of Pittsburgh, Pittsburgh, Feb. 1999.

[52] Myers, B. A. *State of the Art in User Interface Software Tools*, vol. 4. Ablex Publishing, 1993, ch. pp110-150.

[53] Myers, B. A. Why are human-computer interfaces difficult to design and implement? Technical Report CS-93-183, Carnegie Mellon University, School of Computer Science, July 1993.

[54] Myers, B. A. User interface software tools. *ACM Transactions on Computer-Human Interaction 2*, 1 (1995), 64–103.

[55] Myers, B. A., Hollan, J. D., and Cruz, I. F. Strategic directions in human-computer interaction. *ACM Computing Surveys 28*, 4 (Dec. 1996), 794–809.

[56] Myers, B. A., and Olsen, Jr., D. R. User interface tools. In *Proceedings of ACM CHI'94 Conference on Human Factors in Computing Systems* (1994), vol. 2 of *TUTORIALS*, pp. 421–422.

[57] Myers, B. A., Olsen, Jr., D. R., and Bonar, J. G. User interface tools. In *Proceedings of ACM INTERCHI'93 Conference on Human Factors in Computing Systems – Adjunct Proceedings* (1993), Tutorials, p. 239.

[58] Ostrand, T., Anodide, A., Foster, H., and Goradia, T. A visual test development environment for GUI systems. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA-98)* (New York, Mar.2–5 1998), ACM Press, pp. 82–92.

[59] Ostrand, T. J., and Balcer, M. J. The category-partition method for specifying and generating functional tests. *Communications of the ACM, CACM 31*, 6 (June 1988), 676–686.

[60] Pavlopoulou, C., and Young, M. Residual test coverage monitoring. In *Proceedings of the 1999 International Conference on Software Engineering* (1999), IEEE Computer Society Press / ACM Press, pp. 277–284.

[61] Pednault, E. *Toward a Mathematical Theory of Plan Synthesis*. PhD thesis, Dept of Electrical Engineering, Stanford University, Stanford, CA, Dec. 1986.

[62] Pednault, E. P. D. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proceedings of KR'89* (Toronto, Canada, pp 324-331, May 1989).

[63] Penberthy, J. S., and Weld, D. S. UCPOP: A sound, complete, partial order planner for ADL. In *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning* (Cambridge, MA, Oct. 1992), W. Nebel, Bernhard; Rich, Charles; Swartout, Ed., Morgan Kaufmann, pp. 103–114.

[64] Perry, W. *Effective Methods for Software Testing*. John Wiley & Sons, Inc., New York, N.Y., 1995.

[65] Peters, D., and Parnas, D. L. Generating a test oracle from program documentation. In *Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA)* (1994), T. Ostrand, Ed., pp. 58–65.

[66] Pollack, M. E., Joslin, D., and Paolucci, M. Flaw selection strategies for partial-order planning. *Journal of Artificial Intelligence Research 6*, 6 (1997), 223–262.

[67] Pressman, R. S. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, 1994.

[68] Rapps, S., and Weyuker, E. J. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering 11*, 4 (Apr. 1985), 367–375.

[69] Richardson, D. J. TAOS: Testing with analysis and oracle support. In *Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA): August 17–19, 1994, Seattle, Washington, USA* (New York, NY 10036, USA, 1994), T. Ostrand, Ed., ACM Sigsoft, ACM Press, pp. 138–153.

[70] Richardson, D. J., Leif-Aha, S., and OMalley, T. O. Specification-based Test Oracles for Reactive Systems. In *Proceedings of the 14th International Conference on Software Engineering* (May 1992), pp. 105–118.

[71] Rosenblum, D., and Rothermel, G. A comparative study of regression test selection techniques. In *Proceedings of the IEEE Computer Society 2nd International Workshop on Empirical Studies of Software maintenance* (Oct. 1997), pp. 89–94.

[72] Rosenblum, D. S., and Weyuker, E. J. Predicting the cost-effectiveness of regression testing strategies. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering* (New York, Oct.16–18 1996), vol. 21 of *ACM Software Engineering Notes*, ACM Press, pp. 118–126.

[73] Rosenblum, D. S., and Weyuker, E. J. Using coverage information to predict the cost-effectiveness of regression testing strategies. *IEEE Transactions on Software Engineering 23*, 3 (Mar. 1997), 146–156.

[74] Rothermel, G., and Harrold, M. J. A safe, efficient algorithm for regression test selection. In *Proceedings of the Conference on Software Maintenance* (1993), IEEE Computer Society Press, pp. 358–369.

