Using a Goal-driven Approach to Generate Test Cases for GUIs

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
The widespread use of GUIs for interacting with software is leading to the construction of more and more complex GUIs. With the growing complexity comes challenges in testing the correctness of a GUI and the underlying software. In this paper, we present a new technique to automatically generate test cases for GUIs that exploits **planning**, a well developed and used technique in artificial intelligence. Given a set of operators, an initial state and a goal state, a planner produces a sequence of the operators that will change the initial state to the goal state. Our test case generation technique first analyzes a 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, extended to produce multiple plans. 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 also putting verification commands into 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, generating alternate plans

1 INTRODUCTION

Graphical User Interfaces or GUIs have become an important and accepted way of interacting with today's software. Although they make software easy to use from a user's perspective, they complicate the software development process. In particular, the testing of GUIs is more complex than testing conventional software, for not only does the underlying software have to be tested but the GUI itself must be exercised and tested to check for bugs in the GUI implementation. Even when tools are used to generate GUIs automatically, they are not bug free, and these bugs may manifest themselves in the generated GUI, leading to software failures. Hence, testing of GUIs continues to remain an important aspect of software testing.

Testing the correctness of a GUI is difficult for a number of reasons. First of all, the space of possible interactions with a GUI is enormous, in that each sequence of GUI commands can result in a different state, and a GUI command may need to be evaluated in all of these states. Secondly, there are many ways to use a GUI. The large number of input permutations [20] result in extensive testing needs, normally forcing software companies to release many copies of beta releases (almost 400,000 for Miscrosoft's Windows95 [8]) targeted at finding program failures. Another problem relates to determining the coverage of a set of test cases. For conventional software, coverage is measured using the amount and type of underlying code exercised. These measures do not work well for GUI testing, because what matters is not only how much of the code is tested, but in how many different possible states of the system each piece of code is tested. An important aspect of GUI testing is verification of its state at each step of test case execution. An incorrect GUI state can lead to an unexpected screen, making further execution of the test case useless since actions in the test case will not match the right buttons on the GUI screen. Thus, execution of the test case must be terminated as soon as an error is detected. Also, if verification checks are not inserted at each step, it may become difficult to identify the actual cause of the error. And lastly, regression testing presents special challenges for GUIs, because the input-output mapping does not remain constant across successive versions of the software [14].

An important component of testing is the generation of test cases. Manual creation of test cases and their maintenance, evaluation and conformance to coverage criteria are very time consuming. Thus some automation is necessary when testing GUIs. Current tools to aid the test designer in the testing process are capture replay tools [18, 6]. These tools 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. Several attempts have also been made to automate test case generation for GUIs. One popular technique is programming the test case generator [10]. However, this requires that the test designer program all possible decision points in the GUI. This approach is time consuming, and is susceptible to missing important GUI decisions. Other automation techniques include using variable finite state machines (VFSMs) [16]. Work has also been done to reduce the total number of test cases either by focusing the test generation process [5, 8, 11, 12] or by establishing an upper bound on the number of test cases [20]. Many of these techniques are not in common use either because of their lack of generality or because they are difficult to use.

In this paper, we present a new technique to automatically generate test cases for GUI systems. Our approach exploits planning techniques developed and used extensively in artificial intelligence (AI). The key idea is that the test designer is likely to have a good idea of the possible goals of a GUI user, and it is simpler and more effective to specify these goals than to specify sequences of actions that achieve them. Our test generation system takes these goals as input and generates sequences of actions that achieve these goals. These sequences of actions or "plans" become test cases for the GUI. Our testing system first performs an automated analysis of the hierarchical structure of the GUI actions to identify operators that will be used in the plan generation. The test designer next describes the preconditions and effects of these planning operators, which are then input to the planner. To generate test cases, a set of initial and goal states is input into the planning system, which applies hierarchical plan generation to produce multiple hierarchical plans. We implemented our technique and demonstrate its effectiveness and efficiency through a set of experiments.

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

- We make innovative use of a well known and used technique in AI, which has been shown to be capable of solving problems with large state spaces [9]. Combining the unique properties of GUIs and planning, we are able to demonstrate the practicality of the approach.
- Our technique exploits structural features present in GUIs to reduce the model size and complexity and improve the efficiency of test case generation.
- In contrast to earlier approaches where verification commands have to be inserted manually at points determined by the test designer, our system automatically inserts the verification commands at every step of the test case. This helps pinpoint the errors faster.
- Exploiting the structure of the GUI makes regression testing easier. Changes made to one part of the GUI do not affect the entire test suite. Most of our generated scripts are updated by making local changes.
- Platform specific details are incorporated at the very end, making the entire test suite portable. Portability assures that test cases written for GUI systems on one platform also work on other platforms [19].

