EventFlowSlicer: Goal Based Test Generation for Graphical User Interfaces

Jonathan Saddler and Myra B. Cohen
Department of Computer Science & Engineering
University of Nebraska-Lincoln
Lincoln, NE, 68588-0115 USA
{jsaddle,myra}@cse.unl.edu

ABSTRACT

Automated test generation techniques for graphical user interfaces include model-based approaches that generate tests from a graph or state machine model, capture-replay methods that require the user to demonstrate each test case, and pattern-based approaches that provide templates for abstract test cases. There has been little work, however, in automated goal-based testing, where the goal is a realistic user task, a function, or an abstract behavior. Recent work in human performance regression testing has shown that there is a need for generating multiple test cases that execute the same user task in different ways, however that work does not have an efficient way to generate tests and only a single type of goal has been considered.

In this paper we expand the notion of goal based interface testing to generate tests for a variety of goals. We develop a direct test generation technique, *EventFlowSlicer*, that is more efficient than that used in human performance regression testing, reducing run times by 92.5% on average for test suites between 9 to 26 steps and 63.1% across all test suites. Our evaluation shows that the number of tests generated is non-trivial – more than can be easily captured manually. On average EventFlowSlicer generated 38 test cases per suite, and as many as 200 test cases which all achieve the same goal for a specified task.

CCS Concepts

•**Software and its engineering** → **Software verification and validation;**

Keywords

software test generation, graphical user interfaces, goal-based testing

1. INTRODUCTION

Many of our systems today use graphical user interfaces (GUIs) as their front ends. In these systems the user interacts with elements of the interface to perform a task, which consists of a sequence of events to achieve some end state. For instance, to open a new file in most editors the user will click on the *File* menu item, select the *Open* menu choice and then browse for the specified file, eventually double-clicking on the file that they wish to open. Often, the same task may be performed in multiple ways. Instead of opening the file using a menu, there may be a button that provides a single click to open the *Browse* file widget, or the user may be able to type in a file name as a shortcut instead. Each variant of this task can be thought of as one way of achieving a user goal (an end state in the application achieved through a series of steps). Goals represent behaviors of a typical user and are the basis for system validation, usability evaluation and for providing workflow help for users who want to navigate the system.

While goals are often functional (e.g. open file), *how* this functionality is achieved can vary. For instance, in work on human performance regression testing, Swearngin et al. found that there were 81 ways to enter text, make it bold, and then center it in a common word processor, LibreOffice, using different combinations of buttons, menu items and keyboard shortcuts [15]. For that goal the functionality of each of the tests is the same, however how each different test to achieve that goal differs slightly.

Goals may also be more abstract. For instance, a goal might be to change the appearance of a text box and the exact change may be undefined. In this case any method that changes the end state of the text box satisfies this goal.

While there has been a large body of work on software testing (and test case generation) for GUIs [2, 3, 5–8, 10, 12, 17], there has been less work for automatically generating test cases that target specific user goals. The closest work other than the human regression testing is that of AI planning [9] and pattern-based testing [12], however, neither of these approaches aims to generate all test cases that satisfy a particular goal, such as the work of HPRT. A parallel to testing based on pre-defined goals was devised by Moreira and Paiva [11] but this work does not allow the user to expand freely upon the goals that may be tested.

The contributions of this paper are:

1. EventFlowSlicer: a technique to generate all tests for a given user goal;
2. A study showing that EventFlowSlicer is feasible and that an automated technique is needed.

In the next section we present some motivation and foundations. We then present EventFlowSlicer in Section 3. Our Case study is presented in Sections 4 and 5. We then present

A-TEST'16, November 18, 2016, Seattle, WA, USA
© 2016 ACM. 978-1-4503-4401-2/16/11...$15.00
http://dx.doi.org/10.1145/2994291.2994293

