A Planning-based Approach to GUI Testing*

Atif M. Memon†, Martha E. Pollack, Mary Lou Soffa
Dept. of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260 USA
+1 412 624-8850
{atif, pollack, soffa}@cs.pitt.edu

Abstract
Graphical user interfaces (GUIs) have become nearly ubiquitous as a means of interacting with software systems. The widespread use of GUIs is leading to the construction of more and more complex GUIs. With the growing complexity comes challenges in testing the correctness of GUIs and the underlying software. Some of the important challenges include test-case generation, test-oracle creation, and regression testing. In this paper, we present the design of Planning Assisted Tester for grapHical user interface Systems (PATHS) – a research project designed with the primary goal of facilitating the automation of GUI testing. PATHS uses a new GUI testing technique based on user event interaction sequences. The key idea is to test the GUI software using interactions most likely to be exercised in actual use. A novel feature of PATHS is its reliance on AI plan generation techniques to generate testing information. Given a set of operators, an initial state and a goal state, a planning system produces a sequence of operators that transforms the initial state to the goal state. Using PATHS, GUI test designers can generate likely user interaction sequences by specifying typical goals that users of the GUI software might have. PATHS first analyzes the GUI and derives hierarchical planning operators from the actions in the GUI. The test designer determines the preconditions and effects of the hierarchical operators, which are then input into a planning system. With the knowledge of the GUI and the way in which the user will interact with the GUI, the test designer creates sets of initial and goal states. Given these initial and final states of the GUI, a hierarchical planner produces plans, or a set of test cases, that enable the goal state to be reached. Our technique has the additional benefit of associating oracle information with the test cases automatically. We implemented our technique by developing the GUI analyzer and extending a planner. We generated test cases for Microsoft's WordPad to demonstrate the viability and practicality of the approach.

Keywords
GUI testing, application of planning, GUI regression testing, automated test-case generation,

1 Introduction

Testing is a critical component of the software development process and is required to ensure the safety, robustness and usability of software. Unfortunately, it is also labor and resource intensive, accounting for 50%-60% of the total cost of software development [9, 29]. Hence, there has been significant research aimed at automating the testing process. Although some success has been achieved, many problems remain. In particular, it is not yet clear how to automate the testing of user interfaces, which constitute an increasingly large portion of software systems, as much as 45-60% of the total software code [21, 25]. The most popular form of user interfaces are direct-manipulation interfaces called Graphical User Interfaces (GUIs) [26]. GUIs have become an important and accepted way of interacting with today's software. As GUIs become more and more popular, they are increasingly being used in critical systems [35] and testing them is necessary to avert catastrophes [26].

Testing the correctness of a GUI is difficult for a number of reasons. First, the space of possible interactions with a GUI is enormous. Each sequence of GUI actions can result in a different state of the combined system (i.e., the GUI and underlying software). In general, a GUI action might have different results in each state, and thus need to be tested in a very large number of states: the amount of testing required can be enormous [34]. Related to this is the fact that measures of coverage that have been defined for testing conventional software systems do not work well for GUIs. For conventional software, coverage is measured using the amount and type of underlying code exercised. In testing GUIs, while one must still be concerned with how much of the code is tested, there needs also to be significantly increased focus on the number of different possible states

* *Partially supported by the Air Force Office of Scientific Research (F49620-98-1-0436) and by the National Science Foundation (IRI-9619579) (EIA0906525).*
† *Partially supported by the Andrew Mellon Pre-doctoral Fellowship, awarded by the Andrew Mellon Foundation.*

in which each piece of code is exercised. Existing metrics do not allow one to say whether a GUI has been "well-enough" tested. As a result, GUI testing often relies on extensive beta testing: for example, Microsoft released almost 400,000 beta copies of Windows95 targeted at finding program failures [15].

GUI testing involves several steps. Initially, a set of test cases must be generated. This is particularly challenging for GUI testing, because of the difficulties mentioned above: the set of possible test cases is huge, and conventional metrics for selecting "good" test case sets do not apply. After test cases are constructed, they must be executed: this is when the actual "testing" occurs, to check whether the GUI is performing correctly. An incorrect GUI state can lead to an unexpected screen, making further execution of the test case useless because events in the test case might not match the corresponding GUI components on the screen. Consequently, the execution of the test case must be terminated as soon as an error is detected. Verification checks, performed by using test oracles, must therefore be inserted after each step, to catch errors as soon as they occur. Yet another challenge is posed by regression testing, i.e., updating the set of test cases and the verification check after changes are made to the GUI during development or maintenance. Regression testing presents special challenges for GUIs, because the input-output mapping often does not remain constant across successive versions of the software [24].

Current GUI testing practices involve a significant amount of manual effort on the part of the test designer. Most test designers employ tools that automate certain aspects of the testing cycle. Most common among such tools are *capture/playback tools* [13, 11] used to capture 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. These tools generally store information at a low level of abstraction, capturing actual mouse positions, button clicks and storing bit-maps. Representing the information at such a low level of abstraction makes it difficult to tailor the recorded session for other test cases. A popular alternative to using capture/playback tools is to program a test case generator. Programming requires that the test designer program all possible decision points present in the GUI. However, this approach is time consuming, and is susceptible to missing important GUI decisions. Moreover, the expected output must also be determined by the programmer.

In this paper, we present the design of Planning Assisted Tester for grapHical user interface Systems (PATHS) – a research project designed with the primary goal of facilitating the automation of GUI testing. PATHS is based on **planning** – a well developed and used technique in Artificial Intelligence (AI). Given a set of operators, an initial state and a goal state, a planning system produces a sequence of operators that will transform the initial state to the goal state. The key idea of using planning as the core of PATHS is that the GUI test designers will often find it easier to specify typical goals that users of the GUI software might have than to specify sequences of GUI actions that users might perform to achieve those goals. Thus we cast GUI testing as an instance of planning. PATHS has the goals of generating test cases automatically, incorporating oracle information into the testing process, and then automatically generating a regression test suite when re-testing is done.