The next section gives a brief overview of our system using an example GUI. Section 3 briefly reviews the fundamentals of AI plan generation. Section 4 describes how planning was applied to GUI test case generation problem. In Section 5 we describe a prototype system and give timing results for its execution. In Section 6 we discuss other work done on automated test case generation and conclude in Section 7.

2 OVERVIEW

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

Figure 1 present an simple example GUI for editing files, which is used throughout the paper. This GUI can be used for loading objects from files, manipulating these graphical objects (by cutting and pasting) and then saving the objects in another file. At the highest level, the GUI has a pull down menu with two buttons that can be clicked to open new buttons. For example the `File` button opens a menu with `Open`, `Merge`, `Save` and `SaveAs` buttons. The `Edit` button opens a menu with `Cut` and `Paste` buttons, which are used to cut and paste objects from the main screen. Each of `Open`, `Merge` and `SaveAs` buttons open windows with several more buttons. These are used to traverse the directory hierarchy and select a file. `Up` moves up one level in the directory hierarchy. `Select` is used to either enter subdirectories or select files. `Home` changes the current directory to the home directory. The window is closed by clicking on the `OK` button.

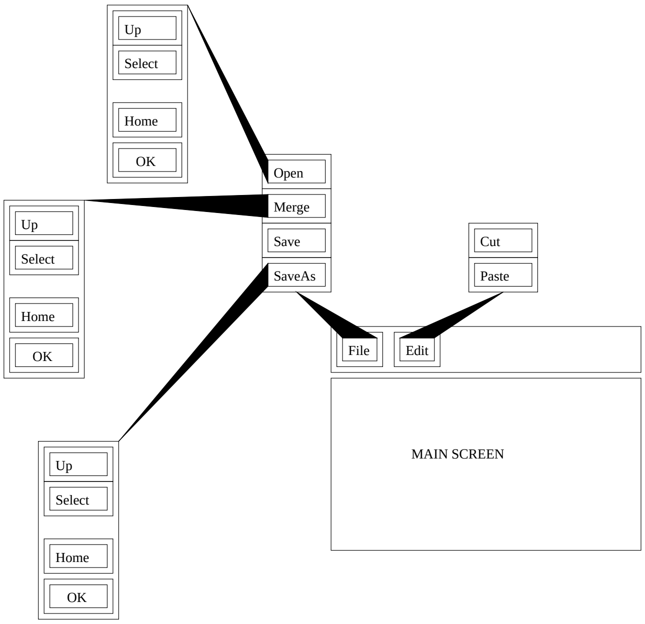

Figure 1: The Example GUI.

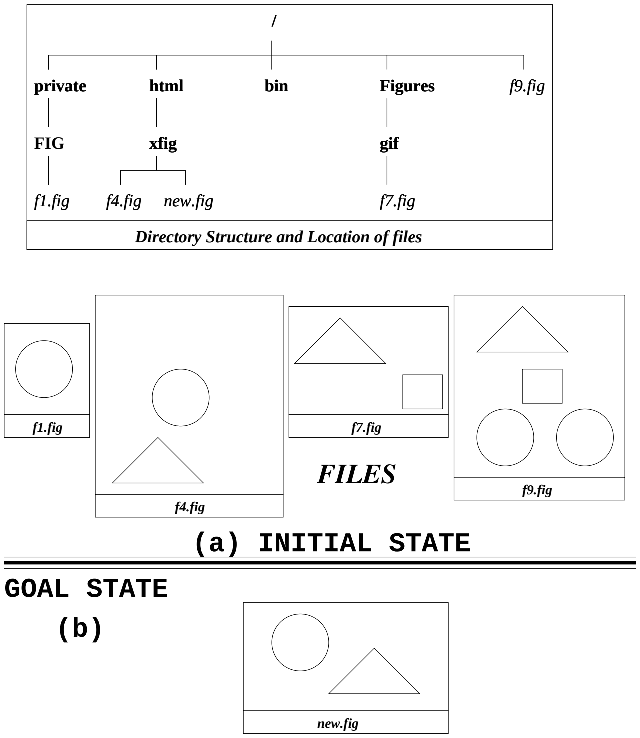

Figure 2: (a) Initial State and (b) Goal State.

The central component of our test case generation system is a planner. A planner is a set of algorithms and mechanisms developed to solve a planning problem. A planning problem is one in which a initial state, a goal state, a set of objects and a set of "operators" (or domain actions) are given. Operators are usually described in terms of preconditions and effects: conditions that must be true for the action to be performed, and conditions that will be true after the action is performed. A solution to a given planning problem is a sequence of instantiated operators that are guaranteed to result in the goal state when generated from the start state. In our example GUI, the operators relate to the buttons.

