Plan Generation for GUI Testing†

Atif M. Memon‡ and **Martha E. Pollack** and **Mary Lou Soffa**
Dept. of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260 USA
{atif, pollack, soffa}@cs.pitt.edu

Abstract

Graphical user interfaces (GUIs) have become nearly ubiquitous as a means of interacting with software systems. GUIs are typically highly complex pieces of software, and testing their correctness poses a large challenge. In this paper, we present a new approach to automatic testing of GUIs that builds on AI planning techniques. The motivating idea is that GUI test designers will often find it easier to specify typical goals that users of their software might have than to specify sequences of GUI actions that users might perform to achieve those goals. Thus we cast GUI test-case generation as an instance of plan generation: given a specification of initial and goal states for a GUI, a planner is used to generate sequences of GUI actions that lead from the initial state to the goal state. We describe our GUI testing system, PATHS (Planning Assisted Tester for grapHical user interface Systems), and we report on experiments using PATHS to generate test cases for Microsoft's WordPad.

Introduction

Graphical user interfaces (GUIs) have become nearly ubiquitous as a means of interacting with software systems. GUIs are typically highly complex pieces of software, and testing their correctness poses a large challenge (Myers 1993; Wittel, Jr. & Lewis 1991), due to the fact that the space of possible GUI interactions is enormous and that measures of coverage developed for conventional software testing do not apply well to GUI testing.

In this paper, we present a new approach, based on AI planning techniques, for partially automating GUI testing. The motivating idea is that GUI test designers will often find it easier to specify typical user goals than to specify sequences of GUI actions that users might perform to achieve those goals. The software underlying any GUI is designed with certain intended uses in mind; thus the test designer can describe those intended uses. It is typically harder to specify all the ways in which a user might interact with the GUI to achieve typical goals. Users interact in idiosyncratic ways, which the designer might not anticipate. Additionally, there can be a large number of ways to achieve any given goal, and it would be very tedious for the GUI tester to specify even those action sequences that s/he can anticipate.

We thus cast GUI test-case generation as an instance of plan generation, in which the human tester provides a specification of initial and goal states for likely uses of the system. An automated planning system then generates multiple plans for each specified planning problem. Each plan generated represents a test case that is, at least intuitively, a reasonable candidate for helping achieving good test coverage, because it reflects an intended use of the system. Of course, most systems can and will be used in ways other than those initially intended by the GUI designers. Thus, the test cases generated by our system will likely need to be supplemented with other test cases as is the usual case for any test-case generation scheme, for example, randomly generated sequences of GUI actions could also be used as test cases.[1]

Earlier work made similar use of planning, generating test cases for a robot tape-library command language (Howe, von Mayrhauser, & Mraz 1997). Our work extends the idea in several ways: by developing techniques for semi-automatically constructing the planning operators; by using hierarchical operators that reflect structural properties of GUIs and lead to increased efficiency; and by developing means for generating multiple, alternative plans for the same goal. We are also using our approach to automate other aspects of GUI testing, including verification, oracle creation and regression testing, but those topics are outside the scope of the current

† 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.

[1] Experimental analysis could be performed to measure the actual coverage of our test-case generation procedure, e.g., by comparing the plans it produces against actual sequences of GUI actions performed by users once it has been fielded. Such experimentation is deferred to future research.

paper.

In the next section, we briefly review the issues in GUI testing. We then describe our approach to planning GUI test cases. In particular, we show how the operators that represent GUI events can be created in a semi-automatic fashion and how AI planning techniques are used to generate test cases. We also discuss how our use of a restricted form of hierarchical planning leads both to an improvement in planning efficiency and to the generation of multiple alternative test cases. We have implemented our approach in Planning Assisted Tester for graphical user interface Systems (PATHS), and in the section entitled "Feasibility Experiments," we describe PATHS and report on the results of experiments in which it generated test cases for Microsoft's Word Pad system. Finally, we conclude with a summary of our main contributions to date and our plans for continued research.

GUI Testing and An Example

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 (White 1996). 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 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 (Kasik & George 1996).