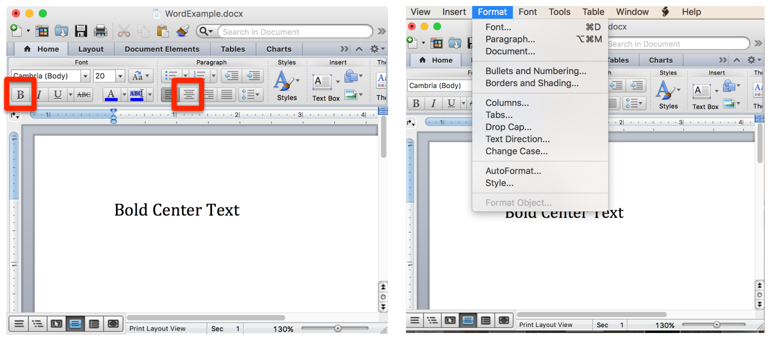

Figure 1: Microsoft Word Menus vs. Toolbars

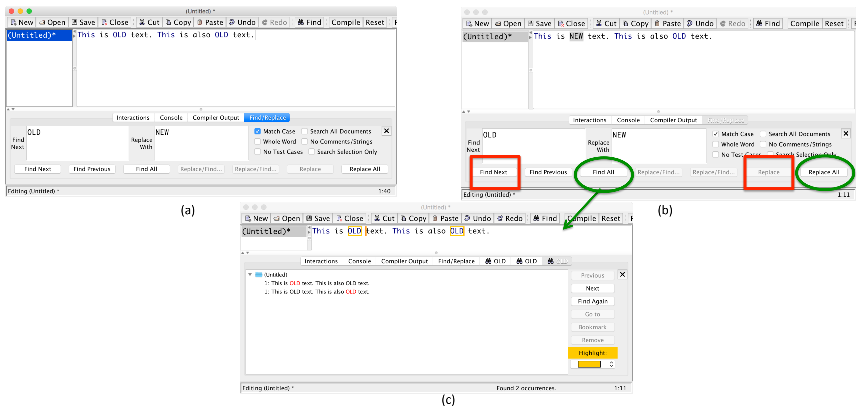
(a)
(b)
(c)
Find Next
Find Previous
Find All
Replace
Replace All
This is OLD text. This is also OLD text.
This is NEW text. This is also OLD text.

Figure 2: DrJava Search and Replace Options

related work (Section 6) and end with conclusions and future work in Section 7.

2. MOTIVATING EXAMPLE

We begin by motivating a few different types of goals that we might encounter in a GUI application. Figure 1 shows the interface of the MS Word application. Suppose we want to bold and center some text. This can be achieved either by clicking buttons (left side of the figure) or by using menus (right side). The same task can also be achieved using a combination of menus and toolbars. We are showing concrete test cases, however, we can state this as an abstract functional goal and any single test case that makes some text bold and centered in a direct manner would satisfy the given use case. Most model-based techniques would not be able to generate a specific goal unless it happened by chance, and given that this test case requires up to 10 events it is unlikely that the commonly used length 2 or 3 test cases would be of use for this goal.

Even if we generate a single test case for this particular functionality, we miss other ways that the goal can be achieved. The differences in how tests are performed (i.e. whether you center or bold first and if you use menus, buttons or combinations of these) is important in domains such as usability where designers measure the time taken to perform tasks on an interface. Since the concrete path taken to achieve a goal will impact the time it takes to perform a task, having the full set of possibilities can be of interest to an interface designer. In functional testing, it may also be important to test a particular goal in multiple ways, to ensure that particular requirements or use cases are covered completely.

We call the different tests in this task *structurally different*, meaning that the goal is achieved using different structural elements of the interface (using buttons versus using menus). In the work of Swearngin et al. [14,15] they defined the notion of generating test cases for human performance regression testing (HPRT) and showed that there is a need to generate test cases that perform this task in all possible ways to evaluate user performance. If adding buttons to the interface provides paths that increase the time it takes a user to make text bold and centered, then this can mean that the application has declined in quality and users may opt to switch to a competitor. In the case of safety-critical applications (such as a cockpit screen) the time taken to perform a task can mean success or failure of a life-saving maneuver [15].

In the HPRT study, a test generation technique to automatically generate test cases like this was presented, however it did not directly generate tests – instead it enumerates all possible paths of events on an event flow graph that extend to the length of provided parameter l, and then removes tests based on a set of provided constraints $C_1, C_2, ..$ to weed out tests that do not actually perform the given task in a direct manner (as would an expert user). While this has a similar effect as the test case generator here, HPRT does not scale since it requires exhaustive test case generation on large models. In the example above, for instance, there are test cases of length 4, 5, 6, 7, 8, 9 and 10, which requires the generation of all possible tests at each length which takes over 2 hours. As the lengths get longer this grows exponentially with the number of widgets. Prior work using AI planning by Memon et al. [9] performed goal-based testing, but did not consider structural differences when generating tests (e.g. only a sample of test cases would be generated from that technique).