[75] Rothermel, G., and Harrold, M. J. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology 6*, 2 (Apr. 1997), 173–210.

[76] Rothermel, G., and Harrold, M. J. Empirical studies of a safe regression test selection technique. *IEEE Transactions on Software Engineering 24*, 6 (June 1998), 401–419.

[77] Rothermel, G., Harrold, M. J., Ostrin, J., and Hong, C. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Proceedings; International Conference on Software Maintenance* (1998), T. M. Koshgoftaar and K. Bennett, Eds., IEEE Computer Society Press, pp. 34–43.

[78] Schach, S. R. *Software Engineering*, second ed. Richard D. Irwin/Aksen Associates, 1993.

[79] Shehady, R. K., and Siewiorek, D. P. A method to automate user interface testing using variable finite state machines. In *Proceedings of The Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing (FTCS'97)* (Washington - Brussels - Tokyo, June 1997), IEEE Press, pp. 80–88.

[80] Siepman, E., and Newton, A. R. TOBAC: Test Case Browser for Object-Oriented Sofwtare. In *Proc. International Symposium on Software Testing and Analysis* (New York, Aug. 1994), ACM Press, pp. 154–168.

[81] Software Research, I. Testworks for windows ver. 3 - overview. Available from http://www.soft.com/eValid/, 2001.

[82] Su, J., and Ritter, P. R. Experience in testing the Motif interface. *IEEE Software 8*, 2 (Mar. 1991), 26–33.

[83] The, L. Stress Tests For GUI Programs. *Datamation 38*, 18 (Sept. 1992), 37.

[84] Veloso, M., and Stone, P. FLECS: Planning with a flexible commitment strategy. *Journal of Artificial Intelligence Research 3* (June 1995), 25–52.

[85] Vogel, P. An integrated general purpose automated test environment. In *Proceedings of the International Symposium on Software Testing and Analysis* (New York, NY, USA, June 1993), T. Ostrand and E. Weyuker, Eds., ACM Press, pp. 61–69.

[86] Weld, D. S. An introduction to least commitment planning. *AI Magazine 15*, 4 (1994), 27–61.

[87] Weld, D. S. Recent advances in AI planning. *AI Magazine 20*, 1 (Spring 1999), 55–64.

[88] Weyuker, E. J. The applicability of program schema results to programs. *International Journal of Computer and Information Sciences 8*, 5 (Oct. 1979), 387–403.

[89] Weyuker, E. J. Translatability and decidability questions for restricted classes of program schemas. *SIAM Journal on Computing 8*, 4 (1979), 587–598.

[90] White, L. Regression testing of GUI event interactions. In *Proceedings of the International Conference on Software Maintenance* (Washington, Nov.4–8 1996), IEEE Computer Society Press, pp. 350–358.

[91] White, L., and Almezen, H. Generating test cases for GUI responsibilities using complete interaction sequences. In *Proceedings of the International Symposium on Software Reliability Engineering* (Oct. 8–11 2000), pp. 110–121.

[92] Wick, D. T., Shehad, N. M., and Hajare, A. R. Testing the human computer interface for the telerobotic assembly of the space station. In *Proceedings of the Fifth International Conference on Human-Computer Interaction* (1993), vol. 1 of *II. Special Applications*, pp. 213–218.

[93] Wolfram, S. *Mathematica: A System for Doing Mathematics by Computer*. Addison-Wesley, Reading, Massachusetts, 1988.

[94] Wong, A. Y. K., Donkers, A. M., Dillon, R. F., and Tombaugh, J. W. Usability testing: Is the whole test greater than the sum of its parts? In *Proceedings of ACM CHI'92 Conference on Human Factors in Computing Systems – Posters and Short Talks* (1992), Posters: Helping Users, Programmers, and Designers, p. 38.

[95] Young, R. M., Pollack, M. E., and Moore, J. D. Decomposition and causality in partial order planning. In *Second International Conference on Artificial Intelligence and Planning Systems* (1994). Also Technical Report 94-1, Intelligent Systems Program, University of Pittsburgh.

[96] Zhu, H., and Hall, P. Test data adequacy measurements. *Software Engineering Journal 8*, 1 (Jan. 1993), 21–30.

[97] Zhu, H., Hall, P., and May, J. Software unit test coverage and adequacy. *ACM Computing Surveys 29*, 4 (Dec. 1997), 366–427.