Using Tasks to Automate Regression Testing of GUIs

Atif M. Memon
Department of Computer Science
& Institute for Advanced Computer Studies
& Fraunhofer Center for Experimental Software Engineering
University of Maryland, College Park, Maryland, USA
email: atif@cs.umd.edu

ABSTRACT
Graphical User Interfaces (GUIs) present unique challenges for software testing. In this paper, we demonstrate that a test suite originally used to test a GUI contains a large number of *unusable* test cases for the modified GUI. We present a novel technique to recreate unusable test cases by associating meta-information (called a *task*) with each test case. Tasks represent activities that can be performed by using the software. The sequence of events in a test case represents the actions needed to complete its associated task. Even when changes to the GUI make test cases unusable, many tasks remain valid across successive GUI versions. We experimentally show that our technique is able to automatically and efficiently regenerate a large number of test cases.

KEY WORDS
Regression testing, AI planning, GUI testing

1 Introduction

A GUI test case contains two parts: (a) an *initial GUI state* S_0 in which the test case is executed, and (b) a sequence of events $e_1; e_2; \ldots; e_n$ which is the test input to the GUI [12]. The initial state is used to initialize the GUI to a desired state before events are executed on it. A change in the GUI may render some of the test cases useless, either because they specify an unreachable initial state or an incorrect sequence of events for the modified GUI. Such test cases are called *affected* and cannot be executed on the modified GUI. In practice, since a large number of the original test cases cannot be reused, GUI regression testing requires redeveloping new test cases from scratch, expending considerable resources [18].

In this paper, we reuse affected test cases by associating meta-information (called a *task*) with each test case. Tasks are activities that can be performed by using the software. The sequence of events in a test case represents the actions needed to complete its associated task. An example of a task for a drawing program may be: "duplicate an object", i.e., given a GUI in a state $S_0 = \{circle, C_1\}$, we want a state $S_1 = \{two\ circles, C_1, C_2\}$. A commonly used sequence of events needed to achieve this task would be $<$ $select(C_1)$, $copy(C_1)$, $paste(C_2)$ $>$. Even when changes to the GUI make test cases unusable, tasks remain valid across successive GUI versions. In the above example, even if $copy$ and $paste$ were removed from the GUI and replaced with a $duplicate$ event, the task would remain valid even though the sequence of events needed to perform it would change. The test designer maintains a *task pool*, modifying it when the GUI changes. The test designer labels tasks as *new*, *obsolete*, or *continuing* for the modified GUI. We represent tasks using an initial and goal state pair. In prior work we have used tasks in a system called Planning Assisted Tester for grapHical user interface Systems (PATHS) [13, 17] that employs **AI planning** to generate test cases for GUIs. We now leverage our experience to show how maintaining these tasks can be used for automatic GUI regression testing.

In the next section, we give an overview of the design of the regression tester and our replanning technique by using an example. In Section 3, we give a brief overview of AI planning. In Section 4, we formally define a GUI test case. The detailed design of the regression tester is described in Section 6. We then describe experiments and their results in Section 7. Finally, we conclude in Section 9 with a discussion of ongoing and future work.

2 Overview

Figure 1 shows a high-level view of the regression tester. The inputs are the original test suite, generated to test the original GUI, representations of both the original and modified GUIs, and the task pool. The outputs are the test case that need to rerun and discarded test cases. The key components of the regression tester include:

- **Test case selector** that partitions the original test suite into (1) unaffected test cases, (2) test cases for obsolete tasks, and (3) test cases that are affected.
- **Planning-based test case regenerator** that uses planning to regenerate the affected test cases that have an illegal event sequence. If successful, then the regenerated test case is used for regression testing; otherwise, if the planner fails to find a plan, then the task is obsolete and the test case is discarded.