Figure 2 shows another example of a user goal. In this scenario we see the search and replace function within the DrJava Editor. In this screen there are multiple ways to find and replace text. On the left part of this figure you see that we can enter text into the box to find and replace a word. We can select specific options (such as matching case) or not, and on the right we can find and replace multiple instances by selecting the find and replace multiple times or by simply clicking *Find all*. While also structurally different, we believe that the tests generated for this task amount to very different sets of tasks, even though the final state will be the same. We view these as having the same *Functional Goal*, but utilizing different steps in different ways to get there. We differentiate this difference from the prior type of task which differed only in structure. In HPRT, this type of task is not supported – these would be considered different tasks.

The last type of task that we identify in this work are those tasks which have *Abstract Goals*. This type of goal is loosely defined. We might say that we want to generate tests to *change the background*, but do not have a complete specification for what changing the background means. In this case the user can change the background color, pattern, transparency, etc. All achieve the same goal, but are different. The work on pattern-based testing is the most similar to this type of goal [12], however the aim of that work is to define a general pattern and then generate some tests for it, not to generate all tests. Pattern based testing also does not utilize event flow graphs.

In this work we support all three types of tasks and for each generate all possible ways to perform that task. We summarize them here:

Structural Differences Only Test cases are differentiated by only the type of widgets used (e.g. menus versus buttons) to perform each basic test step (i.e. bold or center) on the interface. There can be a direct mapping created between (sets of) events in one test case to another, to functionally perform the same core steps that are needed to achieve the goal from the starting state. What is important is that the function of each of the core steps does not differ. The steps can occur in different orders, however, if that still achieves the goal.

Same Functional Goal All test cases in a test suite of this type must lead from the same initial state to a single end state. In this type of test suite, the steps to get to that single state can differ significantly and a direct mapping between test cases may not exist. The variation can occur both in the number of steps taken to achieve the goal, and the types of steps taken. In this type of goal, the steps to reach the end state are disimilar in their function, while the test case end goal is still the same.

Same Abstract Goal A tester's end goal for each test case may be defined more coarsely than just by what differentiates the state of two instances. Different test cases in this type of goal stem from the same initial state, but may have a different end state, therefore the steps taken to achieve these goals may be quite different. The end state of the test is defined based on an abstract definition of how the interface state *should change* according to the tester. Judging the correctness of the end state in this type of goal is more difficult than it is for the other two types of goals.

2.1 Foundations

In this paper we utilize the model-based approach for testing GUIs based on graphs [8, 13]. We extract an event flow graph (EFG) automatically from a running application through a ripping process that extracts events on the interface. In event flow graphs, nodes represent events and edges represent the follows relationships between nodes (i.e. this node can follow another node). In the HPRT work [15], Swearngin et al. presented a technique to reduce the EFG by ripping only those events that are of interest to the user. In that work they manually defined the events (we use a capture mechanism for this). They also defined a set of global constraints that must be satisfied if a task is to represent a realistic user task (performed by an expert user). A realistic task is one that is performed directly without unnecessary steps. These constraints are (1) task must end in a main window (2) an expand event must be followed by a child event (3) a window or tab open and close without any intermediate events is not allowed and (4) there are no repeat events (unless overridden by local constraints). They also provided local rules that the user needed to specify:

1. **Exclusion.** This rule says which events cannot appear together. If for instance, there is a button to make text bold as well as a menu item, these are included in an exclusion rule.

2. **Order.** This constraint provides a partial order of events in the test case. For instance, in the bold and center example, *Select* must occur before both *Bold* and *Center*, but *Bold* and *Center* are equivalent.

3. **Requires.** This rule says which events must be in every test case. The HPRT generator does not use this rule fully during generation. Instead it relies on the test case length. We discuss a modification to this rule in the next section.

4. **Repeat.** This rule allows us to override the global rule that says no events can be repeated. This is necessary for tasks which use the same widget (e.g. a menu item) multiple times.

The global rules are built into the test case generator, however the user provides local rules. In order to generate test cases for HPRT, the user must identify all possible lengths that a particular task can take, and then run the generator for each length. The generator will then traverse the graph starting at entry nodes (nodes that the user can immediately access) using an exhaustive traversal strategy, such as that of GUITAR [13]. The generator searches all possible test cases that are of the given length, and for each test case checks constraints, saving only those which pass the rules.

This does not scale as we show in our case study, because it is an exhaustive generation strategy and grows exponentially with test case length. EventFlowSlicer does not require the test case length parameter and thus requires only one run to generate the entire test suite. It also removes graph edges prior to traversal that violate certain constraints, and as it generates tests it prunes paths, cutting off a subpath if it will violate other constraints. We describe its generation strategy in more detail in the following sections.

