Hierarchical GUI Test Case Generation Using Automated Planning

Atif M. Memon, *Student Member*, *IEEE*, Martha E. Pollack, and Mary Lou Soffa, *Member*, *IEEE*

Abstract—The widespread use of GUIs for interacting with software is leading to the construction of more and more complex GUIs. With the growing complexity come challenges in testing the correctness of a GUI and its underlying software. We present a new technique to automatically generate test cases for GUIs that exploits planning, a well-developed and used technique in 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. Our test case generation technique enables efficient application of planning by first creating a hierarchical model of a GUI based on its structure. The GUI model consists of hierarchical planning operators representing the possible events in the GUI. The test designer defines the preconditions and effects of the hierarchical operators, which are input into a plan-generation system. The test designer also creates scenarios that represent typical initial and goal states for a GUI user. The planner then generates plans representing sequences of GUI interactions that a user might employ to reach the goal state from the initial state. We implemented our test case generation system, called Planning Assisted Tester for grapHical user interface Systems (PATHS) and experimentally evaluated its practicality and effectiveness. We describe a prototype implementation of PATHS and report on the results of controlled experiments to generate test cases for Microsoft's WordPad.

Index Terms—Software testing, GUI testing, application of AI planning, GUI regression testing, automated test case generation, generating alternative plans.

1 Introduction

Graphical User Interfaces (GUIs) have become an important and accepted way of interacting with today's software. Although they make software easy to use from a user's perspective, they complicate the software development process [1], [2]. In particular, testing GUIs is more complex than testing conventional software, for not only does the underlying software have to be tested but the GUI itself must be exercised and tested to check whether it confirms to the GUI's specifications. Even when tools are used to generate GUIs automatically [3], [4], [5], these tools themselves may contain errors that may manifest themselves in the generated GUI leading to software failures. Hence, testing of GUIs continues to remain an important aspect of software testing.

Testing the correctness of a GUI is difficult for a number of reasons. First of all, the space of possible interactions with a GUI is enormous, in that each sequence of GUI commands can result in a different state and a GUI command may need to be evaluated in all of these states. The large number of possible states results in a large number of input permutations [6] requiring extensive testing, e.g., Microsoft released almost 400,000 beta copies of Windows95 targeted at finding program failures [7]. Another problem relates to determining the coverage of a set of test cases. For conventional software, coverage is measured using the amount and type of underlying code exercised. These measures do not work well for GUI testing, because what matters is not only how much of the code is tested, but in how many different possible states of the software each piece of code is tested. An important aspect of GUI testing is verification of its state at each step of test case execution. An incorrect GUI state can lead to an unexpected screen, making further execution of the test case useless since events in the test case may not match the corresponding GUI components on the screen. Thus, the execution of the test case must be terminated as soon as an error is detected. Also, if verification checks are not inserted at each step, it may become difficult to identify the actual cause of the error. Finally, regression testing presents special challenges for GUIs, because the input-output mapping does not remain constant across successive versions of the software [1]. Regression testing is especially important for GUIs since GUI development typically uses a rapid prototyping model [8], [9], [10], [11].

An important component of testing is the generation of test cases. Manual creation of test cases and their maintenance, evaluation, and conformance to coverage criteria are very time consuming. Thus, some automation is necessary when testing GUIs. In this paper, we present a new technique to automatically generate test cases for GUI systems. Our approach exploits planning techniques developed and used extensively in artificial intelligence (AI). The key idea 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 the user might employ to achieve them. Our test case generation system, called Planning Assisted Tester for grapHical user interface Systems

- *The authors are with the Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260.*
  E-mail: {atif, pollack, soffa}@cs.pitt.edu.
- *M.E. Pollack is also with the Intelligent Systems Program.*

Manuscript received 15 Nov. 1999; revised 10 Apr. 2000; accepted 1 May 2000.
Recommended for acceptance by D. Garlan.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 112144.

(PATHS) takes these goals as input and generates such sequences of events automatically. These sequences of events or "plans" become test cases for the GUI. PATHS first performs an automated analysis of the hierarchical structure of the GUI to create hierarchical operators that are then used during plan generation. The test designer describes the preconditions and effects of these planning operators, which are subsequently input to the planner. Hierarchical operators enable the use of an efficient form of planning. Specifically, to generate test cases, a set of initial and goal states is input into the planning system; it then performs a restricted form of hierarchical plan generation to produce multiple hierarchical plans. We have implemented PATHS and we demonstrate its effectiveness and efficiency through a set of experiments.

The important contributions of the method presented in this paper include the following:

- We make innovative use of a well-known and used technique in AI, which has been shown to be capable of solving problems with large state spaces [12]. Combining the unique properties of GUIs and planning, we are able to demonstrate the practicality of automatically generating test cases using planning.
- Our technique exploits structural features present in GUIs to reduce the model size, complexity, and to improve the efficiency of test case generation.
- Exploiting the structure of the GUI and using hierarchical planning makes regression testing easier. Changes made to one part of the GUI do not affect the entire test suite. Most of our generated test cases are updated by making local changes.
- Platform specific details are incorporated at the very end of the test case generation process, increasing the portability of the test suite. Portability, which is important for GUI testing [13], assures that test cases written for GUI systems on one platform also work on other platforms.
- Our technique allows reuse of operator definitions that commonly appear across GUIs. These definitions can be maintained in a library and reused to generate test cases for subsequent GUIs.

The next section gives a brief overview of PATHS using an example GUI. Section 3 briefly reviews the fundamentals of AI plan generation. Section 4 describes how planning is applied to the GUI test case generation problem. In Section 5, we describe a prototype system for PATHS and give timing results for generating test cases. We discuss related work for automated test case generation for GUIs in Section 6 and conclude in Section 7.

2 Overview

In this section, we present an overview of PATHS through an example. The goal is to provide the reader with a high-level overview of the operation of PATHS and highlight the role of the test designer in the overall test case generation process. Details about the algorithms used by PATHS are given in Section 4.