Details of the design of each of the above components is presented in Section 6. We now give an overview of

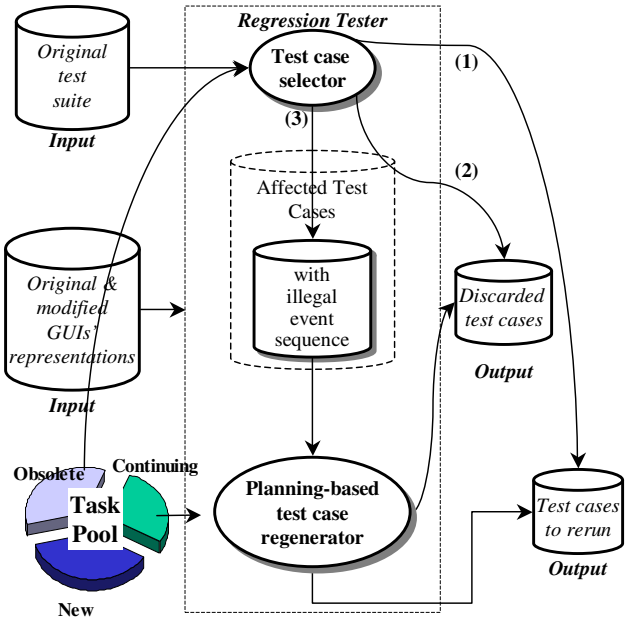

Figure 1. A High-level Overview of the Regression Tester.

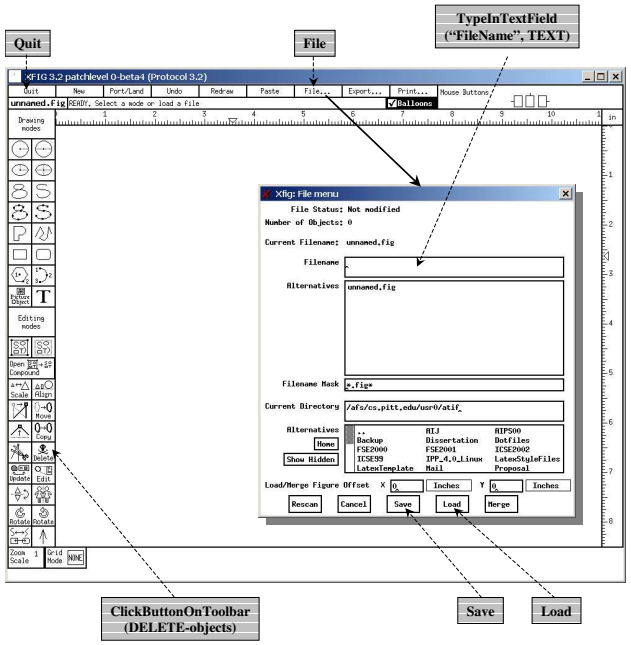

Figure 2. The Original Software (`Xfig 3.2 patch-level 0-beta4 (Protocol 3.2)`).

their operation by taking an example of the popular `xfig` software available on most Unix platforms[1]. We use versions `Xfig 3.2 patchlevel 0-beta4 (Protocol 3.2)` and `Xfig 3.2 patchlevel 3c (Protocol 3.2)`, which we will refer to as the *original* and *modified* software respectively. The original software is shown in Figure 2. Note that the shaded boxes are not a part of the GUI. They represent labels that we have used to represent events in the GUI. Dashed lines show the relationships between these names and the corresponding events. For example, the shaded box `TypeInTextField` represents an event used to enter text in the text-field. Solid line arrows show the relationships between windows and the events used to open them. For example, the event `File` invokes the window entitled "`Xfig: File menu`".

Consider the test case, generated for the original software, shown in Figure 3[2]. The test case uses a new event `ClickOnObject(OBJECT)`, which translates to clicking the left mouse button on an `OBJECT`. Note that we have represented events using the function notation with parameters. The exact representation details are discussed later in Section 4. The test case first launches `xfig`, loads a file named `original.fig`, cuts a circle object, saves the resulting drawing in `new.fig`, and exits the application. An example of running this test case using a specific instance of the file `original.fig` is shown in Figure 4.

Now consider the modified software shown in Figure 5. The modified software contains most of the functionality of the original software. Specifically, a user is able to load/save files and delete objects. A test designer perform-

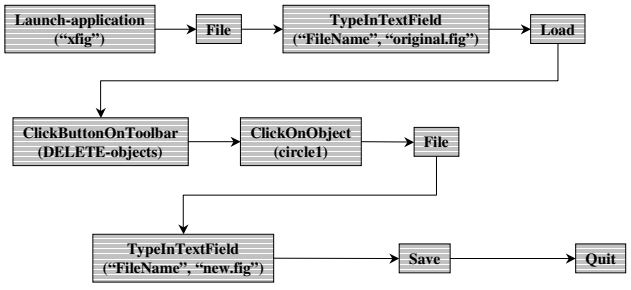

Figure 3. Test Case for the Original Software.