The main contributions of PATHS are as follows.

- Most of the GUI testing tasks are automated so that the test designer's work is simplified as much as possible.
- The overall testing cycle defined by PATHS is efficient since software testing is usually a tedious and expensive process.
- PATHS is robust in that whenever the GUI enters an unexpected state, the testing algorithms detect the error state immediately, recover from it and report all information necessary to debug the GUI.
- The testing information generated by PATHS is portable. Test information (e.g., test cases, oracle information, coverage report, error report) generated and/or collected on one platform is usable on all other platforms on which the GUI can be executed.
- Finally, PATHS is general enough to be applied to a wide range of GUIs.

In the next section, we present a high-level overview of the design of PATHS. Section 3 presents a discussion of AI planning. In Section 4, we show how PATHS automatically models the GUI hierarchically so that a restricted form of hierarchical planning can be applied to efficiently generate testing information. In particular, we show in Section 5 how the hierarchical model is used to generate test cases. In Section 6, we also indicate how the hierarchical model is used to create test oracles and for regression testing. We present a discussion on related work in Section 7 and conclude in Section 8.

2 Overview of PATHS

PATHS uses a new GUI testing technique based on user event interaction sequences. The key idea is to test the GUI software using interactions most likely to be exercised in actual use. The primary function of PATHS is

to generate likely user interaction sequences and then to test the GUI using these sequences (test cases) as input. A novel feature of PATHS is its reliance on AI plan generation techniques to generate test cases. The central component of PATHS is a planning based test case generator. In addition to planning algorithms, PATHS is supplemented with techniques and algorithms for effective regression testing, coverage evaluation, and incorporation of domain specific knowledge. The test case generator is given a description of the GUI and test scenarios consisting of pairs of initial and goal states as input, and it generates test cases as output. The GUI description is assumed to provide a complete working of the GUI. An oracle information augmentation tool associates additional information with each test case to be used to verify the state of the GUI during test case execution. Currently, the PATHS system is capable of using GUI specifications to automatically generate a test suite, exploit the planning model to create test oracles, execute the test cases and pinpoint errors in the GUI. Various measures for coverage are undergoing development. The coverage evaluation algorithms will measure the quality of the generated test cases. The regression testing algorithms will use results from prior testing sessions to guide regression testing. Additional user supplied domain information will be used for increased efficiency and effectiveness. Actual execution of the test cases will be done by a GUI exerciser. The high-level interactions between the components of PATHS are shown in Figure 1. The ovals represent the processes that control the test case generation and execution. The outputs include the final test suite, coverage report, and the error report. Other entities are either generated by the components or provided as input by the test designer.

3 AI Plan Generation

PATHS makes use of planning for GUI testing. This section gives a brief introduction to planning and the different planning techniques.

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 sequence of actions (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 [32] and a survey of the recent advances in planning technology [33].

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. The solution to the planning problem is a plan: a tuple $< S, O, L, B >$ where S are steps (instances of operators – typically represented as sets of preconditions and effects), O are ordering constraints on the elements of S, L are causal links representing the causal structure of the plan, and B are binding constraints on the variables 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. Typically, the ordering constraints only induce a partial ordering, so the set of solutions are all linearizations of S consistent with O.

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 [8] which specifies operators in terms of parameterized preconditions and effects. STRIPS was developed more 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 [28, 27], 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), used in the AIPS'98 planning competition. 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.

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) [19], a system which extends the ideas of the Graphplan system [2] 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.* [22]).

IPP uses ADL for the representation of actions in which preconditions and effects can be parameterized: subse-