GUIs typically consist of components such as labels, buttons, menus, and pop-up lists. The GUI user interacts

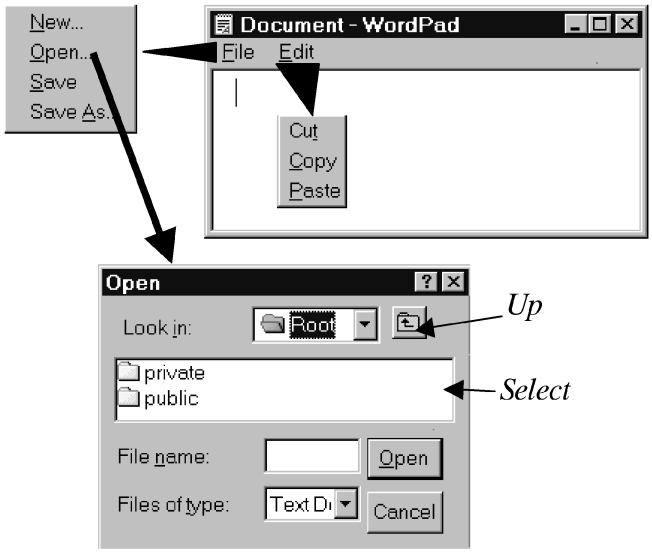


Fig. 1. The example GUI.

with these components, which in turn generate events. For example, pushing a button `Preferences` generates an event (called the `Preferences event`) that opens a window. In addition to these visible components on the screen, the user also generates events by using devices such as a mouse or a keyboard. For the purpose of our model, GUIs have two types of windows: *GUI windows* and *object windows*. GUI windows contain GUI components, whereas object windows do not contain any GUI components. Object windows are used to display and manipulate objects, e.g., the window used to display text in MS WordPad.

Fig. 1 presents a small part of the MS WordPad's GUI. This GUI can be used for loading text from files, manipulating the text (by cutting and pasting), and then saving the text in another file. At the highest level, the GUI has a pull-down menu with two options (`File` and `Edit`) that can generate events to make other components available. For example, the `File` event opens a menu with `New`, `Open`, `Save`, and `SaveAs` options. The `Edit` event opens a menu with `Cut`, `Copy`, and `Paste` options, which are used to cut, copy, and paste objects, respectively, from the main screen. The `Open` and `SaveAs` events open windows with several more components. (Only the `Open` window is shown; the `SaveAs` window is similar.) These components are used to traverse the directory hierarchy and select a file. `Up` moves up one level in the directory hierarchy and `Select` is used to either enter subdirectories or select files. The window is closed by selecting either `Open` or `Cancel`.

The central feature of PATHS is a plan generation system. Automated plan generation has been widely investigated and used within the field of artificial intelligence. The input to the planner is an initial state, a goal state, and a set of operators that are applied to a set of objects. Operators, which model events, are usually described in terms of preconditions and effects: conditions that must be true for the action to be performed and conditions that will be true after the action is performed. A solution to a given planning problem is a sequence of

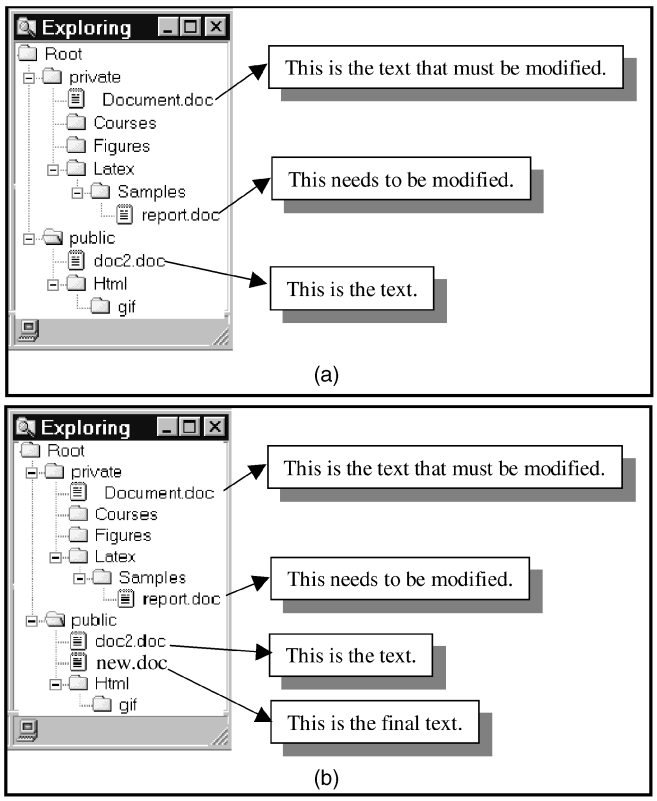


Fig. 2. A task for the planning system. (a) The initial state and (b) the goal state.

instantiated operators that is guaranteed to result in the goal state when executed in the initial state.[1] In our example GUI, the operators relate to GUI events.

Consider Fig. 2a, which shows a collection of files stored in a directory hierarchy. The contents of the files are shown in boxes and the directory structure is shown as 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, we can define a goal of creating the new document shown in Fig. 2b and then storing it in file *new.doc* in the `/root/public` directory. Fig. 2b shows that this `goal state` 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`, or by loading file `doc2.doc` and inserting text, or by creating the document from scratch by typing in the text.

Our test case generation process is partitioned into two phases, the *setup* phase and *plan-generation* phase. In the first step of the setup phase, PATHS creates a hierarchical model of the GUI and returns a list of operators from the model to the test designer. By using knowledge of the GUI, the test designer then defines the preconditions and effects of the operators in a simple language provided by the planning system. 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 scenarios. The test designer can iterate through the plan-generation phase any number of times, defining more scenarios and generating more test cases. Table 1 summarizes the tasks assigned to the test designer and those automatically performed by PATHS.

1. We have described only the simplest case of AI planning. The literature includes many techniques for extensions, such as planning under uncertainty [14], but we do not consider these techniques in this paper.

TABLE 1
Roles of the Test Designer and PATHS during Test Case Generation

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

For our example GUI, the simplest approach in Step 1 would be for PATHS to identify one operator for each GUI event (e.g., `Open`, `File`, `Cut`, `Paste`). (As a naming convention, we disambiguate with meaningful prefixes whenever names are the same, such as `Up`.) The test designer would then define the preconditions and effects for all the events shown in Fig. 3a. Although conceptually simple, this approach is inefficient for generating test cases for GUIs as it results in a large number of operators. Many of these events (e.g., `File` and `Edit`) merely make other events possible, but do not interact with the underlying software.

An alternative modeling scheme, and the one used in this work, models the domain hierarchically with high-level operators that decompose into sequences of lower level ones. Although high-level operators could in principle be developed manually by the test designer, PATHS avoids this inconvenience by automatically performing the abstraction. More specifically, PATHS begins the modeling process by partitioning the GUI events into several classes. The details of this partitioning scheme are discussed later in Section 4. The event classes are then used by PATHS to create two types of planning operators—*system-interaction operators* and *abstract operators*.