3. EVENTFLOWSLICER

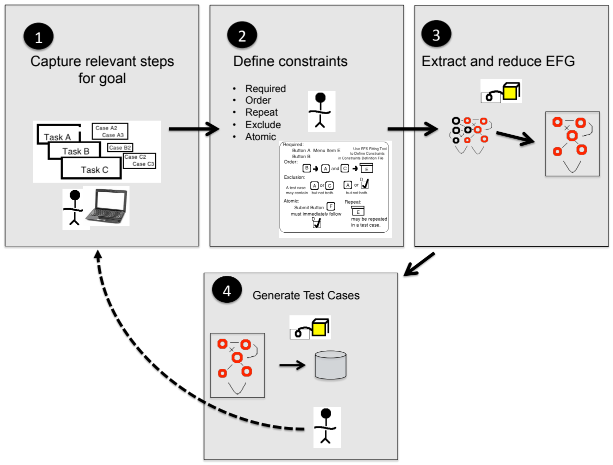

Figure 3: EventFlow Slicer

We now present EventFlowSlicer (EFS), our strategy for goal-based test case generation in GUIs. EventFlowSlicer uses a 4-step process to generate test cases from an event flow graph model, in which paths in the EFG that are not relevant to the goal are pruned using a set of global constraints imposed by the generation platform, and by rules the user can specify. Figure 3 shows the process which allows for optional iteration back to the first step since we have learned that modeling sometimes is an iterative process.
Capture. First the tester captures relevant steps for a goal using a GUI interface, where only the events captured will be nodes in the EFG.
Define Constraints. In step 2, the user defines the constraints, by selecting from the widgets captured and placing them in "constraints groups" (discussed below).
Reduce EFG. In step 3, the application is ripped using a model-based ripper such as that provided in GUITAR, but modified to extract *only those events captured* in step 1. The EFG is then reduced in several stages, removing edges that violate the first three global constraints (defined in the HPRT work [15]). In brief, these constraints reduce the EFG by removing common threats to the generation of succinct test cases– those that don't help maintain direct progress toward a goal. One of the reductions, for instance, removes edges on an EFG that allow a window to be opened and then closed without performing any additional actions. Another ensures that a menu cannot be opened and then closed again without some intermediate event. These reductions are essential to the test case generation process; they cannot be turned off. The last global constraint, "No Repeat Events" cannot be applied to reduce the graph at this stage, because it may remove edges that are necessary for other paths, and additionally, the rule can be overridden by a local rule.
Test Case Generation. In the last step, test case generation occurs by generating all tests in the EFG (paths) using a depth first search algorithm on the reduced EFG that are valid with respect to both local constraints provided by the user, and those that do not violate the merged repeat rules. Test case generation is both direct and focused. Tests are generated top down, and paths are pruned whenever branches of the EFG are reached that violate a constraint. This is in contrast to HPRT which enumerates all tests first, followed by a post generation removal of invalid tests.

Finally, if the user is not satisfied with the structure of the test cases, the tester can return to the beginning of the process to recapture new widgets and constraints, or can skip the first step and redesign constraints if the constraints defined don't require a larger or smaller scope of widgets. In the end, all test cases are generated from the reduced EFG given the set of constraints prepared.

3.1 Key Differences

We highlight some differences between the HPRT and EFS approaches. The HPRT approach is an exhaustive enumeration solution. It generates all possible sequences of events of a particular length which gives it a halting criterion. The EFS solution on the other hand, operates on the same initial graph and does not miss any test cases that satisfy constraints. By cutting off paths that violate constraints (e.g. illegally repeat vertices, paths that violate rules limiting when two widgets may be grouped together, or paths violating the order in which events may appear) it does not have to exhaustively traverse the graph. We have also eliminated the search-depth parameter in EFS so that all possible lengths of test cases are generated with a single run. The EFS is perhaps more fragile in that a sufficient set of constraints are required in order to halt generation since there is no maximum search depth given. We did not find this to be a limitation in our experiments.

Another key difference is the type of goal supported. HPRT was built to support only structural types of differences in test cases.

3.2 New Constraints

We modified one of the local constraints from HPRT and added a new one that we found was needed in some of our tasks. We describe these next.

1. **Atomic.** This is a new constraint that allows the user to specify strict sub-sequences with a test case. We have found some scenarios where the partial order is not strong enough.