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

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

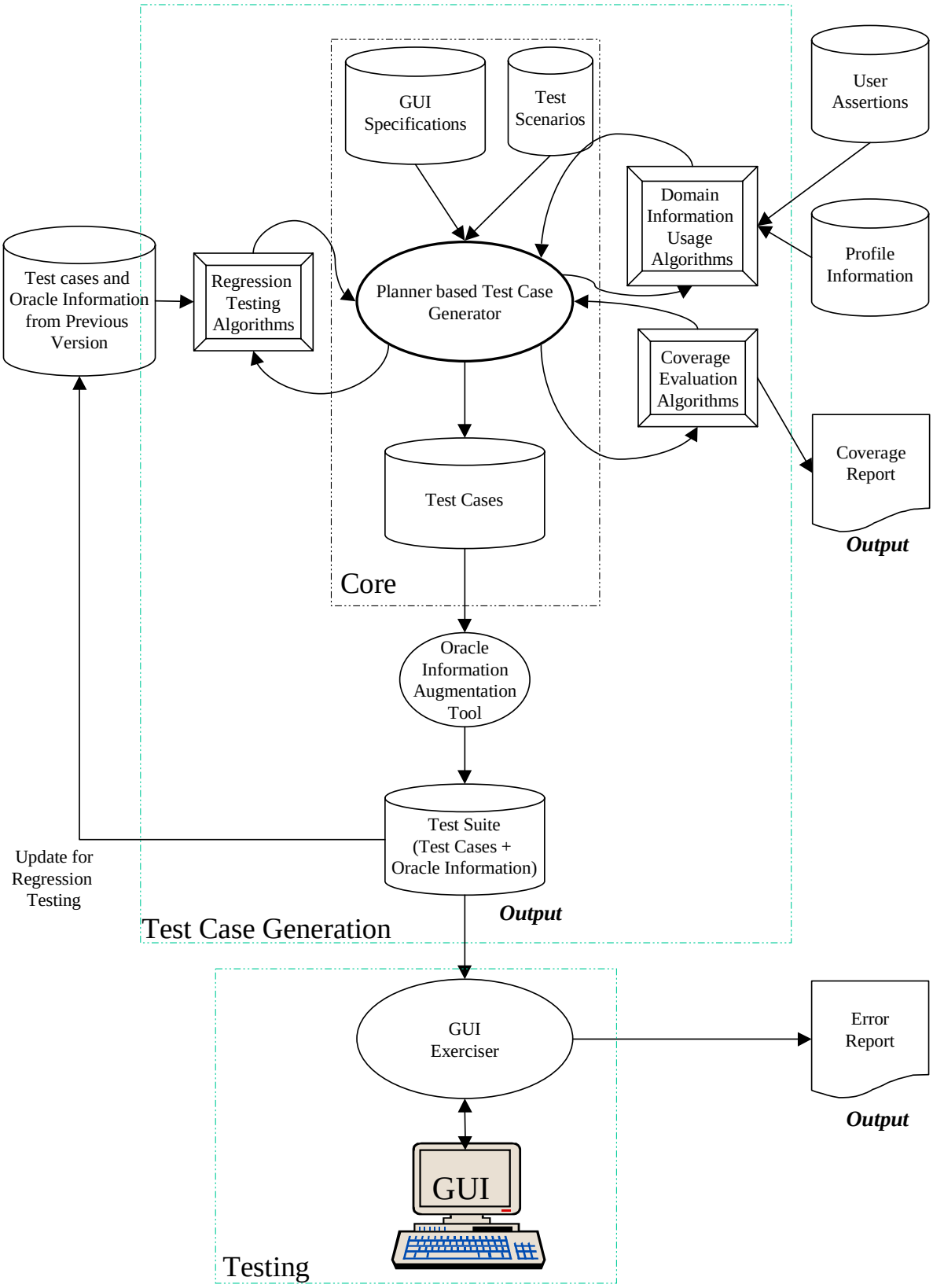


Figure 1: The Components of PATHS.

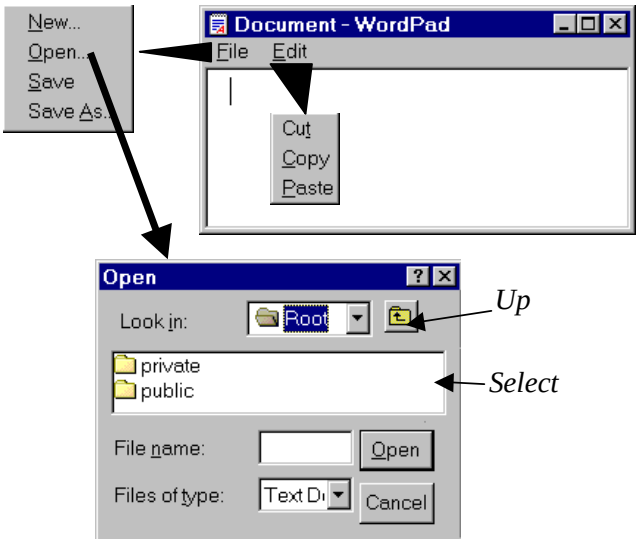


Figure 2: The Example GUI.

quent 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*.

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; this is typically called Hierarchical Task Network (HTN) planning [36, 6, 5]. 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 an action is said to "decompose" into its methods. HTN planning focuses on resolving conflicts among alternative methods of decomposition at each level.

4 Hierarchical Model of the GUI

In order to efficiently use planning to generate test information for GUIs, we first develop an efficient model of the GUI, based on its structure. In this section, we show how to create a hierarchical model of the GUI through an example.

A GUI Example

Figure 2 illustrates a small part of the Microsoft WordPad's GUI. With this GUI, the user can load text from files, manipulate the text by cutting and pasting, and save the text into a file. At the highest level, the GUI contains a menu bar that allows the user to perform two possible actions: clicking `File` and clicking `Edit`. When either of these are clicked, other menus open, making other actions available to the user. We say that a user performs a GUI action (e.g., clicks the `File` command), and thereby generates a GUI event (e.g., opening up a pull-down menu). For convenience, we sometimes also speak of the `File` action, meaning the action of clicking `File`. Note that the user can also generate events by using the keyboard, e.g., by entering text onto the screen.

Finally, we also distinguish between two types of windows: *GUI windows* and *object windows*. The former contain only GUI components (labels, buttons, commands, etc.); the "Open" window at the bottom of the Figure 2 is an example. In contrast, object windows display and manipulate other, non-GUI objects; an example is the MS WordPad window that displays text.

In the example, we include a number of user actions that involve clicking a component, e.g., clicking `File` or clicking `Cut`. These components have their usual meanings. We also provide labels for two other user actions: `Up`, which involves clicking the arrow-in-a-folder icon, and generating the event of moving one level up in the directory hierarchy; and `Select`, which is used to either enter subdirectories or select files.

Finally, Figure 3 illustrates a planning problem for the planning-based test-case generator. The initial state, depicted in part (a), shows the contents of a collection of files stored in a directory hierarchy. It also shows the contents of some of those files. The goal state is shown in part (b) of the figure. The goal is to create a new document, with the specified text ("This is the final text."), and store it in the file *new.doc* in the `/root/public` directory. Note that the goal can be obtained in various ways. In particular, to get the text into `new.doc`, one could load file `Document.doc` and delete the extra text, or could load file `doc2.doc` and insert text, or could create the document from scratch by typing in the text.