GUI Events = {File, Edit,
New, Open, Save, SaveAs,
Cut, Copy, Paste,
Open.Up, Open.Select, Open.Cancel, Open.Open,
SaveAs.Up, SaveAs.Select, SaveAs.Cancel, SaveAs.Save}.

(a)

Planning Operators = {
File_New, File_Open, File_Save, File_SaveAs,
Edit_Cut, Edit_Copy, Edit_Paste}.

(b)

Fig. 3. The example GUI: (a) original GUI events and (b) planning operators derived by PATHS.

TABLE 2
Operator-Event Mappings for the Example GUI

Operator Name	Operator Type	GUI Events
FILE_NEW	Sys. Interaction	<File, New>
FILE_OPEN	Abstract	<File, Open>
FILE_SAVE	Sys. Interaction	<File, Save>
FILE_SAVEAS	Abstract	<File, SaveAs>
EDIT_CUT	Sys. Interaction	<Edit, Cut>
EDIT_COPY	Sys. Interaction	<Edit, Copy>
EDIT_PASTE	Sys. Interaction	<Edit, Paste>

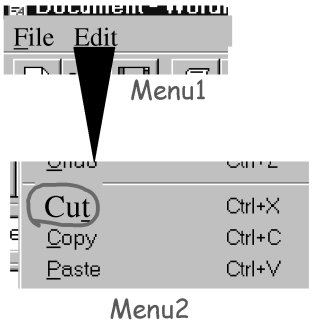
The system-interaction operators are derived from those GUI events that generate interactions with the underlying software. For example, PATHS defines a system-interaction operator `EDIT_CUT` that *cuts* text from the example GUI's window. Examples of other system-interaction operators are `EDIT_PASTE` and `FILE_SAVE`.

The second set of operators generated by PATHS is a set of abstract operators. These will be discussed in more detail in Section 4, but the basic idea is that an abstract operator represents a sequence of GUI events that invoke a window that monopolizes the GUI interaction, restricting the focus of the user to the specific range of events in the window. Abstract operators encapsulate the events of the restricted-focus window by treating the interaction within that window as a separate planning problem. Abstract operators need to be decomposed into lower level operators by an explicit call to the planner. For our example GUI, abstract operators include `File_Open` and `File_SaveAs`.

The result of the first step of the setup phase is that the system-interaction and abstract operators are determined and returned as planning operators to the test designer. The planning operators returned for our example are shown in Fig. 3b.

In order to keep a correspondence between the original GUI events and these high-level operators, PATHS also stores mappings, called *operator-event mappings*, as shown in Table 2. The operator name (column 1) lists all the operators for the example GUI. Operator type (column 2) classifies each operator as either abstract or system-interaction. Associated with each operator is the corresponding sequence of GUI events (column 3).

The test designer then specifies the preconditions and effects for each planning operator. An example of a planning operator, `EDIT_CUT`, is shown in Fig. 4. `EDIT_CUT` is a system-interaction operator. The operator definition contains two parts: preconditions and effects. All the conditions in the preconditions must hold in the GUI before the operator can be applied, e.g., for the user to generate the `Cut` event, at least one object on the screen should be selected (highlighted). The effects of the `Cut` event are that the selected objects are moved to the clipboard and removed from the screen. The language used to define each operator is provided by the planner as an interface to the planning system. Defining the preconditions and effects is not difficult as this knowledge is already built into the GUI structure. For example, the GUI structure requires that `Cut` be made active (visible) only after an object is selected. This is precisely the precondition defined for our example operator (`EDIT_CUT`) in Fig. 4. Definitions



Operator :: EDIT_CUT
Preconditions:
EXISTS Obj in Objects
Selected(Obj).

Effects:
FORALL Obj in Objects
Selected(Obj) ⇒
ADD inClipboard(Obj)
DEL onScreen(Obj)
DEL Selected(Obj).

Fig. 4. An example of a GUI planning operator.

of operators representing events that commonly appear across GUIs, such as `Cut`, can be maintained in a library and reused for subsequent similar applications.

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 Fig. 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 Fig. 5. (Note that `TypeInText()` is an operator representing 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 operator-event mappings stored during the modeling process. One such translation is shown in Fig. 6. This figure shows the abstract 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 subplans at lower levels. These subplans are generated much faster than generating the high-level plan and can be substituted into

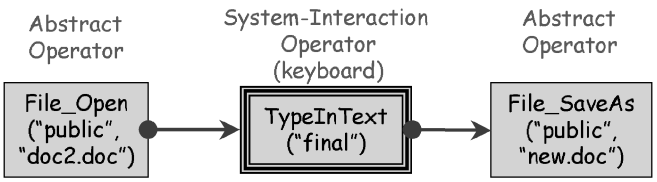


Fig. 5. A plan consisting of abstract operators and a GUI event.

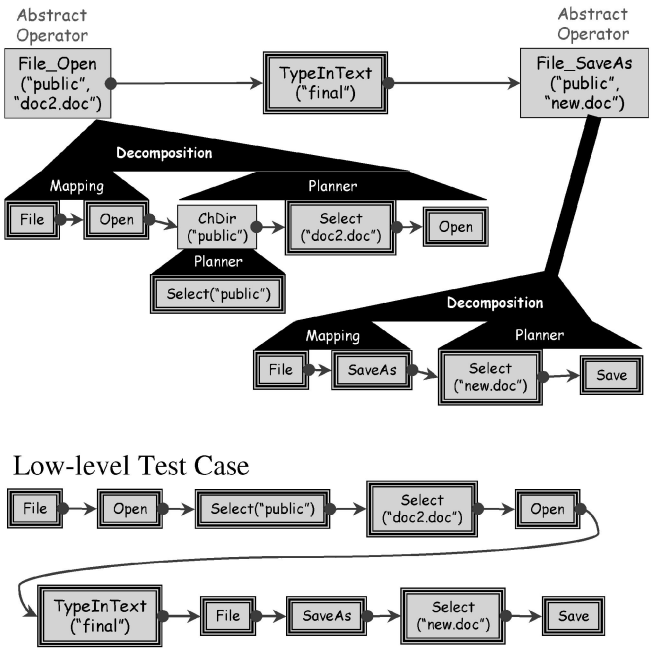


Fig. 6. Expanding the higher level plan.