Consider Figure 2 (a), which shows the directory structure and a collection of files in a directory. Assume this to be the `initial state`. Using these files and the GUI, we define a goal of making a drawing containing two objects shown in Figure 2(b) and then storing it in file *new.fig*. This is the `goal state`. Note that *new.fig* can be obtained in numerous ways, e.g., loading file *f9.fig* and cutting the extra objects, by merging *f1.fig* and *f7.fig* and cutting the square, or by constructing the figure by cutting and pasting objects.

The roles of the test designer and the tasks automatically performed by the system are summarized in Table 1. The test case generation process is partitioned into two phases, the *setup* phase and *plan generation* phase. In the first step of setup, the testing system creates an abstract model of the GUI and returns a list of derived operators from the model to the test designer. The test designer then defines the preconditions and effects of the operators in a simple language provided by the planning system using knowledge about the GUI. During the plan generation phase, the test designer describes scenarios by defining a set of initial and goal states for test case generation. In the fourth step, our system generates a test suite for the scenarios. The test designer can iterate through the plan generation phase any number of times generating more test cases.

For our example GUI, the simplest approach in step 1 would be for the system to identify one operator for each directly executable GUI action (e.g., `Open`, `File`, `Cut`, `Paste`, etc.). These are called *primitive* operators and are in a one-to-one correspondence with the directly executable actions. Using this approach, the test designer would need to define the preconditions and effects for all the operators shown in column 1 of Table 2. (As a naming convention, we disambiguate with meaningful prefixes whenever operator names are the same, such as `Up` in Figure 1.) Although conceptually simple, this approach is inefficient for generating test cases for GUIs. This is because many of these operators merely make other operators available, but do not interact with the underlying software. These simple types of operators can be tested in isolation.

An alternative modeling scheme, and the one used in this work, models the domain hierarchically with high level operators that decompose into sequences of lower

Phase	Step	Test Designer	Automatic System
`Setup`	1		Derive Hierarchical GUI Operators
	2	Define Preconditions and Effects of Operators	
`Plan Generation`	3	Identify a Task	
	4		Generate Test Cases

Iterate 3 and 4 for Multiple Scenarios

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

level ones. Although high level operators could be developed manually by the test designer, our system avoids this inconvenience by automatically performing most of the abstraction. More specifically, the modeling process begins with *primitive* operators, and uses certain structural properties of GUIs to generate additional levels of operators. The first of these are *intermediate* operators which are composed of a sequence of primitive operators. For example, our system would define an intermediate operator `EDIT_CUT` to be composed of a sequence of clicks on `EDIT` and `CUT`, i.e., a mapping `EDIT_CUT = <EDIT, CUT>`. Note that intermediate operators can also be viewed as macros. This approach of modeling prevents generation of test cases that contain the `Edit` action in isolation whose only effect is to open a menu containing `Cut` and `Paste`. Examples of other intermediate operators in the example GUI are `EDIT_PASTE` and `FILE_SAVE`.

The second set of operators generated by our system are *abstract* operators. These will be discussed in more detail in Section 4. The basic idea is that these are operators that need to be expanded by a later call to the planner. Abstract operators for our example GUI include `LOAD_FILE`, `COMBINE_FILE` and `STORE_FILE`. The result of the first step of the setup phase is a set of operators that are returned to the test designer. The hierarchical operators returned for our example are shown in column 2 of Table 2. In order to keep a correspondence between the original GUI and these modeled operators, the system also stores mappings, given in Table 3.

The test designer then specifies the preconditions and effects for each derived operator. An example of a planning operator, `EDIT_CUT` is shown in Figure 3. `EDIT_CUT` is an intermediate operator obtained by combining the `Edit` and `Cut` actions. The operator contains two parts,

Primitive Operators	Derived Operators
FILE	LOAD_FILE
EDIT	COMBINE_FILE
OPEN	STORE_FILE
MERGE	EDIT_CUT
SAVE	EDIT_PASTE
SAVEAS	FILE_SAVE
CUT	
PASTE	
OPEN.UP	
OPEN.SELECT	
OPEN.HOME	
OPEN.OK	
MERGE.UP	
MERGE.SELECT	
MERGE.HOME	
MERGE.OK	
SAVEAS.UP	
SAVEAS.SELECT	
SAVEAS.HOME	
SAVEAS.OK	
SAVE	

Table 2: Operators for the Example GUI.