2. **Mutually Required.** This is an extension to the HPRT required constraint. It allows users to specify that at least one of a set of widgets must appear (but not all). The original required rule only allows individual widgets to be required. But when we have different options (menu item versus a toolbar) this rule is needed to halt test case generation.

4. CASE STUDY

In this section we present a case study to evaluate EventFlowSlicer. Supplementary material and artifacts can be

Table 1: Table of Tasks

Goal Name	Task Variants	**Description**	Source
LibreOffice Writer *Format Text (FT)* (**Structural**)	Menus Only (M) Menus, Keyboards (MK) MK + toolbars (MKT)	Type some text Make the text bold and centered	HPRT
LibreOffice Writer *Insert Hyperlink (IH)* (**Structural**)	Menus Only (M) Menus, Keyboards (MK) MK + toolbars (MKT)	Open "Hyperlink" window Add a link Add link text Make the link text uppercase	HPRT
LibreOffice Calc *Absolute Value (AV)* (**Structural.**)	Menus Only (M) Menus, Keyboards (MK) MK + toolbars (MKT)	Open "Function Wizard" window Type a value in the cell Shift the cell just edited one cell to the right Turn off column and row headers	HPRT
LibreOffice Impress *Insert Table (IT)* (**Structural**	Menus Only (M) Menus, Keyboards (MK) MK + toolbars (MKT)	Open the "Table" window Type number in the columns text box add a new slide Hide the task pane	HPRT
JEdit *Comment Indent (CI)* (**Structural**)	N/A	File is already open with some text Change line to a java source line comment Shift the line of text one tab to right	N/A
DrJava *Search Options (SO)* (**Functional**)	N/A	File is already open with some text Open "Find/Replace" panel Enter word in the [Find Next] box Enter word in the [Replace With] box Search for and replace the word	N/A
TerpWord *Bold Center (BC)* (**Structural**)	N/A	Type text in a document Make text bold and centered	N/A
DrJava *Compile File (CF)* (**Structural**)	N/A	A Java HelloWorld Program is open Compile the program Run the program with no arguments	StackOverflow
JEdit *Commented Text (CT)* (**Abstract**)	Text Color (TC) Background (BG) Both (FULL)	A Java program is open with a comment Open the "Style Editor" window Change appearance of all comments Confirm any changes, and exit	StackOverflow
JEdit *FourParagraphs (FP)* (**Functional**)	Window (W) Menu Dominant (MD)	Java file is open with four lines of text Open the "Search and Replace" window Enter a regular expression to find and replace content in all 4 lines	StackOverflow

found on our website.[1] We ask the following research questions.

RQ1 How does EventFlowSlicer compare with the HPRT test generation algorithm in terms of efficiency?

RQ2 Can we effectively generate test cases for the different types of tasks identified?

4.1 Objects of Study

For RQ1 we use the 12 tasks from the HPRT study [15]. These include test cases for four tasks in LibreOffice 3.4.3, that utilize three of the applications modules, `Writer`, `Calc`, and `Impress`. In addition we added 9 more tasks on three more applications to show generalization. These are tasks for JEdit version 5.1.0, DrJava version 20140826-r5761, and TerpWord 3 (2003 release). For the TerpWord task we used the bold and center task from LibreOffice since this is also a word processor. For the other tasks we found user questions on StackOverflow for JEdit and DrJava and translated those into our goals. We show the task details in Table 1 along with their classification and source. Some of the tasks have different variants. For instance in the HPRT work there are tasks that work on a version only with Menus (M), only with Menus and Keyboards (MK) and on an application version that has Menus, Keyboards and Toolbars (MKT). We provide this information in the table. For each task we say if it is structural, functional or abstract. Most are structural differences only, since these were the subject of the HPRT work. However, we do add two tasks (one in JEdit and one in Dr. Java) that are of the type same functional goal. These are both tasks to perform search and replace which can be achieved in very different ways in the two applications. We also have one abstract goal task in JEdit. This task aims to change the appearance of comments in the current document.

All test cases generated for this study were validated to ensure that they *make direct progress* toward the desired output - the state of the application should monotonically progress toward resembling the intended final state. We verified using a manual review of all test suites that each test case met this requirement. For the test cases for HPRT, we validated that the same exact set of test cases were generated using both techniques. This ensures that the test cases are realistic for an expert user (as in HPRT) and would be potentially useful in industrial real-world use: they don't repeat widgets unnecessarily and that each completes the task in a direct manner.