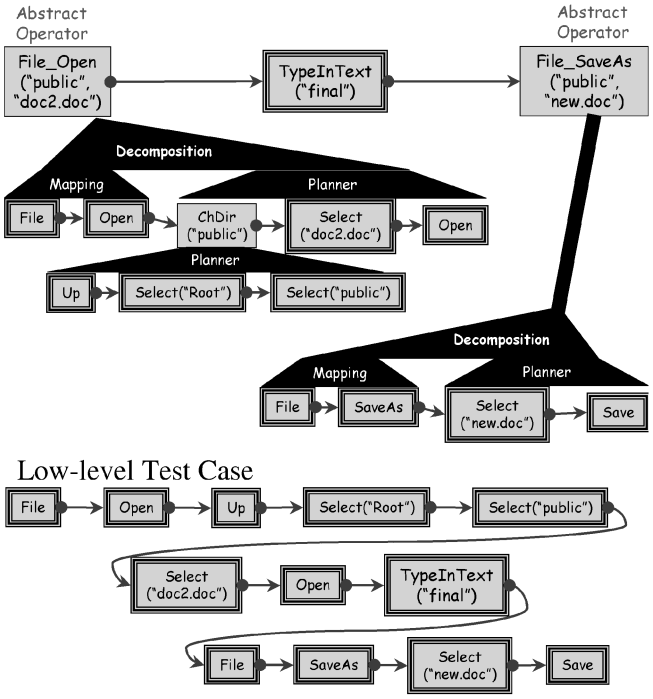


Fig. 7. An alternative expansion leads to a new test case.

the high-level plan to obtain alternative test cases. One such alternative low-level test case generated for the same task is shown in Fig. 7. Note the use of nested invocations to the planner during abstract-operator decomposition.

The hierarchical mechanism aids regression testing since changes made to one component do not necessarily invalidate all test cases. The higher level plans can still be retained and local changes can be made to subplans specific to the changed component of the GUI. Also, the steps in the test cases are platform independent. An additional level of translation is required to generate platform-dependent test cases. By using a high-level model of the GUI, we have the advantage of obtaining platform-independent test cases.

3 Plan Generation

We now provide details on plan generation. Given an initial state, a goal state, a set of operators, and a set of objects, a planner returns a set of steps (instantiated operators) to achieve the goal. Many different algorithms for plan generation have been proposed and developed. Weld presents an introduction to least-commitment planning [15] and a survey of the recent advances in planning technology [16].

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 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 is both an effect of S_i and a precondition for S_j.[2] 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.

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. 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.

Fig. 8a shows the partial-order plan obtained to realize the goal shown in Fig. 2 using our example GUI. In the figure, 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. Fig. 8b 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

2. More generally, c represents a proposition that is the unification of an effect of S_i and a precondition of S_j.

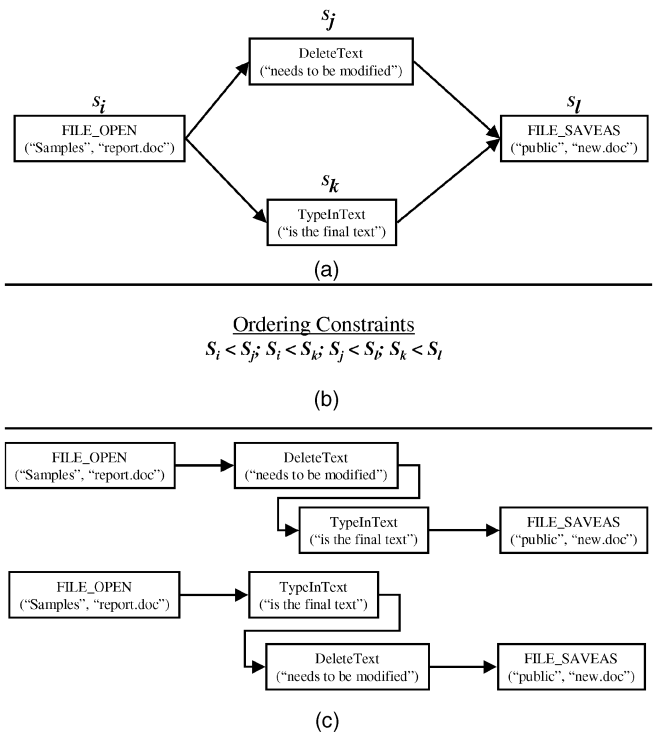


Fig. 8. (a) A partial-order plan, (b) the ordering constraints in the plan, and (c) the two linearizations.

order, but they must precede the `FILE_SAVEAS()` action and must be performed after the `FILE_OPEN()` action. We obtain two legal orders, both of which are shown in Fig. 8c and, thus, two high-level test cases are produced that may be decomposed to yield a number of low-level test cases.

In this work, we employ recently developed planning technology that increases the efficiency of plan generation. Specifically, we generate single-level plans using the Interference Progression Planner (IPP) [17], a system that extends the ideas of the Graphplan system [18] 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 graph. 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 that rely on a first-order logic representation and a graph search requiring unification of unbound variables [18]. IPP uses a standard representation of actions in which preconditions and effects can be parameterized: Subsequent processing performs the conversion to the propositional form.[3] As is common in planning, IPP produces partial-order plans.

IPP forms plans at a single level of abstraction. Techniques have been developed in AI planning to generate plans at multiple levels of abstraction called Hierarchical Task Network (HTN) planning [19]. In HTN planning, domain actions are modeled at different levels of abstraction and, for each operator at level n, one specifies one or more "methods" at level $n-1$. A method is a single-level partial plan and we say that an action "decomposes" into its methods. HTN planning focuses on resolving conflicts among alternative methods of decomposition at each level. The GUI test case generation problem is unusual in that, in our experience at least, it can be modeled with hierarchical plans that do not require conflict resolution during decomposition. Thus, we are able to make use of a restricted form of hierarchical planning, which assumes that all decompositions are compatible. Hierarchical planning is valuable for GUI test case generation as GUIs typically have a large number of components and events and the use of a hierarchy allows us to conceptually decompose the GUI into different levels of abstraction, resulting in greater planning efficiency. As a result of this conceptual shift, plans can be maintained at different abstraction levels. When subsequent modifications are made to the GUI, top-level plans usually do not need to be regenerated from scratch. Instead, only subplans at a lower level of abstraction are affected. These subplans can be regenerated and re-inserted in the larger plans, aiding regression testing.