ing regression testing on the modified software would need to verify whether the tested functionality that was available in the original software works correctly in the modified software. However, examining the test case of Figure 3 shows that it cannot even be executed on the modified software; performing `File` opens a pull-down menu, (not the window as originally done) preventing event `TypeInTextField` from being performed. Hence the modifications have made the test case of Figure 3 useless for the modified software because the event sequence of the test case has become illegal. The test case selector marks this test case as "affected" so that it can be regenerated by the planning-based test case regenerator. From our prior experience of using `xfig`, we know that the modified software can be used to perform the task of Figure 4. If we were to perform the task manually, we would need to adapt the event sequence to the modified software. The test case needed to achieve the same task using the modified soft-

[1] `http://www.xfig.org/`

[2] The initial state is not shown for space reasons.

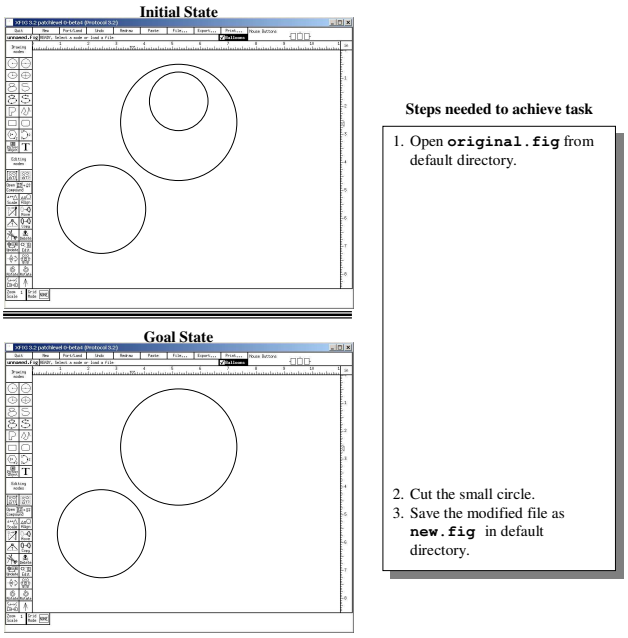

Figure 4. Running the Test Case on a Specific Instance of `original.fig`.

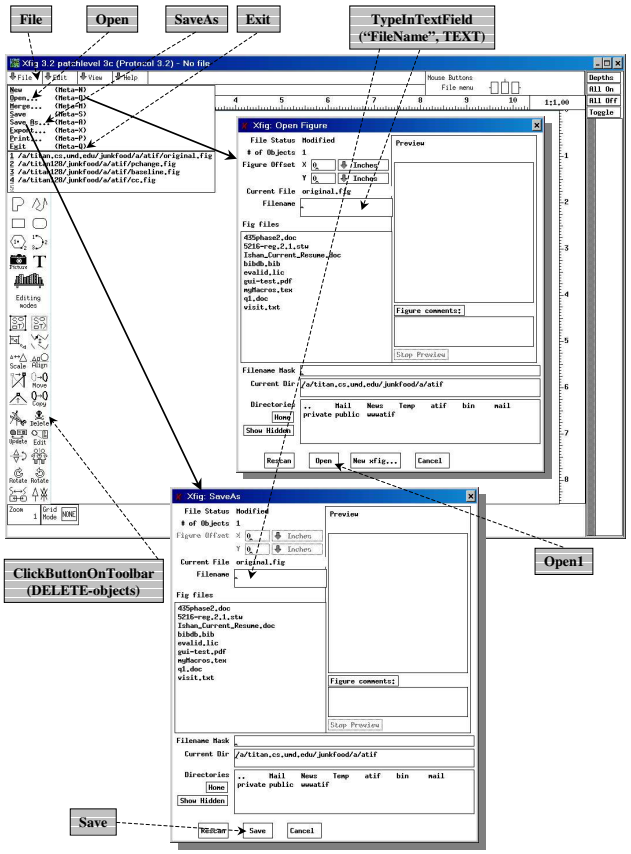

Figure 5. The Modified Software (`Xfig 3.2 patch-level 3c (Protocol 3.2)`).

ware is shown in Figure 6. Our regression tester uses planning to generate the modified test case automatically. This test case can be used for regression testing.