Operator Name	Expansion
LOAD_FILE	<File, Open, ..., OK>
COMBINE_FILE	<File, Merge, ..., OK>
STORE_FILE	<File, SaveAs, ..., OK>
FILE_SAVE	<File, Save>
EDIT_CUT	<Edit, Cut>
EDIT_PASTE	<Edit, Paste>

Table 3: Mappings of Intermediate Operators to Low Level Operators.

preconditions and effects. All the components in the preconditions must hold in the GUI before the operator can be applied, e.g., for the user to click on the `Cut` button, the current menu should be at the second menu level, i.e. `Menu2`, as shown in Figure 3. One of the effects of the `Cut` action is that the selected objects are moved to the clipboard. The language used to define each operator is provided as an interface to the planning system. Specifying the preconditions and effects is not difficult as this knowledge is already built into the GUI. For example, the GUI specifications require that `Cut` and `Paste` be made active (visible) only after `Menu2` is opened. This is precisely the precondition defined for our example operator (`EDIT-CUT`) in Figure 3. The test designer can also further reduce the set of operators and define more abstractions by making use of properties specific to the GUI being tested and by employing domain information.

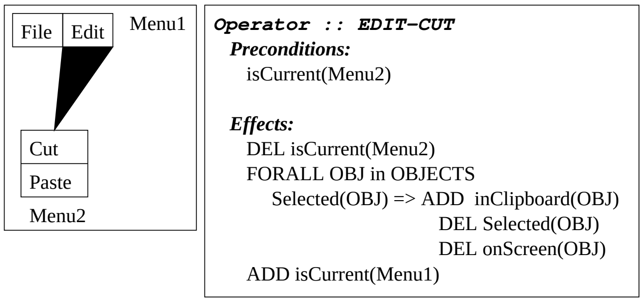

Figure 3: A GUI Example of Planning Operators.

The test designer begins the test case generation process by identifying a task (initial and goal states) and inputs the defined operators into the planning system. The system automatically generates a set of test cases that achieve the goal. Also inserted in the test cases are validation statements that help verify the state of the GUI at each step. An example of a plan for the task in Figure 2 is shown in Figure 4. This is a high level plan that must be translated into primitive GUI actions. The translation process makes use of the mappings stored during the modeling process. One such translation or test case is shown in Figure 5. Note the abstract operators `OP1` and `OP2`, which will be decomposed by further planning. Since the maximum time is spent in generating the high level plan, it is desirable to generate a family of test cases from this single plan. This is achieved by generating alternate sub-plans at lower levels. These sub-plans are generated much faster and can be substituted into the high level plan to obtain alternate test cases. One such alternate low level test case generated from the same plan is shown in Figure 6. Such a hierarchical mechanism also makes regression testing faster, since changes that are made to one component does not invalidate all test scripts. The higher level plans can still be retained and local changes can be made to scripts specific to the changed component of the GUI.

Figure 4: A Plan for the Goal.

3 PLAN GENERATION

We now provide some details on plan generation. Automated plan generation has been widely investigated and used within the field of artificial intelligence. Given an initial state, a goal state, a set of operators, and a set of objects, a planner returns a set of steps (instantiated operators) to achieve the goal. Many different algorithms for plan generation have been proposed and developed. The interested reader can consult [17] for

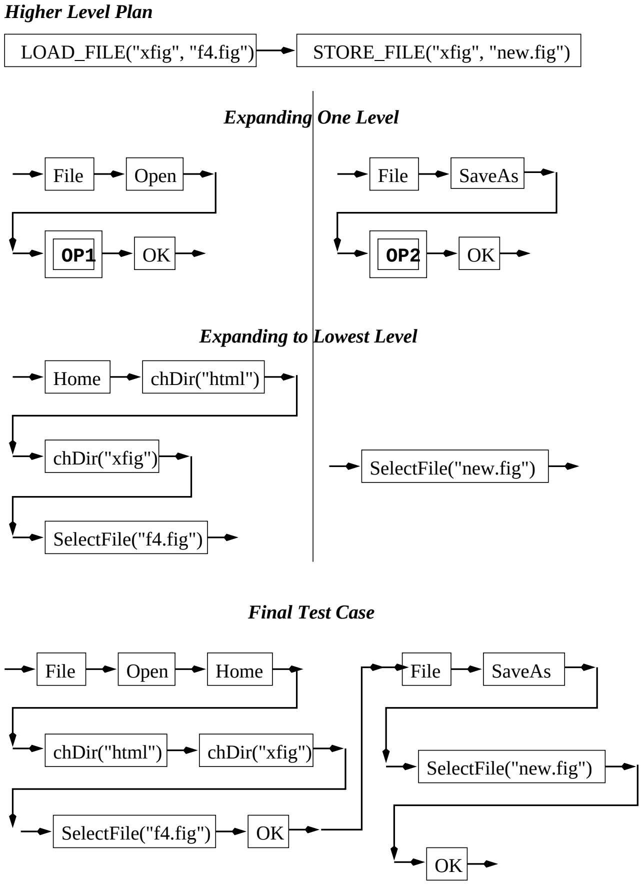

Figure 5: Expanding the Higher Level Plan.

examples of recent work in the field.