3. In fact, IPP generalizes Graphplan precisely by increasing the expressive power of its representation language, allowing for conditional and universally quantified effects.

4 Planning GUI Test Cases

Having described AI planning techniques in general, we now present details of how we use planning in PATHS to generate test cases for GUIs.

4.1 Developing a Representation of the GUI and Its Operations

In developing a planning system for testing GUIs, the first step is to construct an operator set for the planning problem. As discussed in Section 2, the simplest approach of defining one operator for each GUI event is inefficient, resulting in a large number of operators. We exploit certain structural properties of GUIs to construct operators at different levels of abstraction. The operator derivation process begins by partitioning the GUI events into several classes using certain structural properties of GUIs. Note that the classification is based only on the structural properties of GUIs and, thus, can be done automatically by PATHS using a simple depth-first traversal algorithm. The GUI is traversed by clicking on buttons to open menus and windows; for convenience, the names of each operator are taken off the label of each button/menu-item it represents. Note that several commercially available tools also perform such a traversal of the GUI, e.g., WinRunner from Mercury Interactive Corporation.

The classification of GUI events that we employ is as follows:

- *Menu-open events* open menus, i.e., they expand the set of GUI events available to the user. By definition, menu-open events do not interact with the underlying software. The most common example of menu-open events are generated by buttons that open pull-down menus, e.g., `File` and `Edit`.
- *Unrestricted-focus events* open GUI 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 PowerPoint software, the Basic Shapes are displayed in an unrestricted-focus window. For the purpose of test case generation, these events can be treated in exactly the same manner as menu-open events; both are used to expand the set of GUI events available to the user.

- *Restricted-focus events* open GUI windows that have the special property that once invoked, they monopolize the GUI interaction, restricting the focus of the user to a specific range of events within the window, until the window is explicitly terminated. Preference setting is an example of restricted-focus events in many GUI systems; the user clicks on Edit and Preferences, a window opens and the user then spends time modifying the preferences and, finally, explicitly terminates the interaction by either clicking OK or Cancel.
- *System-interaction events* interact with the underlying software to perform some action; common examples include cutting and pasting text and opening object windows.

The above classification of events are then used to create two classes of planning operators.

- *System-interaction operators* represent all sequences of zero or more menu-open and unrestricted-focus events followed by a system-interaction event. Consider a small part of the example 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 while 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 greater 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. 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.[4]

4. 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.

- *Abstract operators* are created from the restricted-focus events. Abstract operators encapsulate the events of the underlying restricted-focus window by creating a new planning problem, the solution to which represents the events a user might generate during the focused interaction. The abstract operators implicitly divide the GUI into several layers of abstraction, so that test cases can be generated for each GUI level, thereby resulting in greater efficiency. The abstract operator is a complex structure since it contains all the necessary components of a planning problem, including the initial and goal states, the set of objects, and the set of operators. The *prefix* of the abstract operator is the sequence of menu-open and unrestricted-focus events that lead to the restricted-focus event. This sequence of events is stored in the operator-event mappings. The *suffix* of the abstract operator represents the restricted-focus user interaction. The abstract operator is decomposed in two steps: 1) using the operator-events mappings to obtain the abstract operator prefix and 2) explicitly calling the planner to obtain the abstract operator suffix. Both the prefix and suffix are then substituted back into the high-level plan. For example, in Fig. 6, the abstract operator FILE_OPEN is decomposed by substituting its prefix (File, Open) using a mapping and suffix (ChDir, Select, Open) by invoking the planner.

  Fig. 9a shows a small part of the example GUI: a File menu with two options, namely Open and SaveAs. When either of these events is generated, it results in another GUI window with more components being made available. The components 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. 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. 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. These properties lead to the following two abstract operators:

  File_Open = <File, Open>, and
  File_SaveAs = <File, SaveAs>.

  In addition to the above two operator-event mappings, an abstract operator definition template is created for each operator as shown in Fig. 9b. This template contains all the essential components of the planning problem, i.e., the set of operators that are available during the restricted-focused user interaction and the initial and goal states, both determined dynamically at the point before the call. Since the higher-level planning problem has already been solved before invoking the planner for the abstract operator, the preconditions and effects of

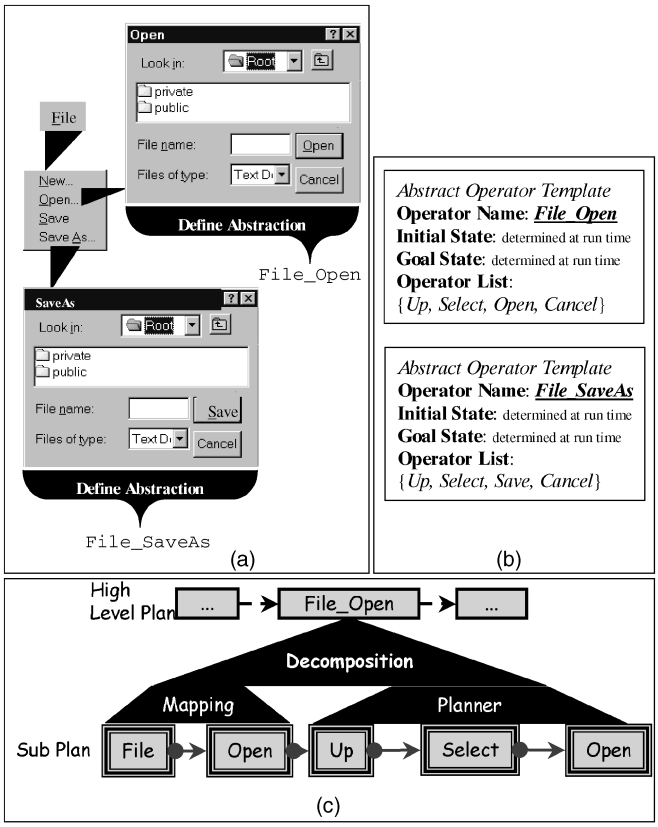


Fig. 9. (a) `Open` and `SaveAs` windows as abstract operators, (b) abstract operator templates, and (c) decomposition of the abstract operator using operator-event mappings and making a separate call to the planner to yield a subplan.

the high-level abstract operator are used to determine the initial and goal states of the subplan. At the highest level of abstraction, the planner will use the high-level operators, i.e., `File_Open` and `File_SaveAs` to construct plans. For example, in Fig. 9c, 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 subplan (`Up`, `Select`, `Open`). `File_Open` is then replaced by the sequence (`File`, `Open`, `Up`, `Select`, `Open`).