Software 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

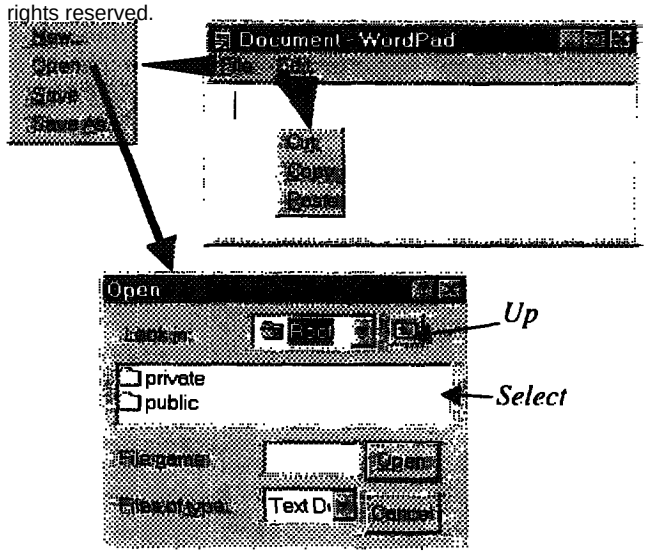

Figure 1: The Example GUI.

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 (Myers 1993). In this paper, we focus on the first step: test-case generation. The test-case generator that we describe here is part of a larger GUI testing system that we are building called PATHS. We plan to address other aspects of testing such as verification and regression testing as part of the overall design of PATHS in future work.

A GUI Example

Figure 1 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 1 is an example. In contrast, object windows display and manipulate other, non-GUI objects; an ex-

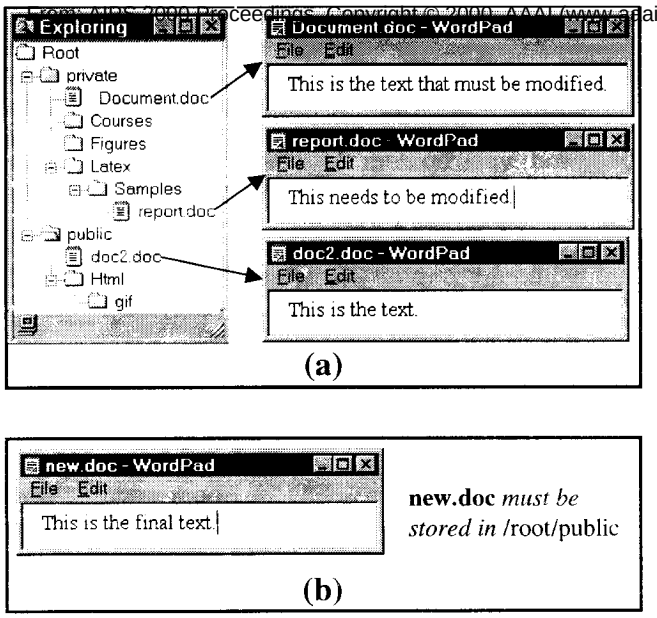

Figure 2: A Task for the Planning System; (a) the Initial State, and (b) the Goal State.

ample 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 all 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 2 illustrates a planning problem for a GUI 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.

Planning GUI Test Cases

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 an abstract model of a GUI and uses that to produce a list of operators representing GUI events. These are returned to the test designer, who uses his/her knowledge of the GUI to define the preconditions and effects of the operators; this constitutes the second step of the

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 $\mathcal{T}$

Iterate 3 and 4 for Multiple Scenarios

Table 1: Roles of the Test Designer and PATHS During Test-Case Generation.

setup phase. Next comes the plan-generation phase, in which the test designer first describes scenarios (tasks) by defining a set of initial and goal states. Finally, in step four of the overall testing process, PATHS generates a set of test cases for each scenario. 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.

Deriving GUI Operators

We now describe the setup phase. This 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 second, *abstract operators* are constructed to model GUI events that lead to sequences of GUI events which are themselves best viewed as "subplans". 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 2.

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

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 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.

The idea behind abstract operators can be clarified with an example. Figure 3 focuses again on a small part of the running example from Figure 1. 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 3(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