In this work, we employed a recently developed planning technology that increases the efficiency of plan generation. Specifically, we generate single level plans using the Interference Progression Planner (IPP) [13], a system which extends the ideas of the Graphplan system [1] 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 that rely on the full propositional representation and a graph search requiring unification of unbound variables.

IPP use a standard representation of actions in which preconditions and effects can be parameterized: subsequent processing does the conversion to propositional

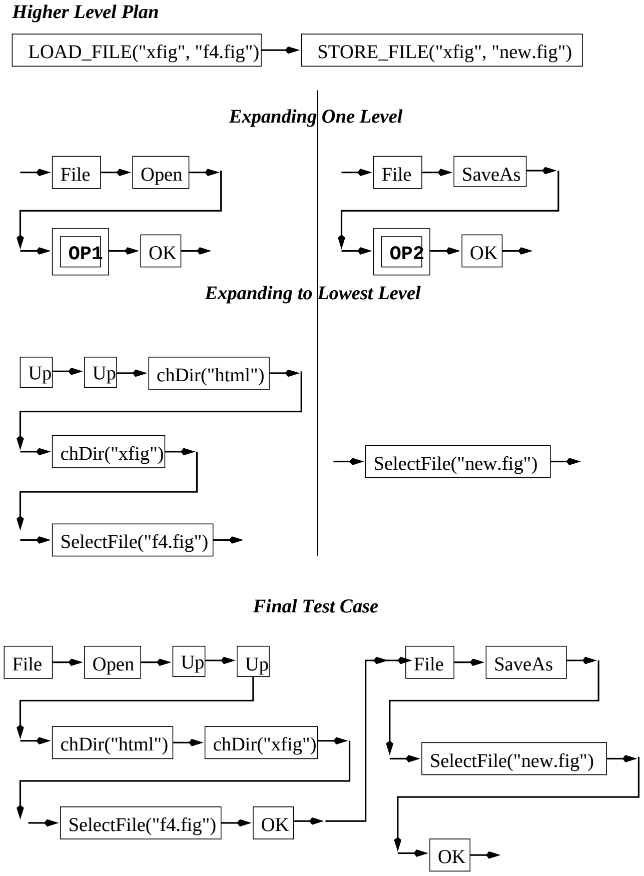

Figure 6: An Alternate Expansion Leads to a New Test Case.

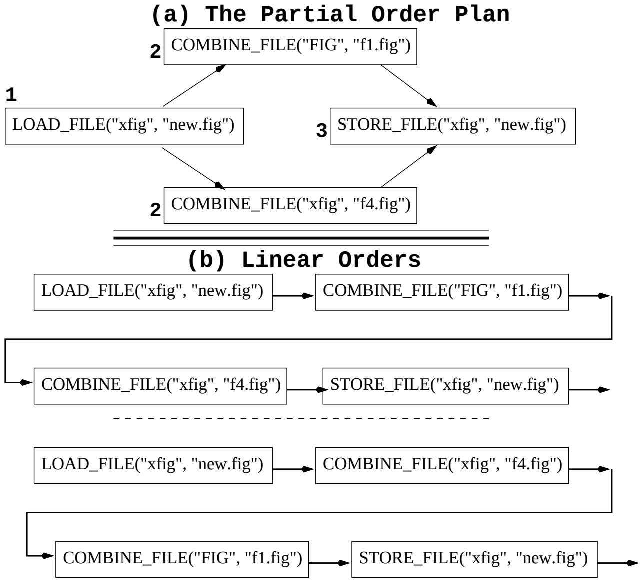

Figure 7: A Partial Order Plan and its Two Linearizations.

form.[1] As is common in planning, IPP produces partial order plans. A partial order plan is a solution to a planning problem if and only if every consistent linearization of the partial order plan meets the solution conditions. Figure 7 (a) shows the partial order plan obtained to realize the goal shown in Figure 2 using our example GUI. The steps are labeled with numbers that indicate the order of plan execution. At any time i, all steps labeled i can be performed in any order with respect to one another but must precede all operators at time $i+1$. For example, the two `COMBINE_FILE` actions (labeled 2), can be performed in either order, but they must precede the `STORE_FILE` action (labeled 3) and must be performed after the `LOAD_FILE` action (labeled 1). We obtain two legal orders, both of which are shown in Figure 7 (b), and thus two high level test cases that may be expanded to yield a number of low level test cases.