The abstract and system-interaction operators are given as input to the planner. The operator set returned for the running example is shown in Fig. 3b.

4.2 Modeling the Initial and Goal State and Generating Test Cases

The test designer begins the generation of particular test cases by identifying a task, consisting of initial and goal states (see Fig. 2). The test designer then codes the initial and goal states or uses a tool that automatically produces the code.[5] The code for the initial state and the changes needed to achieve the goal states is shown in Fig. 10. Once the task has been specified, the system automatically

5. A tool would have to be developed that enables the user to visually describe the GUI's initial and goal states. The tool would then translate the visual representation to code, e.g., the code shown in Fig. 10.

```
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)
…….
```

Fig. 10. Initial State and the changes needed to reach the Goal State.

generates a set of test cases that achieve the goal. The algorithm to generate the test cases is discussed next.

4.3 Algorithm for Generating Test Cases

The test case generation algorithm is shown in Fig. 11. The operators are assumed to be available before making a call to this algorithm, i.e., Steps 1, 2, and 3 of the test case generation process shown in Table 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 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 an abstract 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 subplan, and 3) substituting both these results into the higher level plan. Extraction functions are used to access the planning problem's components at lines 24..27. The lowest level test cases, consisting of GUI events, are returned as a result of the algorithm (line 33).

As noted earlier, one of the main advantages of using the planner in this application is to automatically generate alternative plans for the same goal. Generating alternative

```
                                                                    Lines
Algorithm :: GenTestCases(
  Λ = Operator Set; D = Set of Objects;                              1, 2
  I = Initial State; G = Goal State;                                 3, 4
  T = Threshold) {                                                   5

  planList ← {}; c ← 0;                                              6, 7
   /* Successive calls to the planner (Φ),
    modifying the operators before each call */
  WHILE ((p == Φ(Λ, D, I, G)) != NO_PLAN)                            8
     && (c < T) DO {                                                 9
    InsertInList(p, planList);                                      10
    Λ ← RecordPlan(Λ, p); c++}                                      11, 12

  linearPlans ← {};/* No linear Plans yet */                        13
  /* Linearize all partial order plans */
  FORALL e ∈ planList DO {                                          14
    L ← Linearize(e);                                               15
    InsertInList(L, linearPlans)}                                   16

  testCases ← linearPlans;                                          17
  /* decomposing the testCases */
  FORALL tc ∈ testCases DO {                                        18
    FORALL C ∈ Steps(tc) DO {                                       19
      IF (C == systemInteractionOperator) THEN {                    20
        newC ← lookup(Mappings, C);                                 21
        REPLACE C WITH newC IN tc}                                  22
      ELSEIF (C == abstractOperator) THEN {                         23
        ΛC ← OperatorSet(C); GC ← Goal(C);                          24, 25
        IC ← Initial(C); DC ← ObjectSet(C);                         26, 27
        /* Generate the lower level test cases */
        newC ← APPEND(lookup(Mappings, C),
          GenTestCases(ΛC, DC, IC, GC, T));                         28

        FORALL nc ∈ newC DO {                                       29
          copyOftc ← tc;                                            30
          REPLACE C WITH nc IN copyOftc;                            31
          APPEND copyOftc TO testCases}}}}                          32
  RETURN(testCases)}                                                33
```

Fig. 11. The complete algorithm for generating test cases.

TABLE 3
Some WordPad Plans Generated for the Task of Fig. 2

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")

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 that the first plan found is a high quality plan. In PATHS, we generate alternative plans in the following two ways:

1. Generating multiple linearizations of the partial-order plans. Recall from the earlier discussion 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. We are free to choose any linear order consistent with the partial order. 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 Fig. 8.
2. Repeating the planning process, forcing the planner to generate a different test case at each iteration.

5 Experiments

A prototype of PATHS was developed and several sets of experiments were conducted to ensure that PATHS is practical and useful. These experiments were executed on a Pentium-based computer with 200MB RAM running Linux OS. A summary of the results of some of these experiments is given in the following sections:

5.1 Generating Test Cases for Multiple Tasks

PATHS was used to generate test cases for Microsoft's WordPad. Examples of the generated high-level test cases are shown in Table 3. The total number of GUI events in WordPad was determined to be approximately 325. After analysis, PATHS reduced this set to 32 system-interaction and abstract operators, 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 5.2.

Defining preconditions and effects for the 32 operators was fairly straightforward. The average operator definition required five preconditions and effects, with the most complex operator requiring 10 preconditions and effects. Since mouse and keyboard events are part of the GUI, three additional operators for mouse and keyboard events were defined.

Table 4 presents the CPU time taken to generate test cases for MS 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* subplans, and the fourth column shows the *total* time needed to generate the

TABLE 4
Time Taken to Generate Test Cases for WordPad

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

test case (i.e., the sum of the two previous columns). These results show 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 subplans. These subplans 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 two high-level plans, each of which was decomposed to give several low-level test cases, the shortest of which consisted of 25 GUI events.

The plans shown in Table 3 are at a high level of abstraction. Many changes made to the GUI have no effect on these plans, making regression testing easier and less expensive. For example, none of the plans in Table 3 contain any low-level physical details of the GUI. Changes made to fonts, colors, etc. do not affect the test suite in any way. Changes that modify the functionality of the GUI can also be readily incorporated. For example, if the WordPad GUI is modified to introduce an additional file opening feature, then most of the high-level plans remain the same. Changes are only needed to subplans that are generated by the abstract operator `FILE-OPEN`. Hence, the cost of initial plans is amortized over a large number of test cases.

We also implemented an automated test execution system, 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 our test cases are represented at a high level of abstraction, we translate the high-level actions into physical events. The actual screen coordinates of the buttons, menus, etc. were derived from the layout information.

5.2 Hierarchical vs. Single-Level Test Case Generation

In our second experiment, we compared the single-level test case generation with 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. The operators have a one-to-one correspondence with the GUI events. On the other hand, in the hierarchical test case generation approach, the hierarchical modeling of the operators is used.

TABLE 5
Comparing the Single-Level with the Hierarchical Approach

	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

"-" indicates that no plan was found in one hour.