In principle, the approach outlined above can be used for regression testing of GUIs if tasks are maintained with each test case. However, regenerating test cases from scratch is unnecessary since parts of the original test case are still valid and may be reused. We employ a form of hierarchical planning with caching to reuse these parts. We provide details of this technique in Section 6. Later in Section 7, we show that employing this technique considerably speeds up the regression testing process.

3 Plan Generation

We now provide some an overview 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 [24] for 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) [9], a system which extends the ideas of the Graphplan system [5] for plan generation. Graphplan introduced the idea of performing plan generation by converting the representation of a planning problem into a propositional encoding.

IPP forms plans at a single level of abstraction. We have extended this to hierarchical planning, 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.

4 Representing GUI Test Cases

In this section, we first present a model of GUIs that we have used earlier for test case generation [15, 16, 17], test coverage evaluation [19], and test oracles [14, 12]. We then formally define a GUI test case, and explain what we mean

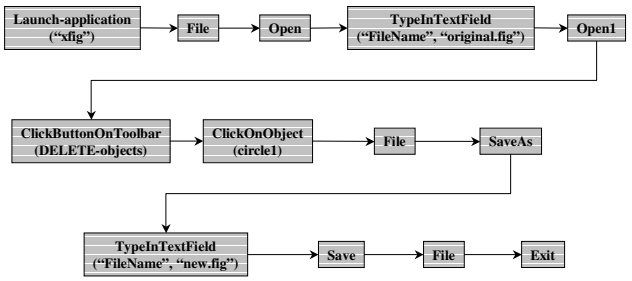

Figure 6. Test Case for the Modified GUI.

by affected and unaffected test cases.

A GUI is modeled as a set of *objects* $O = \{o_1, o_2, \ldots, o_m\}$ (e.g., `label`, `form`, `button`, `text`) and a set of *properties* $P = \{p_1, p_2, \ldots, p_l\}$ of those objects (e.g., `background-color`, `font`, `caption`). Each GUI will use certain types of objects with associated properties; at any specific point in time, the state of the GUI can be described in terms of all the objects that it contains, and the values of all their properties. Formally we define the state of a GUI as follows:

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

With each GUI is associated a distinguished set of states called its *valid initial state set*:

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

The state of a GUI is not static; events performed on the GUI change its state. These states are called the *reachable states* of the GUI. Events are modeled as state transducers. The function notation $S_j = e(S_i)$ is used to denote that S_j is the state resulting from the execution of event e in state S_i. Events may be grouped together into sequences. Of importance to testers are sequences that are permitted by the structure of the GUI. We restrict our testing to such *legal* event sequences, defined as follows:

Definition: A *legal event sequence* of a GUI is $e_1; e_2; e_3; ...; e_n$ where e_{i+1} can be performed immediately after e_i.

Finally, we define a GUI test case as:

Definition: A **GUI test case** T is a triple $< S_0, e_1; e_2; \ldots; e_n, S_n >$, consisting of a *reachable state* S_0, called the *initial state for T*, a legal event sequence $e_1; e_2; \ldots; e_n$, and S_n, the final state of the GUI after the test case is executed.

Note that S_0 and S_n are a part of the test case and represent the task associated with the test case.

5 Modeling GUI Events as Operators

Formally, each event is represented by an operator. These operators are used by AI planners to generate plans. Details of operator definitions have been presented in earlier reported work [14]. Here, we give an overview of operators by presenting an example. Intuitively, an operator specifies the preconditions and effects of the event it represents. Preconditions represent the conditions that must hold before the event can be executed on the GUI. Effects represent the modifications made to the GUI after the execution of the event. For example, the following operator represents an event called `set-background-color` used to change the background-color of a GUI window:

Name: `set-background-color(wX: window, Col: color)`
Preconditions: `is-open(wX)`
Effects: `background-color(wX, Col)`

The operator definition shows that the event `set-background-color( wX, Col )` takes two parameters: a `window` `wX` and `color` `Col`; `wX` and `Col` may take specific values in the context of a particular GUI execution. The preconditions require that the event `set-background-color( wX, Col )` can only be executed in a state in which window `wX` is open; the effects being the background color of `wX` becomes `Col`.

6 Detailed Design of the Regression Tester

As shown earlier in Figure 1, the regression tester consists of two main components: test case selector and test case regenerator. This section provides details of the design of these components.