Deriving Hierarchical Operators

We now describe how PATHS models the GUI hierarchically thus enabling the application of hierarchical planning. The modeling process starts with PATHS creating a list of operators to be used during planning. The simplest approach would be to list exactly one operator per GUI action. Although conceptually simple, this approach turns out to be inefficient, and can be improved upon by exploiting the GUI structure to derive hierarchical operators that are decomposed during planning. We use two distinct forms of decomposition. In the first, *system-interaction operators* are constructed to model sequences of GUI events $E_1, \ldots, E_n$ such that for $1 \leq i \leq n-1$, E_i makes available the user action that generates E_{i+1}. When these operators are used in a plan, they are later decomposed by a process we call *mapping*, which is similar to macro expansion. In the

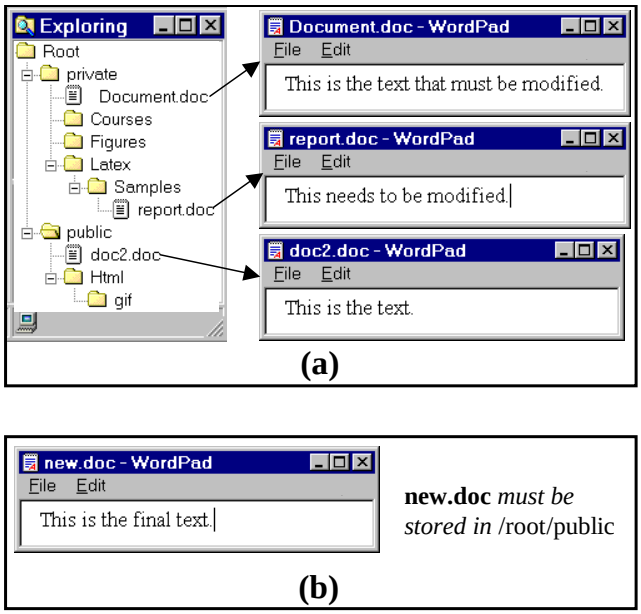


Figure 3: A Task for the Planning System; (a) the Initial State, and (b) the Goal State.

second, *abstract operators* are constructed to model GUI events that lead to sequences of GUI events which are themselves best viewed as "sub-plans". These operators are similar to abstract operators in HTN planning; when they are used in a plan, they are later decomposed by an embedded call to the planner. We give examples of both types of operators below.

The first step in deriving the operators is to partition the GUI events into three classes, listed below. The classification is based only on the structural properties of GUIs and can thus be done automatically by PATHS.

Unrestricted-focus events open GUI windows that do not restrict the user's focus; they merely expand the set of GUI actions available to the user. Often such events open menus, e.g., the events generated by clicking `File` or `Edit` in our example.

Restricted-focus events open GUI windows that have the special property that once invoked, they monopolize the GUI interaction. These windows restrict the the user to a specific range of GUI actions available within the window; other GUI actions cannot be performed until the window is explicitly terminated. An example of a restricted-focus event is preference setting in many GUI systems. The user clicks on `Edit` and then `Preferences`, after which a "Preferences" window opens. The user can them modify preferences, but cannot interact with the system in any other way until s/he explicitly terminates the interaction by either clicking `OK` or `Cancel`.

System-interaction events interact with the underlying software. Common examples include cutting and pasting text, saving files, etc.

Note that these three classes are exhaustive and mutually exclusive.

System-Interaction Operators

Once the GUI events have been classified, two types of planning operators can be automatically constructed. The first are *system-interaction operators*, which represent sequences of GUI actions that a user might perform to eventually interact with the underlying software. More specifically, a system-interaction operator is a sequence of zero or more unrestricted-focus events, followed by a single system-interaction event. Consider a part of the example GUI: a menu bar with one option (`Edit`), which can be clicked to provided more options, i.e., `Cut` and `Paste`. The complete set of actions available to the user is thus `Edit`, `Cut` and `Paste`. `Edit` generates an unrestricted-focus event, while `Cut` and `Paste` generate system-interaction events. Using this information, PATHS would create two system-interaction operators: `Edit_Cut` and `Edit_Paste`.

The use of system-interaction operators reduces the total number of operators made available to the planner, resulting in greater planning efficiency. In our small example from the previous paragraph, the events `Edit`, `Cut` and `Paste` would be hidden from the planner; only the system-interaction operators namely, `Edit_Cut` and `Edit_Paste`, would be made available to the planner. This event-hiding prevents generation of test cases in which `Edit` is used in isolation; any test case that includes an `Edit` will also include an immediately following `Cut` or with `Paste`. To overcome this restriction and to increase coverage, `Edit` can be tested in isolation, and/or additional test cases can be created by inserting `Edit` at random places in the generated test cases.

When a generated test case includes system-interaction operators, PATHS must eventually decompose those operators to primitive GUI events, so that the test case can be directly executed. Thus, PATHS keeps track of the sequence of GUI events that corresponds to each system-interaction operator it derives, storing this information in a a table of *operator-event mappings*. The event operator table for the sub-example of the previous paragraph is shown in Table 1.

Abstract Operators

The second type of operators that are constructed by PATHS are *abstract operators*. These are created from the restricted-focus events, which contain two parts. The *prefix* of an abstract operator is a sequence of zero

Operator Name	GUI Event Sequence
File_New	<File, New>
File_Save	<File, Save>
Edit_Cut	<Edit, Cut>
Edit_Copy	<Edit, Copy>
Edit_Paste	<Edit, Paste>

Table 1: Operator-event Mappings of the System-interaction Operators for the Example GUI.

or more unrestricted-focus events followed by a single restricted-focus event. As was the case with system-interaction operators, PATHS stores an operator-event mapping for the prefix of each abstract operator. The *suffix* of an abstract operator represents the possible events that may occur while the restricted-focus window is opened. However, the representation is only indirect, specifying the information that is needed for an embedded call to the PATHS planner.