Results of this experiment are summarized in Table 5. We have shown CPU times for six 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 shows its corresponding CPU time. The same task was then used to generate another test case but this time using the hierarchical operators. Column 4 shows the length of the high-level plans and Column 5 shows the time needed to generate this high-level plan and then decompose it. Plan 1, obtained from the hierarchical algorithm, expands to give a plan length of 18, i.e., exactly the same plan obtained by running its corresponding single-level algorithm. The timing results show the hierarchical approach is more efficient than the single-level approach. This 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, our modeling algorithm is able to obtain increasing number of levels of abstraction. We performed some exploratory analysis for the much larger GUI of Microsoft Word. There, the automatic modeling process reduced the number of operators by a ratio of 20:1.

6 Related Work

Current tools to aid the test designer in the testing process include record/playback tools [20], [21]. These tools record the user events and GUI screens during an interactive session. The recorded sessions are later played back whenever it is necessary to recreate the same GUI states. Several attempts have been made to automate test case generation for GUIs. One popular technique is programming the test case generator [22]. 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 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 [23], [24], [25], [26]. 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 the test generation for testing hardware circuits [27]. 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 an FSM; 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 FSM models have scaling problems [28]. To aid in the scalability of the technique, variations such as variable finite state machine (VFSM) models have been proposed by Shehady and Siewiorek. [28].

Test cases have also been generated to mimic novice users [7]. The approach relies on an expert to manually generate the initial sequence of GUI events and, then uses genetic algorithm techniques to modify and extend the sequence. 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. The final test suite depends largely on the paths taken by the expert user.

AI planning has been found to be useful for generating focused test cases [29] for a robot tape library command language. 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 can generate a "plan" which can be executed on the software as a test case. Note that although this technique has similarities to our approach, several differences exist: A major difference is that in [29], each command in the language is modeled with a distinct operator. This approach works well for systems with a relatively small command language. However, because GUIs typically have a large number of possible user actions, a hierarchical approach is needed.

7 Conclusions

In this paper, we presented a new technique for testing GUI software and we showed its potential value for the test designer's tool-box. Our technique employs GUI 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. Our technique is unique in that we use an automatic planning system to generate test cases from GUI events and their interactions. We use the description of the GUI to automatically generate alternative sequences of events from pairs of initial and goal states by iteratively invoking the planner.

We have demonstrated that our technique is both practical and useful by generating test cases for the popular MS WordPad software's GUI. Our experiments showed that the planning approach was successful in generating test cases for different scenarios. We developed a technique for decomposing the GUI at multiple levels of abstraction. Our technique not only makes test case generation more intuitive, but also helps scale our test generation algorithms for larger GUIs. We experimentally showed that the hierarchical modeling approach was necessary to efficiently generate test cases.

Hierarchical test case generation also aids in performing regression testing. Changes made to one part of the GUI do not invalidate all the test cases. Changes can be made to lower level test cases, retaining most of the high-level 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 our low-level test cases to sequences of physical actions. Such platform-dependent mappings can be maintained in libraries to customize our generated test cases to low-level, platform-specific test cases.

We note some current limitations of our approach. First, the test case generator is largely driven by the choice of tasks given to the planner. Currently in PATHS, these tasks are chosen manually by the test designer. A poorly chosen set of tasks will yield a test suite that does not provide adequate coverage. We are currently exploring the development of coverage measures for GUIs. Second, we depend heavily on the hierarchical structure of the GUI for efficient test case generation. If PATHS is given a poorly structured GUI then no abstract operators will be obtained and the planning will depend entirely on primitive operators, making the system inefficient. Third, our approach must be used in conjunction with other test case generation techniques to adequately test the software as is generally the case with most test case generators.

One of the tasks currently performed by the test designer is the definition of the preconditions and effects of the operators. Such definitions of commonly used operators can be maintained in libraries, making this task easier. We are also currently investigating how to automatically generate the preconditions and effects of the operators from a GUI's specifications.

Acknowledgments

This research was partially supported by the US Air Force Office of Scientific Research (F49620-98-1-0436) and by the US National Science Foundation (IRI-9619579). Atif Memon was partially supported by the Andrew Mellon Predoctoral Fellowship.

The authors would like to thank the anonymous reviewers of this article for their comments and Brian Malloy for his valuable suggestions. A preliminary version of the paper appeared in the *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles, May 1999 [30].

References

[1] B.A. Myers, "Why are Human-Computer Interfaces Difficult to Design and Implement?" Technical Report CS-93-183, School of Computer Science, Carnegie Mellon Univ., July 1993.

[2] W.I. Wittel Jr. and T.G. Lewis, "Integrating the MVC Paradigm into an Object-Oriented Framework to Accelerate GUI Application Development," Technical Report 91-60-06, Dept. of Computer Science, Oregon State Univ., Dec. 1991.