Test case selector: The test case selector's primary function is to identify affected test cases. In addition, it performs preliminary identification of discarded test cases. For example, it discards all test cases associated with obsolete tasks.

Test case regenerator: The design of the test case regenerator is essentially the same as that of PATHS [17] except that we introduced a cache to reuse parts of the original, affected test case. We haven't included details of the planning algorithms because of space reasons. The key idea is to exploit the hierarchical planning algorithm and, instead of invoking the planner for each sub-plan, we first check to see whether a valid part of the test case already exists. If it exists, then we reuse that part of the test case.

7 Experiments

To explore the practicality of our new regression test case selection and replanning techniques, we implemented the regression tester and evaluated its performance on `Xfig 3.2 patchlevel 0-beta4 (Protocol 3.2)` and `Xfig 3.2 patchlevel 3c (Protocol`

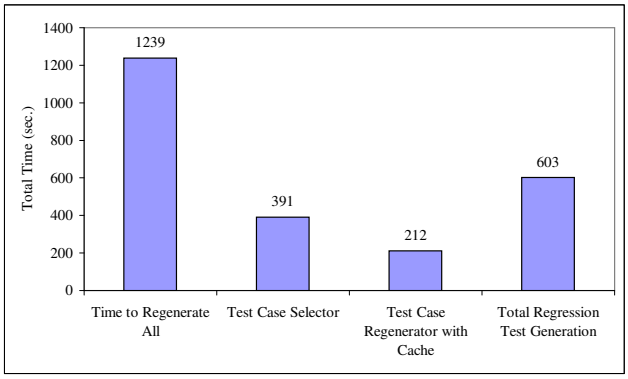

Figure 7. Time Taken to Perform Regression Test Selection/Regeneration vs. Time Taken to Regenerate All.

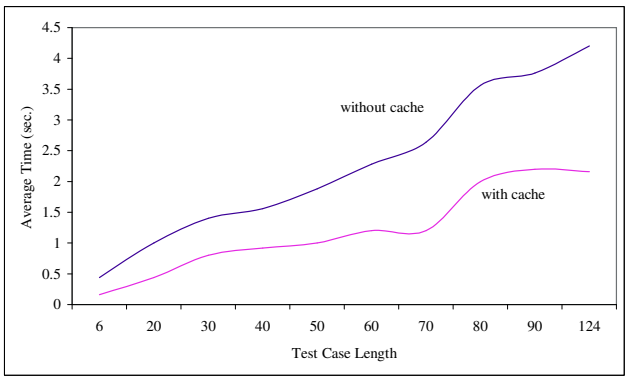

Figure 8. Replanning Using a Cache vs. Not Using a Cache.

3.2). We used the former as our original GUI and the latter as our modified GUI. We specifically wanted to determine (1) how much time it takes to selectively generate the regression test suite and how it compares to regenerating all test cases, (2) how much total time is saved in terms of regeneration and re-execution, and (3) what affect cache has on the regeneration process. The study was conducted on a 1.7 GHz Pentium Workstation with 1 GB of RAM. All execution times are CPU time. The study consisted of the steps described next.

Generating test cases for the original software: There are several automated test case generation techniques that we may have used: (1) task-based using AI planning [17], (2) structural using event-flow graphs and integration trees [19], (3) user-model based using genetic algorithms [8], and (4) random. Since we were going to use the planning-based regression testing technique, we chose to use planning to generate the test cases in the first place. We used the Unix-based IPP [10] planner.

We identified 100 tasks for which we generated 1000 test cases. Note that we can generate multiple test cases for a given task. The test cases varied from length 6 to 124. The total time taken to generate the test cases was 1239 seconds (Figure 7).

Implementation: All the components of the regression tester were implemented, except the planner that we used off-the-shelf. Specifically, the test case selector was implemented in Perl.

Regenerating test cases: The test case regenerator successfully regenerated the affected test cases using the tasks.

Cache vs. no Cache: We wanted to see what effect the cache had on the performance of the test case regenerator. So, we also generated test cases without using the cache. The results are shown in Figure 8. The x-axis shows the test case length, the y-axis shows the average time taken to generate the test cases. In general, the cache cut the regression test case generation time in half.

The regression test cases obtained above form an important part of the regression testing test suite. The test designer will also need to generate additional test cases from new tasks to check the new parts of the GUI.