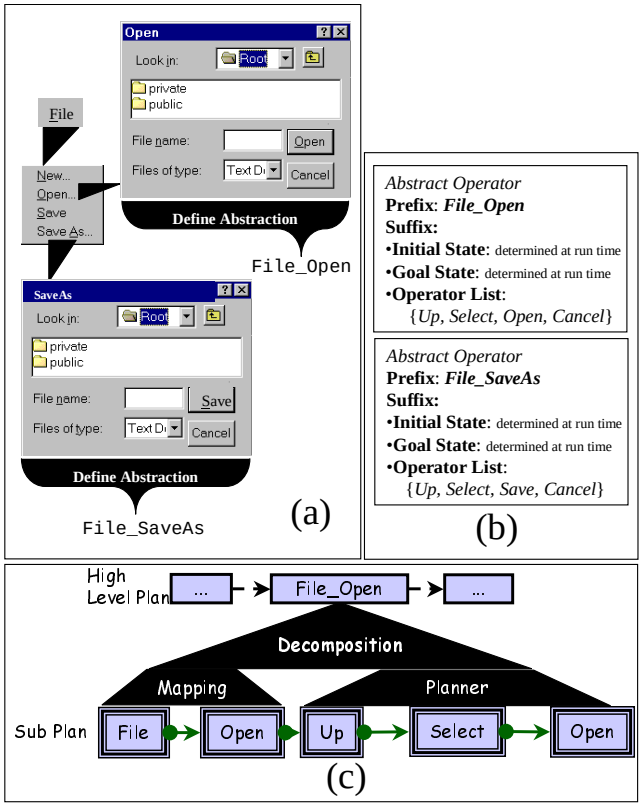


Figure 4: (a) Restricted Focus Operators: `Open` and `SaveAs` (b) Abstract Operators, and (c) Decomposition of the Abstract Operator

The idea behind abstract operators can be clarified with an example. Figure 4 focuses again on a small part of the running example from Figure 2. This time, the `File` menu with two options, namely `Open` and `SaveAs`. `Open` and `SaveAs` are both restricted-focus operators, which cause restricted-focus windows to be opened. Here PATHS creates two abstract operators: `File_Open` and `File_SaveAs`. (For convenience, we name each abstract operator by its prefix.) An operator-event mapping will be created for each prefix:

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

In addition, PATHS creates a suffix for each operator. The suffix contains all the information that is required to define a planning problem π, such that the solution to π is a sequence of GUI events that could reasonably occur in the current context, while the the restricted-focus window is open. The suffix of each abstract operator then is essentially a hierarchical operator, which is decomposed during planning by a separate call to the planner itself. The sub-plans thus produced can be stored and reused, essentially playing the role of methods in traditional HTN planning.

Figure 4(b) shows the abstract operators that are created for the current example. Note that the suffix contains a list of operators that the planner may use in generating a sub-plan for the restricted-window interaction; it also contains slots for the initial state and the goal state of the embedded planning problem. When the test-cases are actually being planned, they are created one level at a time. At the highest level, a plan is created using system-interaction operators and abstract operators; during this stage of planning, the suffixes of any abstract operators used in the plan will remain undecomposed. Subsequently, the system-interaction operators and the prefixes of abstract operators will be decomposed by direct use of the operator-event mappings. The suffixes of abstract operators will be decomposed by calls to the planner itself; by the time of these calls, the initial and goal states for the sub-plan can be directly derived from the high-level plan.

Part (c) of Figure 4 illustrates the decomposition process. At the top is a high-level plan that includes the abstract operator `File_Open`. Its prefix is decomposed with an operator-event mapping, to produce the first two steps in the plan, `File` and `Open`. Its suffix is decomposed with a call to the planner, which produces the linear plan `Up`; `Select`; `Open`.

At the end of the first step of the setup phase, a set of system-interaction and abstract operators have been computed automatically. These are then passed to the test designer for completion. The planning operators returned for the complete example in Figure 2 are shown in Figure 5(b).

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)

Figure 5: The Running Example GUI: (a) GUI Actions, and (b) System-Interaction Operators derived by PATHS.

Operator :: Edit_Cut
Preconditions:
$\exists$ *Obj* $\in$ *Objects* | Selected(*Obj*).

Effects:
$\forall$ *Obj* $\in$ *Objects* | Selected(*Obj*) $\Rightarrow$
ADD inClipboard(*Obj*) Λ
DEL onScreen(*Obj*) Λ
DEL Selected(*Obj*).

Figure 6: An Example of a GUI Planning Operator.

Completing the Planning Operators

In the second step of the setup phase, the test designer specifies the preconditions and effects for each planning operator derived in the first step. As is standard, the preconditions represent all the conditions that must hold for the event represented by the operator to occur, and the effects represent the resulting changes to the environment (i.e., the GUI and/or the underlying software).

An example is given in Figure 6. `Edit_Cut` is a system-interaction operator. Its preconditions express that in order for the user to generate the `Cut` event (by performing the actions `EDIT` followed by `CUT`), at least one object on the screen must be selected (highlighted). The effects express the facts that the selected objects are moved to the clipboard and removed from the screen.

The language used to define the preconditions and effects of each operator is provided by the planning system. Defining the preconditions and effects is not difficult, as much of 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`). Definitions of operators representing events that commonly appear across GUIs, such as `Cut`, can be maintained in a library and reused for subsequent similar applications.