[3] B.A. Myers, "User Interface Software Tools," *ACM Trans. Computer-Human Interaction,* vol. 2, no. 1, pp. 64–103, 1995.
[4] D. Rosenberg, "User Interface Prototyping Paradigms in the 90's," *Proc. Conf. Human Factors in Computing Systems—Adjunct Proc. (ACM INTERCHI '93),* p. 231, 1993.
[5] M.G. El-Said, G. Fischer, S.A. Gamalel-Din, and M. Zaki, "ADDI: A Tool for Automating the Design of Visual Interfaces," *Computers & Graphics,* vol. 21, no. 1, pp. 79–87, 1997.
[6] L. White, "Regression Testing of GUI Event Interactions," *Proc. Int'l Conf. Software Maintenance,* pp. 350–358, Nov. 1996.
[7] D.J. Kasik and H.G. George, "Toward Automatic Generation of Novice User Test Scripts," *Proc. Conf. Human Factors in Computing Systems: Common Ground,* M.J. Tauber, V. Bellotti, R. Jeffries, J.D. Mackinlay, and J. Nielsen, eds., pp. 244–251, Apr. 1996.
[8] R.M. Mulligan, M.W. Altom, and D.K. Simkin, "User Interface Design in the Trenches: Some Tips on Shooting from the Hip," *Proc. Conf. Human Factors in Computing Systems (ACM CHI '91),* pp. 232–236, 1991.
[9] J. Nielsen, "Iterative User-Interface Design," *Computer,* vol. 26, no. 11, pp. 32–41, Nov. 1993.
[10] M.M. Kaddah, "Interactive Scenarios for the Development of a User Interface Prototype," *Proc. Fifth Int'l Conf. Human-Computer Interaction,* vol. 2, pp. 128–133, 1993.
[11] A. Kaster, "User Interface Design and Evaluation—Application of the Rapid Prototyping Tool EMSIG," *Proc. Fourth Int'l Conf. Human-Computer Interaction,* vol. 1, pp. 635–639, 1991.
[12] H. Kautz and B. Selman, "The Role of Domain-Specific Knowledge in the Planning as Satisfiability Framework," *Proc. Fourth Int'l Conf. Artificial Intelligence Planning Systems (AIPS '98),* R. Simmons, M. Veloso, and S. Smith, eds., pp. 181–189, 1998.
[13] A. Walworth, "Java GUI Testing," *Dr. Dobb's J. Software Tools,* vol. 22, no. 2, pp. 30, 32, and 34, Feb. 1997.
[14] M. Peot and D. Smith, "Conditional Nonlinear Planning," *Proc. First Int'l Conf. AI Planning Systems,* J. Hendler, ed., pp. 189–197, June 1992.
[15] D.S. Weld, "An Introduction to Least Commitment Planning," *AI Magazine,* vol. 15, no. 4, pp. 27–61, 1994.
[16] D.S. Weld, "Recent Advances in AI Planning," *AI Magazine,* vol. 20, no. 1, pp. 55–64, 1999.
[17] J. Koehler, B. Nebel, J. Hoffman, and Y. Dimopoulos, "Extending Planning Graphs to an ADL Subset," *Lecture Notes in Computer Science,* vol. 1348, pp. 273, 1997.
[18] A.L. Blum and M.L. Furst, "Fast Planning Through Planning Graph Analysis," *Artificial Intelligence,* vol. 90, no. 1–2, pp. 279–298, 1997.
[19] K. Erol, J. Hendler, and D.S. Nau, "HTN Planning: Complexity and Expressivity," *Proc. 12th Nat'l Conf. Artificial Intelligence (AAAI '94),* vol. 2, pp. 1123–1128, Aug. 1994.
[20] L. The, "Stress Tests For GUI Programs," *Datamation,* vol. 38, no. 18, p. 37, Sept. 1992.
[21] M.L. Hammontree, J.J. Hendrickson, and B.W. Hensley, "Integrated Data Capture and Analysis Tools for Research and Testing a Graphical User Interfaces," *Proc. Conf. Human Factors in Computing Systems,* P. Bauersfeld, J. Bennett, and G. Lynch, eds., pp. 431–432, May 1992.
[22] L.R. Kepple, "The Black Art of GUI Testing," *Dr. Dobb's J. Software Tools,* vol. 19, no. 2, p. 40, Feb. 1994.
[23] J.M. Clarke, "Automated Test Generation from a Behavioral Model," *Proc. Pacific Northwest Software Quality Conf.,* May 1998.
[24] T.S. Chow, "Testing Software Design Modeled by Finite-State Machines," *IEEE Trans. Software Eng.,* vol. 4, no. 3, pp. 178–187, Mar. 1978.
[25] S. Esmelioglu and L. Apfelbaum, "Automated Test Generation, Execution, and Reporting," *Proc. Pacific Northwest Software Quality Conf.,* Oct. 1997.
[26] P.J. Bernhard, "A Reduced Test Suite for Protocol Conformance Testing," *ACM Trans. Software Eng. and Methodology,* vol. 3, no. 3, pp. 201–220, July 1994.
[27] H. Cho, G.D. Hachtel, and F. Somenzi, "Redundancy Identification/Removal and Test Generation for Sequential Circuits Using Implicit State Enumeration," *Trans. Computer-Aided Design of Integrated Circuits and Systems,* vol. 12, no. 7, pp. 935–945, July 1993.
[28] R.K. Shehady and D.P. Siewiorek, "A Method to Automate User Interface Testing Using Variable Finite State Machines," *Proc. 27th Ann. Int'l Symp. Fault-Tolerant Computing (FTCS '97),* pp. 80–88, June 1997.
[29] A. Howe, A. von Mayrhauser, and R.T. Mraz, "Test Case Generation as an AI Planning Problem," *Automated Software Eng.,* vol. 4, pp. 77–106, 1997.
[30] A.M. Memon, M.E. Pollack, and M.L. Soffa, "Using a Goal-Driven Approach to Generate Test Cases for GUIs," *Proc. 21st Int'l Conf. Software Eng.,* pp. 257–266, May 1999.

Atif M. Memon received the BS and MS degrees in computer science in 1991 and 1995, respectively. He enrolled at the University of Pittsburgh in 1996 and is currently a PhD candidate. He was awarded a Fellowship from the Andrew Mellon Foundation for his PhD research. His research interests include program testing, software engineering, artificial intelligence, plan generation, and code improving compilation techniques. He is a member of the ACM and a student member of both the IEEE and the IEEE Computer Society.

Martha E. Pollack received the AB degree (1979) in linguistics from Dartmouth College and the MSE (1984) and PhD (1986) degrees in computer and information science from the University of Pennsylvania. She is a professor of computer science and director of the Intelligent Systems Program at the University of Pittsburgh. From 1985 until 1991, she was employed at the Artificial Intelligence Center at SRI International. Dr. Pollack is a recipient of the Computers and Thought Award (1991), a US National Science Foundation Young Investigator's Award (1992), and is a fellow of the American Association for Artificial Intelligence (1996). She is also a member of the editorial board of the *Artificial Intelligence Journal* and on the advisory board of the *Journal of Artificial Intelligence Research*. Dr. Pollack also served as program chair for IJCAI '97. Her research interests include computational models of rationality, planning and reasoning in dynamic environments, and assistive technology.

Mary Lou Soffa received the PhD degree in computer science from the University of Pittsburgh in 1977. She is a professor of computer science at the University of Pittsburgh. She served as the graduate dean of arts and sciences at the University of Pittsburghfrom 1991 through 1996. In 1999, she received the Presidential Award for Excellence in Science, Mathematics, and Engineering Mentoring. She also was elected an ACM fellow in 1999. She currently serves on the editorial board for *ACM Transactions on Programming Languages and Systems*, *IEEE Transactions of Software Engineering*, *International Journal of Parallel Programming*, *Computer Languages*, and the *South African Journal of Computing*. She serves as vice president for the Computing Research Association (CRA) and also as cochair of the CRA's Committee on the Status of Women in CSE. She is currently a member-at-large for ACM SIGBoard. She has served as conference chair, program chair, and program committee member for conferences in both programming languages and software engineering. Her research interests include optimizing and parallelizing compilers, program analysis, and software tools for debugging and testing programs. She is a member of the IEEE and the IEEE Computer Society.