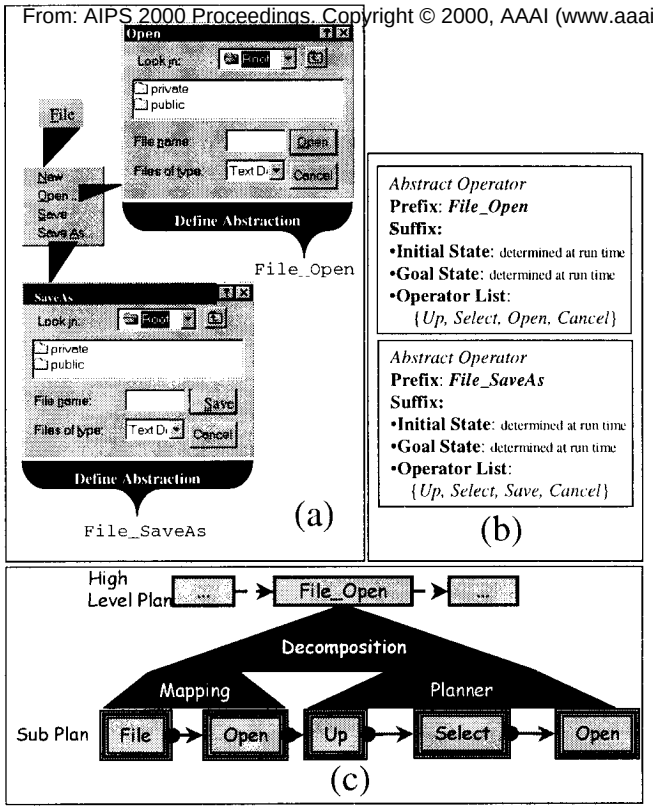

Figure 3: (a) Restricted Focus Operators: `Open` and `SaveAs` (b) Abstract Operators, and (c) Decomposition of the Abstract Operator

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 3 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

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 4: 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*) $\wedge$
DEL onScreen(*Obj*) $\wedge$
DEL Selected(*Obj*).

Figure 5: An Example of a GUI Planning Operator.

to the test designer for completion. The planning operators returned for the complete example in Figure 1 are shown in Figure 4(b).

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 5. `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

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 6: Initial State and Goal State Describing the Task of Figure 2.

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

Modeling the Initial and Goal States and Generating Test Cases

Once the setup phase has been completed, the second phase, 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 6 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. (This is Step 4 of the overall test-case generation process given in Table 1). An example of one such plan is shown in Figure 7.[2] This is a high-level plan that must be decomposed. Figure 8 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:

[2]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.

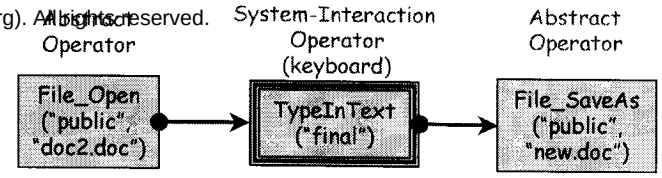

Figure 7: A Plan Consisting of Abstract Operators and a GUI Event.

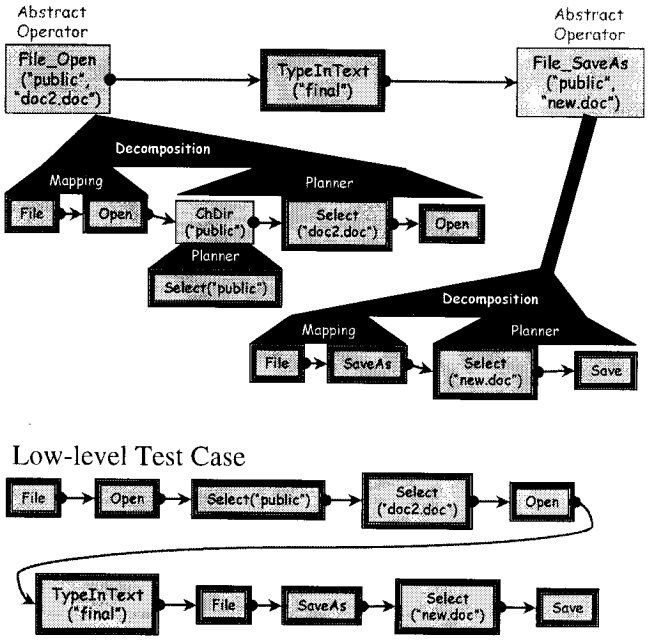

Figure 8: Expanding the Higher Level Plan.

- We run the planner several times, each time producing a distinct high-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. (See the section "Feasibility Experiments.") 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 9 shows an alternative plan created from the high-level plan in Figure 7. It differs from the decomposition in Figure 8 in that it uses a different decomposition of the first abstract operator.