IPP forms plans at a single level of abstraction. We have extended this to hierarchical planning [2] which is valuable for GUI test case generation for several reasons. Firstly, since GUIs tend to be large, the use of a hierarchy allows us to decompose it into parts at different levels of abstraction, resulting in greater efficiency. Secondly, decomposition of the GUI results in generating plans for each level individually. Changes to one component of the GUI does not invalidate all the test cases. In fact, most of the test cases can be retained. Changes need to be made only to the test cases specific to the modified component, aiding regression testing.

One final point concerns the generation of alternative plans. As noted earlier, one of the main advantages of using the planner in this application is to automatically generate alternative plans for the same goal. AI planning systems typically generate only a single plan; the assumption is that the heuristic search control rules will ensure that this is a high quality plan. (but *cf.* [21]).

In our system, we generate alternative plans in the following two ways.

1. Linearizing the partial order plans. The order of some of the actions are not specified in the partial order plan. We are free to choose any linear order. All possible linear orders of a partial order plan result in a family of test cases.

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

[2] Techniques have been developed in AI planning at multiple levels of abstraction; this is typically called Hierarchical Task Network (HTN) planning [4, 3]. In HTN planning, domain actions are modeled at different levels of abstraction, and for each operator at level n, one specifies one or more "methods" at level $n-1$. A method is a single-level partial plan, and we say that an action "decomposes" into its methods. HTN planning focuses on resolving conflicts among alternative methods of decomposition at each level. In our work to date, we have avoided the need for such conflict resolution.

2. Iterating during planning and generating different test cases at each iteration.

4 PLANNING GUI TEST CASES

In developing a planning system useful for testing GUIs, the first step is for the system to identify operators. The simplest approach is to define one operator for each directly executable GUI action. However, as we argued earlier, there are advantages to forming the test case plans in a hierarchical fashion. In our system, we begin with *primitive* operators, which stand in a one-to-one correspondence with the directly executable actions of the GUI. Our system then analyzes the the primitive operators, using structural properties of GUIs, as described below, to generate two additional levels of operator. The *intermediate* operators are expanded like macros to obtain primitive operators, while the *abstract* operators are replaced with a call to the planner, yielding subplans.

To generate intermediate operators, our system first identifies primitive *expansion operators*. These are operators whose only effect is to make available other operators, i.e., they expand the set of actions available to the user. By definition, expansion operators do not interact with the underlying software. The most common example of expansion operators are buttons that generate pull down menus. All other primitive operators will be called *interaction operators* as they interact with the underlying software; common examples include buttons for cutting or pasting text.

The first step in producing the operator hierarchy is to identify all sequences of primitive expansion operators supported by the GUI, and to construct an intermediate level operator for each such sequence. For simplicity, consider a small part of our example GUI, one pull down menu with one option (`Edit`) which can be opened to give more options, i.e., `Cut` and `Paste`. The primitive actions available to the user are `Edit`, `Cut` and `Paste`. We identify `Edit` to be an expansion operator and `Cut` and `Paste` to be interaction operators. Using this information we obtain the following two intermediate operators.

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

Although new operators have been defined, these actually reduce the size of the overall operator set made available to the planner. The system hides `Edit`, `Cut` and `Paste`, making only the intermediate operators namely `EDIT_CUT` and `EDIT_PASTE` available to the planner. These intermediate operators are later replaced by the primitive operators when generating the final test case. This model prevents generation of test cases in which `Edit` is used in isolation, i.e., the model forces the use of `Edit` either with `Cut` or with `Paste`. In order to provide a meaningful interaction with the underlying software, either `Cut` or `Paste` must be used immediately after `Edit`. Test cases in which `Edit` stands in isolation can be handled separately.

Next, our system generates *abstract* operators by identifying expansion operators that have a special property, i.e., once invoked, they monopolize the GUI interaction, limiting the focus of the user to a specific range of other actions until such time as they are explicitly terminated. An example is preference setting in many GUI systems: the user clicks on `Edit` and `Preferences`, then spends an indefinite period of time modifying the preferences, and finally explicitly terminates the interaction by either clicking `OK` or `Cancel`. We can model a complete such interaction with an abstract operator that decomposes into three parts: an expansion operator, followed by an explicit call to the planner, the result of which represents the actions a user might have during the focused interaction, followed by a primitive termination operator. Note that the expansion operator may itself be at the intermediate level. The abstract operator is a complex structure since it contains all the necessary components of a planning problem. These include the initial and goal states, the set of objects and the set of operators.