In this experiment we have successfully demonstrated that our technique is practical and can be used to select test cases for GUI regression testing. The use of this technique helps reduce the cost of GUI regression testing. Our experience with GUI testing has shown that the currently employed techniques, which are largely manual aided with capture/replay tools, require weeks to develop only a few hundred test cases.

8 Related Work

Although regression testing of conventional software has received a lot of attention [4, 21, 22, 23], there has been almost no reported research on GUI regression testing. The exception is White [25] who proposes a Latin square method to reduce the size of the regression test suite. The underlying assumption is that it is enough to check pairwise interactions between components of the GUI. The technique requires that each menu item appears in at least one test case. This strategy seems promising since it also employs GUI events. However, the technique needs to be extended to GUI items other than menus. Moreover, detailed studies need to be conducted to verify whether the pairwise interactions checking assumption is sufficient.

Several strategies for regression testing of conventional software have been proposed [2, 6, 20, 11]. One regression testing strategy proposes rerunning all test cases that have not become obsolete. Since this *retest-all strategy* is resource intensive, numerous efforts have been made to reduce its cost. *Selective retest techniques* [1, 3, 7] attempt to reduce the cost of regression testing by testing only selected parts of the software. These techniques have traditionally focused on two problems: (1) *regression test selection problem*, i.e., selecting a subset of the existing test cases [22], and (2) *coverage identification problem*, i.e., identifying portions of the software that require additional testing.

9 Conclusions

This paper presented a new technique for GUI regression testing based on replanning *affected GUI test cases*. Replanning is facilitated by associating a **task** with each test case. Tasks are activities that can be performed by using the software. The sequence of events in a test case represents the actions needed to complete its associated task. Even when changes to the GUI make test cases unusable, tasks remain valid across successive GUI versions. Affected test cases are identified by employing the specifications of the GUI. Differences between the event-flow graphs and integration trees of the original and modified GUIs are obtained to identify affected test cases. Results of a case study performed on `Xfig 3.2 patchlevel 0-beta4 (Protocol 3.2)` and `Xfig 3.2 patchlevel 3c (Protocol 3.2)` show that the regression testing technique is efficient, in that it identifies test cases that need not be rerun on the modified GUI and helps select a set of affected test cases that are efficiently regenerated and rerun.

References

[1] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. A. London. Incremental regression testing. In *Proceedings of the Conference on Software Maintenance*, pages 348–357, Washington, Sept. 1993.

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

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

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

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

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

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

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

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

[10] J. Koehler, B. Nebel, J. Hoffman, and Y. Dimopoulos. Extending planning graphs to an ADL subset. In S. Steel and R. Alami, editors, *Proceedings of the 4th European Conference on Planning (ECP-97): Recent Advances in AI Planning*, volume 1348 of *LNAI*, pages 273–285, Berlin, Sept.24 –26 1997. Springer.

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

[12] A. M. Memon. *A Comprehensive Framework for Testing Graphical User Interfaces*. Ph.D. thesis, Department of Computer Science, University of Pittsburgh, July 2001.

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

[14] A. M. Memon, M. E. Pollack, and M. L. Soffa. Automated test oracles for GUIs. In *Proceedings of the ACM SIGSOFT 8th International Symposium on the Foundations of Software Engineering (FSE-8)*, pages 30–39, NY, Nov. 8–10 2000.

[15] A. M. Memon, M. E. Pollack, and M. L. Soffa. Plan generation for GUI testing. In *Proceedings of The Fifth International Conference on Artificial Intelligence Planning and Scheduling*, pages 226–235. AAAI Press, Apr. 2000.

[16] A. M. Memon, M. E. Pollack, and M. L. Soffa. A planning-based approach to GUI testing. In *Proceedings of The 13th International Software/Internet Quality Week*, May 2000.

[17] A. M. Memon, M. E. Pollack, and M. L. Soffa. Hierarchical GUI test case generation using automated planning. *IEEE Transactions on Software Engineering*, 27(2):144–155, Feb. 2001.

[18] A. M. Memon and M. L. Soffa. Regression testing of GUIs. In *Proceedings of the 9th European Software Engineering Conference (ESEC) and 11th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-11)*, Sept. 2003.

[19] A. M. Memon, M. L. Soffa, and M. E. Pollack. Coverage criteria for GUI testing. In *Proceedings of the 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9)*, pages 256–267, Sept. 2001.

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

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

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

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

[24] 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.

[25] 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.