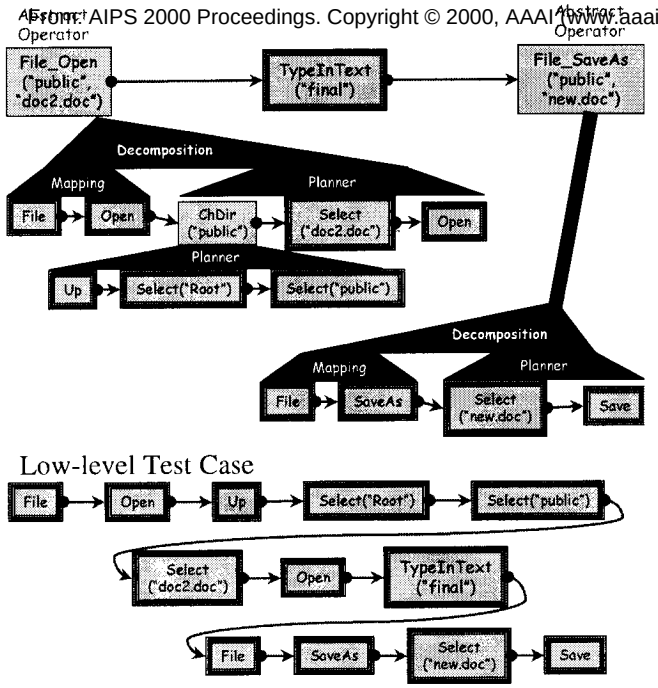

Figure 9: An Alternative Decomposition of the Abstract Operator Leads to a New Test Case.

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 10. The operators are assumed to be available before making a call to this algorithm, i.e., steps 1-3 of the test-case generation process shown in Table 1 must be completed before making a call to this algorithm. The parameters (lines 1..5 of the algorithm)[3] 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 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 (Koehler *et al.* 1997) as the underlying planning system. IPP was chosen based on the results of experiments in which causal-link planners were compared with propositional planners (Memon, Pollack, & Soffa 1999a). 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 3. The total number of GUI events in WordPad was determined to be approximately 325. After deriving hierarchical operators (step 1 of Table 1), 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

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

```
Algorithm :: GenTestCases(                                    Lines
  A = 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(A, D, I, G)) != NO_PLAN)                      8
     && (c < T) DO {                                           9
    InsertInList(p, planList);                                 10
    A ← RecordPlan(A, 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
        AC ← 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(AC, 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 10: The Complete Algorithm for Generating Test Cases

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

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 4 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 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 3 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 3 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.

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

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 record/playback tools (The 1992; Hammontree, Hendrickson, & Hensley 1992). 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 (Kepple 1994), 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 (Clarke 1998; Chow 1978; Esmelioglu & Apfelbaum 1997; Bernhard 1994). 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 (H. Cho, G.D. Hachtel, & F. Somenzi 1993). 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 (Shehady & Siewiorek 1997). To aid in the scalability of the technique, variations such as variable finite state machine (VFSM) models have been proposed by Shehady et al. (Shehady & Siewiorek 1997).

Test cases have also been generated to mimic novice users (Kasik & George 1996). 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 (Esmelioglu & Apfelbaum 1997; Kasik & George 1996; Kitajima & Polson 1992; 1995) or by establishing an upper bound on the number of test cases (White 1996). 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 (Memon, Pollack, & Soffa 1999b). Howe et al. describe a planning based system for generating test cases for a robot tape library command language (Howe, von Mayrhauser, & Mraz 1997). There are strong similarities between their approach and our own; major differences were described in the "Introduction" section. Note in particular 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.

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

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

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

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

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

H. Cho; G.D. Hachtel; and F. Somenzi. 1993. 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.

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

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

Husain, D., and George, H. G. 1996. Toward automatic generation of novice user test scripts. In Tauber, M. J.; Bellotti, V.; Jeffries, R.; Mackinlay, J. D.; and Nielsen, J., eds., *Proceedings of the Conference on Human Factors in Computing Systems : Common Ground*, 244–251. New York: ACM Press.

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

Kitajima, M., and Polson, P. G. 1992. 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, 241–249.

Kitajima, M., and Polson, P. G. 1995. 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.

Koehler, J.; Nebel, B.; Hoffman, J.; and Dimopoulos, Y. 1997. Extending planning graphs to an ADL subset. In Steel, S., and Alami, R., eds., *Proceedings of the 4th European Conference on Planning (ECP-97): Recent Advances in AI Planning*, volume 1348 of *LNAI*, 273–285. Berlin: Springer.

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

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

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

Shehady, R. K., and Siewiorek, D. P. 1997. 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)*, 80–88. Washington - Brussels - Tokyo: IEEE.

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

White, L. 1996. Regression testing of GUI event interactions. In *Proceedings of the International Conference on Software Maintenance*, 350–358. Washington: IEEE Computer Society Press.

Wittel, Jr., W. I., and Lewis, T. G. 1991. Integrating the MVC paradigm into an object-oriented framework to accelerate GUI application development. Technical Report 91-60-06, Department of Computer Science, Oregon State University. Fri, 15 Dec 1995 03:14:52 GMT.