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:
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.)

Figure 7: Initial State and Goal State Describing the Task of Figure 3.

5 Test-Case Generation

We now see how the hierarchical GUI model is used to generate test cases. Once the hierarchical operators have been derived, test-case generation can begin. First, the test designer describes a typical user task by specifying it in terms of initial and goal states. Figure 7 provides an example. In the current version of PATHS, the test designer models the initial and goal states directly. However, we plan to develop a tool that would allow the test designer to visually describe the GUI's initial and goal states, and would then translate the visual representation into a planner encoding.

Once the task has been specified, the system automatically generates a set of distinct test cases that achieve the goal. An example of one such plan is shown in Figure 8.[3] This is a high-level plan that must be decomposed. Figure 9 shows one decomposition; note that it includes both decomposition by mapping and decomposition by planning.

As we have noted before, it is important to generate alternative plans for each specified task, since these correspond to alternative ways in which a user might interact with the GUI. We achieve this goal in three ways:

- We run the planner several times, each time producing a distinct high-level plan.

[3]Note that `TypeInText()` is an operator representing a keyboard event, mentioned in "A GUI Example." This operator has its obvious meaning: it represents the event that occurs when the user types the text to which its parameter is bound.

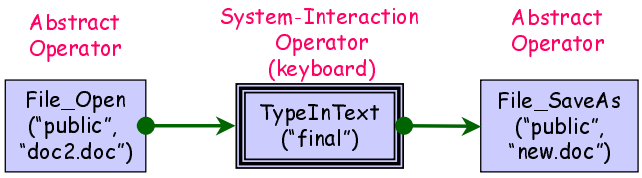
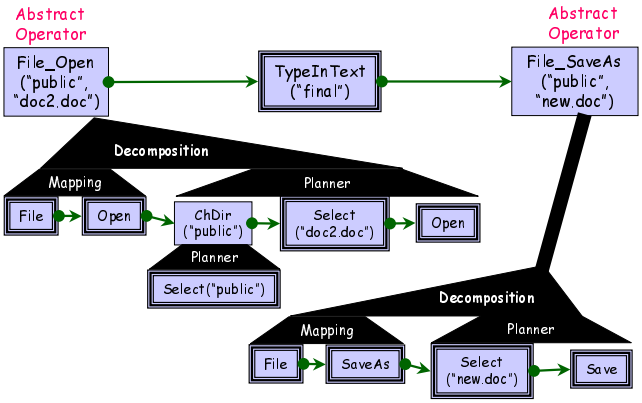


Figure 8: A Plan Consisting of Abstract Operators and a GUI Event.



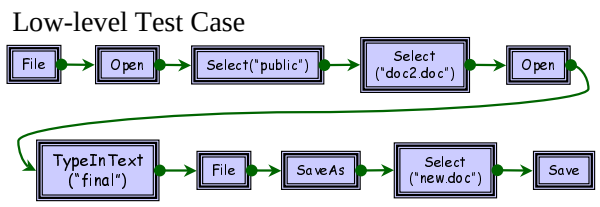


Figure 9: Expanding the Higher Level Plan.

- Because we are using a partial-order planner, each of the high-level plans can potentially be mapped to more than one distinct linearization.
- Each of the linear plans can potentially be decomposed in multiple ways.

This is particularly important, because our experiments have shown that the bulk of the time spent in test-case generation is used in generating the highest level plan. Whenever a plan includes an abstract operator, we invoke the planner multiple times to produce multiple distinct sub-plans that can serve as decomposition of the (suffix of the) abstract operator. Unlike typical HTN planning, the sub-plans in this setting do not interact, because they each take place during a separate restricted-focus phase; thus the application of plan critics is not required here. Figure 10 shows an alternative plan created from the high-level plan in Figure 8. It differs from the decomposition in Figure 9 in that it uses a different decomposition of the first abstract operator.

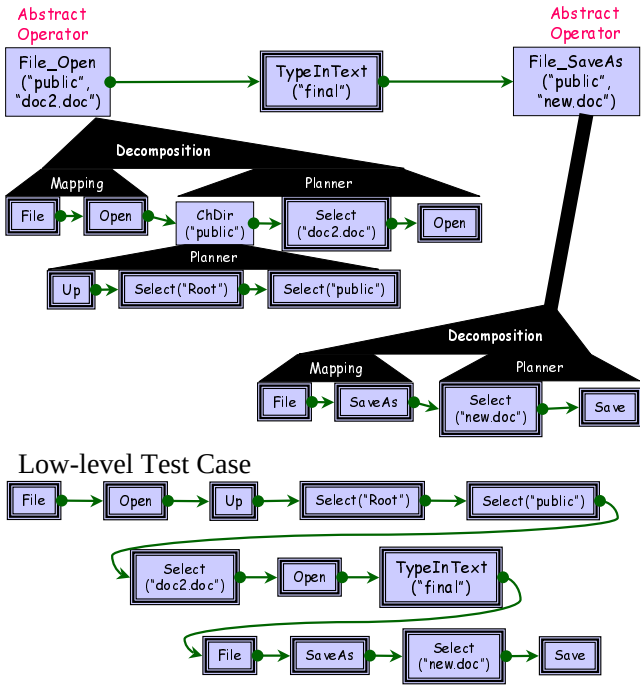


Figure 10: An Alternative Decomposition of the Abstract Operator Leads to a New Test Case.

The decomposition mechanism also aids regression testing, because changes made to one component of the GUI do not necessarily invalidate all test cases. The higher level plans can thus often be retained after changes to the GUI; local changes can instead be made to sub-plans specific to the changed component of the GUI. Another advantage to using decomposition is that the operators can be modeled at a platform-independent level, resulting in platform-independent test cases. An additional layer of mapping-like decomposition can then be added to produce platform-dependent test cases.