Again consider a small part of the example GUI, a menu with two options, namely `Open` and `SaveAs`. Opening each of these, we get further options that are quite similar. In both cases, we can exit after pressing `OK`. The primitive operator set available is `Open`, `SaveAs`, `Open.Select`, `Open.Up`, `Open.Home`, `Open.OK`, `SaveAs.Select`, `SaveAs.Up`, `SaveAs.Home` and `SaveAs.OK`. Once the user selects `Open`, the focus is limited to `Open.Select`, `Open.Up`, `Open.Home` and `Open.OK`. Similarly, when the user selects `SaveAs`, the focus is limited to `SaveAs.Select`, `SaveAs.Up`, `SaveAs.Home` and `SaveAs.OK`. These observations lead the system to define the following two abstract operators.

```
LOAD_FILE(args)   = <Open, IPP(args), OK>
SAVEAS_FILE(args) = <SaveAs, IPP(args), OK>
```

`IPP(args)` is an explicit call to the planner that returns a subplan. The parameters `args` to the planner contain all the essential components of the planning problem, i.e., an initial state, determined dynamically at the point before the call, the goal state, which is the desired GUI state after the call, and the set of operators that are available during the limited focused user interaction. In this case the set of operators is `SaveAs.Select`, `SaveAs.Home` and `SaveAs.Up`.

The final set of operators given to the planner is constructed from the abstract, intermediate and a subset of the primitive operators. The choice of which primitive operators to make available to the planner is made during the modeling phase. First, all the primitive operators that were used to compose intermediate operators are not made available. Also, abstract operators hide lower level GUI details, also hiding the primitive operators. These are also not made available at this level of abstraction.