Algorithm for Generating Test Cases

The complete test-case generation algorithm is shown in Figure 11. The operators are assumed to be available before making a call to this algorithm. The parameters (lines 1..5 of the algorithm)[4] include the initial and goal states for the planning problem, and the set of available operators. An addition parameter specifies a threshold (**T**) that determines the number of iterations performed.

The main loop (lines 8..12) contains the explicit call to the planner (denoted by the function Φ). Each time the planner is invoked, a distinct plan is generated using the available operators, and is then stored in Λ. Note that no decomposition occurs within this loop: the plans

[4]Each command in the program is given a separate number, but for space reasons, we sometimes show two commands on one line.

```
                                                              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),
   to generate distinct solutions */
  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
```

Figure 11: The Complete Algorithm for Generating Test Cases

that are generated are all "flat". To guarantee that the planner does not regenerate duplicate plans, we modify each operator so that it is not used to generate previously generated plans. The key idea is to make the goal state unreachable, if the operator instances are used in a previously generated sequence. Using this approach to generate alternative plans, instead of backtracking in the planner's search space, makes our algorithm planner independent.

Once a set of distinct plans have been generated, linearizations are created for each one (lines 13..16). Each linear plan is then decomposed, potentially in multiple ways. As described earlier, system-interaction steps are decomposed using the operator-event mappings (lines 20..22), while abstract steps are decomposed using the mappings for the prefix, and using a recursive call to the test-case generation algorithm for the suffix (lines 23..28). The initial and goal states for the recursive planning problem are extracted directly from the high-level plan, which is available at the recursive call. The sub-plans obtained as a result of the recursive call are then substituted into the high-level plans (lines 29..31), and the new plans obtained are appended to the list of **testCases** (line 32). The final outcome of the algorithm is a set of distinct, fully decomposed plans for the specified task, that can serve as test cases for the GUI (line 33).

Feasibility Experiments

A prototype of PATHS was developed with IPP [20] as the underlying planning system. IPP was chosen based on the results of experiments in which causal-link planners were compared with propositional planners [22]. The key result of the study was that propositional planners perform better in domains such as GUI testing, which contain a small number objects.

We now present several sets of experiments, that were conducted to ensure that the approach in PATHS is feasible. These experiments were executed on a Pentium based computer with 200MB RAM running Linux OS. A summary of the results of these experiments is given next.

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 2. The total number of GUI events in WordPad was determined to be approximately 325. After deriving hierarchical operators, PATHS reduced this set to 32 system-interaction and abstract operators, a reduction of roughly 10 : 1. This reduction in the number of operators helps to speed up the plan generation process significantly.

Defining preconditions and effects for the 32 operators was fairly straightforward. The average operator definition required 5 preconditions and effects, with the most complex operator requiring 10 preconditions and effects. Since mouse and keyboard events are part of the GUI, additional operators representing mouse (i.e., `Select_Text()`) and keyboard (i.e., `TypeInText()` and `DeleteText()`) events were defined.

Table 3 presents a typical set of CPU execution timings for this experiment. Each row represents one task. The first column identifies the task; the second gives the average time to generate a single high-level plan for the task and the third shows the time taken to generate a family of test cases by producing all the decompositions of the plan. The fourth column gives the total planning time. As can be seen, the bulk of the time is spent generating the high-level plan. Sub-plan generation is

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 2: Some WordPad Plans Generated for the Task of Figure 3.

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 3: Average Time Taken to Generate Test Cases for WordPad.

quite fast, amortizing the cost of initial 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.

The plans shown in Table 2 are at a still at a relatively high level of abstraction. Many changes that might be made to a GUI would have no effect on these plans, making regression testing easier and less expensive. For example, none of the plans in Table 2 contain any GUI details such as font or color. The test suite continues to be useful even in the face of changes to these aspects of the GUI. The same is true for certain changes that modify the functionality of the GUI. For example, if the WordPad GUI were modified to introduce an additional file opening feature, then most of the high-level plans remain the same. Changes would only be needed to sub-plans that are generated by the abstract operator `File_Open`. Hence the cost of initial plans is again amortized over a large number of test cases.

6 Creating Test Oracles and Regression Testing

Although we have only presented details of test-case generation, PATHS is currently being used to generate testing information for other phases of the testing process. In particular, we are using PATHS to automatically create test oracles and for regression testing. We indicate how we achieve this in the next few paragraphs. A detailed discussion is beyond the scope of this paper.

PATHS automatically generates test oracles to check the GUI's state during and after test execution. The key to oracle creation is to have oracle information integrated into the test case so that the GUI exerciser knows what to check at each step. One of the primary motivations of using planning for GUI testing is that much of the

state can be recovered directly from the planning model. During plan construction, the planner keeps track of the intermediate states of the GUI. PATHS extracts these intermediate states and integrates them with the generated test case to act as oracles.

PATHS is being supplemented with efficient regression testing techniques that draw on planning. The hierarchical decomposition of the GUI done by PATHS allows identification and isolation of GUI components that were modified. Changes are then made only to the effected test cases and oracle information. Moreover, hierarchical planning aids in retaining most of the test cases and oracle information defined at the higher levels of abstraction. Changes are made only at the lower levels.