Now we can see how the complete planning system operates. The operators are assumed available before making a call to this algorithm. Figure 8 shows the algorithm for the test generation system. The parameters (lines 1..5) include all the components of a planning problem and a threshold (**T**) that controls the looping in the algorithm. The loop (lines 8..12) contains the explicit call to the **IPP** planner while modifying the operator set once in the loop. At the end of this loop, **planList** contains the entire partial order plan set. Each partial order plan is then linearized (lines 13..16), leading to multiple linear orders. Initially the test cases are just high level linear plans (line 17). The decomposition process leads to lower level test cases. Each step of the plan needs to be expanded to get lower level test cases. If the step is an intermediate operator, then the mappings are used to expand it (lines 20..22). However, if the step is an abstract operator, then it is expanded to a test case on its own and substituted into this higher level plan (lines 23..32). Note the use of extraction functions to access the planning problem's components at lines 24..27. The lowest level test cases are returned as a result of the algorithm (line 33).

```
       ALGORITHM :: GenTestCases(
1, 2     S = Operator Set; G = Goal State;
3, 4     I = Initial State; O = Object Set;
5        T = Threshold) {

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

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

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

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

Figure 8: The Complete Algorithm for Generating Test Cases

5 EXPERIMENTS

We conducted several experiments to ensure that our system is practical and useful. These experiments were run on an Ultra SparcStation running UNIX System V. We summarize the results of some of these experiments in the following paragraphs.

We used our system to generate test cases for a commonly used software GUI, namely Microsoft's WordPad. Some of the generated high level test cases are shown in Table 4. The original number of primitive operators were determined to be approximately 325. The hierarchical model resulted in 32 operators at the highest level of abstraction, i.e., roughly a ratio of 10 : 1. This reduction in the number of operators is impressive and helps speed up the plan generation process. Our earlier experiments compared the performances of the hierarchical and single level approaches and found that the single level approach takes much longer to generate test cases than the hierarchical approach.

Defining preconditions and effects for the 32 operators was fairly straightforward. The average operator size was 5 lines, with the most complex operator requiring 10 lines of code. For the test generation phase, we defined three additional operators for mouse and keyboard events. The results of some of the runs are shown in Table 5. Each row presents CPU execution times for a different test scenario. The average time required to generate a test case was quite low. These results show that the maximum time is spent in generating the high level plan (column 2). This same plan is then used to generate a whole family of test cases by substituting alternate low level sub-plans. These sub-plans are generated relatively faster (column 3 shows the time spent to decompose each subplan once). This helps amortize the cost of plan generation over multiple test cases. Plan 8, which took the longest time to generate, was linearized to obtain 2 high level plans, each of which were decomposed to give several low level test cases, the shortest of which consisted of 25 actions.

Plan Step	Plan Action
1	FILE-OPEN("private", "Document.doc")
2	DELETE-TEXT("muust")
3	TYPE-IN-TEXT("must", Times, Italics, 12pt)
4	FILE-SAVEAS("Samples", "doc2.doc")
1	FILE-OPEN("private", "report.doc")
2	SELECT-TEXT("this")
3	FORMAT-FONT("this", Times, Italics, 12pt)
4	TYPE-IN-TEXT("must", Times, Normal, 12pt)
4	DELETE-TEXT("needs to")
5	FILE-SAVEAS("Samples", "report.doc")
1	DELETE-TEXT("This is")
2	SELECT-TEXT("example")
3	FORMAT-FONT("example", Times, Bold, 14pt)

Table 4: Some WordPad Plans Generated for Different Goals.

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

Table 5: Average Time Taken to Generate Test Cases for WordPad.

The plans shown in Table 4 are at a high level of abstraction. Changes made at lower levels of the GUI have no effect on these plans, making regression testing easier and less expensive since the GUI test cases are still valid. For example, none of the plans in Table 4 contain any low level physical details of the GUI. Changes made to fonts, colors, etc. do not affect the test suite in any way. Changes that modify the functionality of the GUI can also be readily incorporated. For example, if the WordPad GUI is modified such that additional file opening features are introduced, then most of our high level plans remain the same. Changes are needed to subplans that are generated by the abstract operator `FILE-OPEN`. Hence the cost of initial plans is amortized over a large number of test cases.

We are also analyzing the much larger GUI of Microsoft Word. The automatic modeling process has been successful in reducing the number of primitive operators to a ratio of 20 : 1.

6 RELATED WORK

Planning has been found to be useful in generating focused test cases [7] for a robot tape library command language. However, the approach used there, of modeling each command as an operator, has scaling problems for GUI testing. Also, no specific algorithms were presented to generate alternate cases.

In an attempt to improve coverage and guide test case generation, [5] suggests that the test designer inserts artificial constraints in the domain model. The constraints must be manually inserted, which also slows down the test case generation time.

Latin square method [20] for test case generation is used to reduce the number of test cases for GUI testing. The underlying assumption is that it is enough to check pairwise interactions between components of the GUI requiring that each menu item appears in at least one test case. This assumption appears insufficient for GUIs since the sequence of actions performed on the GUI may result in a new state, many of which need to be exercised.

Test cases have been generated to mimic novice users [8]. The approach is to use an expert to generate the initial plan manually and then use genetic algorithm techniques to generate longer paths. The assumption is that experts take a more direct path when solving a problem using GUIs whereas novice users often take longer paths, deviating from the shortest. Although useful for generating multiple scripts, the technique relies on an expert to generate the initial script. The final test case suite depends largely on the paths taken by the expert user.

A finite state machine (FSM) based modeling approach is suggested in [2]. However, FSM models have been found to have scaling problems when applied to GUI test case generation. Slight variations such as variable finite state machine (VFSM) models have been proposed in [16]. These require that verification checks be inserted manually at points determined by the test designer.

One way to improve coverage is to introduce sets of data and path variations. In [15] path variations and data variations are defined by the test designer. One disadvantage is that this technique puts too much burden on the test designer. The overall coverage is influenced by the test designer.

7 CONCLUSIONS

A new approach is presented to generate test cases for GUI system validation. The method is based on plan generation techniques from AI to produce test cases that represent patterns of user interaction with the GUI. Our experiments indicate that our approach is practical.

Several coverage measures for GUIs can be developed from our technique. *Operator coverage* defines the percentage of GUI actions that were used in the test cases. Also, since there are many possible plans to achieve a task, and the test cases are generated from a subset of these, we measure *plan coverage* as the percentage of plans that were used in test case generation.

Since the model exploits the structural properties of GUIs, regression testing is made easier. Changes to parts of the GUI do not affect the entire test suite. These changes can be made locally and incorporated in the test suite automatically. These observations are consistent with our plans obtained in Section 5.

We plan to explore the possibility of inferring some of the preconditions and effects automatically from the GUI. We are also doing more experiments with larger GUIs to determine if more powerful modeling techniques are useful which exploit additional features of GUIs.

REFERENCES

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

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

[3] 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.

[4] 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.

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

[6] 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.

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

[8] 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.

[9] H. Kautz and B. Selman. The role of domain-specific knowledge in the planning as satisfiability framework. In R. Simmons, M. Veloso, and S. Smith, editors, *AIPS 98, Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems*, pages 181–189, 1998.

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

[11] 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.

[12] 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.

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

[14] 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.

[15] T. Ostrand, A. Anodide, H. Foster, and T. Goradia. A visual test development environment for GUI systems. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA-98)*, volume 23,2 of *ACM Software Engineering Notes*, pages 82–92, New York, Mar.2–5 1998. ACM Press.

[16] 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.

[17] R. Simmons, M. Veloso, and S. Smith, editors. *Proceedings of the Fourth International Conference on Atrificial Intelligence Planning Systems*, Pittsburgh, PA, June 1998. AAAI Press.

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

[19] A. Walworth. Java GUI testing. *Dr. Dobb's Journal of Software Tools*, 22(2):30, 32, 34, Feb. 1997.

[20] 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.

[21] M. Williamson and S. Hanks. Optimal planning with a goal-directed utility model. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems*, pages 176–181, 1994.