7 Related Work

The manual creation of test cases and their maintenance and evaluation is in general a very time consuming process. Thus some form of automation is desirable. One class of tools that aid a test designer are capture/playback tools [31, 12]. These tools record the user events and GUI screens during an interactive session; the recorded sessions can later be played back when it is necessary to recreate the same GUI states. Another technique that is popular for testing conventional software involves programming a test-case generator [16], in which the test designer develops software programs to generate test cases. This approach requires that the test designer encode all possible GUI decision points. Programming a test-case generator is thus time-consuming and may lead to a low quality set of test cases if important GUI decisions are overlooked.

Several prior research efforts have focused on finite-state machine (FSM) models have been proposed to generate test cases [4, 3, 7, 1]. 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 also been used extensively for test generation for testing hardware circuits [10]. For small sized software, it is easy to specify the software's behavior in terms of states. Another 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 pertinent to GUI testing, is that FSM models have scaling problems [30]. To aid in the scalability of the technique, variations such as variable finite state machine (VFSM) models have been proposed by Shehady et al. [30].

Test cases have also been generated to mimic novice users [15]. 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.

Finally, techniques have been proposed to reduce the total number of test cases either by focusing the test-case generation process on particular aspects of the GUI [7, 15, 17, 18] or by establishing an upper bound on the number of test cases [34]. Unfortunately, many of these techniques are not in common use, either because of their lack of generality or because they are difficult to use.

As mentioned before, AI planning has been previously used to generate test cases. In an earlier paper, we describe a preliminary version of the PATHS system, focusing on the software engineering aspects of the work [23]. Howe et al. describe a planning based system for generating test cases for a robot tape library command language [14]. Note that in this previous work, each command in the language was modeled with an 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, we had to modify the approach by automatically deriving hierarchical operators.

8 Conclusions

We have presented a new planning-based technique for generating test cases for GUI software, which can serve as a valuable tool in the test designer's tool-box. Our technique models test-case generation as a planning problem. The key idea is that the test designer is likely to have a good idea of the possible tasks of a GUI user, and it is simpler and more effective to specify these tasks in terms of initial and goal state than it is to specify sequences of events that achieve them. Our technique is unique in that we use an automatic planning system to generate test cases given a set of tasks and a set of operators representing GUI events. Additionally, we showed how hierarchical operators can be automatically constructed from a structural description of the GUI.

We have also provided initial evidence that our technique can be practical and useful, by generating test cases for the popular MS WordPad software's GUI. The experiments demonstrated the feasibility of the approach and also showed the value of using hierarchical operators for efficiently generating multiple plans for a specified task.

The use of hierarchical operators in test-case generation also aids in performing regression testing. Changes made to one part of the GUI do not necessarily invalidate entire test cases. Often, it is possible simply to perform a new decomposition of some abstract operator(s) in the high-level plan, and replace the prior decomposition with the new result. Finally, representing the test cases at a high level of abstraction also makes it possible to fine-tune the test cases to different implementation platforms, making the test suite more portable.

One of the tasks currently performed by the human 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 ways of automatically generate the preconditions and effects of the operators from a GUI's specifications. Additionally, we are using our plan-based approach throughout the larger PATHS system, which, in addition to test-case generation, performs such tasks as oracle creation for verification, automated execution of test cases, and test suite management for regression testing.

REFERENCES

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

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

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

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

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

[6] K. Erol, D. Nau, and J. Hendler. 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*, pages 20–23. AAAI Press, Menlo Park, California, 1993.

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

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

[9] J. Gray. 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.

[10] 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):935–945, July 1993.

[11] M. L. Hammontree, J. J. Hendrickson, and B. W. Hensley. Integrated data capture and analysis tools for research and testing on graphical user interfaces. In *Proceedings of ACM CHI'92 Conference on Human Factors in Computing Systems*, Demonstration: Analysis Tools/Multimedia Help, pages 431–432, 1992.

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

[13] J. H. Hicinbothom and W. W. Zachary. A tool for automatically generating transcripts of human-computer interaction. In *Proceedings of the Human Factors and Ergonomics Society 37th Annual Meeting*, volume 2 of *SPECIAL SESSIONS: Demonstrations*, page 1042, 1993.

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

[15] D. J. Kasik and H. G. George. Toward automatic generation of novice user test scripts. In M. J. Tauber, V. Bellotti, R. Jeffries, J. D. Mackinlay, and J. Nielsen, editors, *Proceedings of the Conference on Human Factors in Computing Systems : Common Ground*, pages 244–251, New York, 13–18 Apr. 1996. ACM Press.

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

[17] M. Kitajima and P. G. Polson. A computational model of skilled use of a graphical user interface. In *Proceedings of ACM CHI'92 Conference on Human Factors in Computing Systems*, Modeling the Expert User, pages 241–249, 1992.

[18] M. Kitajima and P. G. Polson. A comprehension-based model of correct performance and errors in skilled, display-based, human-computer interaction. *International Journal of Human-Computer Studies*, 43(1):65–99, 1995.

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

[20] J. Koehler, B. Nebel, J. Hoffman, and Y. Dimopoulos. Extending planning graphs to an ADL subset. In S. Steel and R. Alami, editors, *Proceedings of the 4th*

European Conference on Planning (ECP-97): Recent Advances in AI Planning, volume 1348 of *LNAI*, pages 273–285, Berlin, Sept.24 –26 1997. Springer.

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

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

[23] A. M. Memon, M. E. Pollack, and M. L. Soffa. Using a goal-driven approach to generate test cases for GUIs. In *Proceedings of the 21st International Conference on Software Engineering*, pages 257–266. ACM Press, May 1999.

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

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

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

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

[28] E. P. D. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proceedings of KR'89*, Toronto, Canada, pp 324-331, May 1989. (extended version submitted to Artificial Intelligence, special issue on KR'89).

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

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

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

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

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

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

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

[36] R. M. Young, M. E. Pollack, and J. D. Moore. 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.