[1]http://cse.unl.edu/~myra/artifacts/EventFlowSlicer/

4.2 Metrics

For RQ1 we use the time taken for generation, and the number of constraints into consideration. For time we want to reduce running times of the generator. For the constraints we want to reduce the count since the user must manually define these. For RQ2 we show the number of test cases obtained for each task and validate them visually for correctness.

4.3 Study Method

For all of the structural only difference tasks, we run HPRT and EventFlowSlicer both and compare them. For the tasks that are functional or abstract we run only EventFlowSlicer. We run all tests on the same computing cluster with AMD Opteron(TM) CPUs running at 2000MHz and 128 GB memory. We run the HPRT generator one time due to the long runtimes. For EventFlowSlicer we ran the generator 5 times and took the average time. We redirect the output to a file to allow these to run without human intervention. We validated each of the resulting test cases from EventFlowSlicer manually by executing them using a test case replayer and observing that they perform the required task. This took approximately 23 man hours.

4.4 Threats to Validity

We have only validated our test generation on a limited number of tasks, however we have used ones from the literature and added additional tasks obtained from StackOverflow. We believe that these are representative of small user tasks. We did not consider the time taken to write the constraints in EventFlowSlicer, however this process is similar to what is required to write constraints in HPRT and we do not significantly increase the number that must be defined. Finally, we did not use the test cases to evaluate faults, however we did run each test case in this study to confirm their validity manually. We provide supplementary data on our website with the constraints, EFGs and test cases to reduce this threat.

Table 2: Cumulative Run Times

No.	Task	**EFS time(s)**	**HPRT time(s)**	**Diff**	**% Reduct**
1	FT M	9.2	8	-1.2	-13.0%*
2	FT MK	12.6	107	94.4	88.2%
3	FT MKT	19	11257	11238	99.8%
4	IH M	6.6	7	0.4	5.7%
5	IH MK	9.2	35	25.8	73.7%
6	IH MKT	10.4	295	284.6	96.5%
7	AV M	10	16	6	37.5%
8	AV MK	12	280	268	95.7%
9	AV MKT	18	4254	4236	99.6%
10	IT M	7.2	4	-3.2	-44.4%*
11	IT MK	8	23	15	65.2%
12	IT MKT	15	45	30	66.7%
13	CI	5	45	40	88.9%
14	SO	14.8	261	246.2	94.3%
15	BC	8.0	106	98.0	92.5%
16	CF	4.8	N/A	N/A	N/A
17	CT TC	10.8	N/A	N/A	N/A
18	CT BG	11.6	N/A	N/A	N/A
19	CT FULL	19.6	N/A	N/A	N/A
20	FP W	15.8	N/A	N/A	N/A
21	FP MD	10.4	N/A	N/A	N/A

5. RESULTS

We begin by answering RQ1 (test case efficiency). We show a comparison of the test case generation between HPRT and EventFlowSlicer in Table 2. For each task we show the runtime in seconds of EventFlowSlicer (EFS) followed by HPRT, the difference in seconds and the percent reduction (% Reduct). For tasks that are either functional differences or abstract we have not provided times from the HPRT generator, as HPRT cannot handle these. There is one exception. For the DrJava Search Options (SO) task (#14) we were able to still generate tests using the correct test case lengths (6,7,8,9,10) therefore we do have a comparison for this case.) We also show the lengths of test cases (and the counts at each length) in Table 3.

Table 3: Test Cases Lengths

Task	# TC	breakdown (L) length/ (C) count								Avg. Len
FT M	3	L	**10**	-	-	-	-	-	-	3
		C	3	-	-	-	-	-	-	
MK	24	L	**6**	**7**	**8**	**9**	**10**	-	-	5.5
		C	3	6	6	6	3	-	-	
MKT	81	L	**4**	**5**	**6**	**7**	**8**	**9**	**10**	5.5
		C	12	6	18	21	12	9	3	
IH M	2	L	**9**	-	-	-	-	-	-	9
		C	2	-	-	-	-	-	-	
MK	8	L	**6**	**7**	**8**	**9**	-	-	-	7.5
		C	2	2	2	2	-	-	-	
MKT	18	L	**6**	**7**	**8**	**9**	-	-	-	7
		C	8	4	4	2	-	-	-	
AV M	3	L	**10**	-	-	-	-	-	-	10
		C	3	-	-	-	-	-	-	
MK	32	L	**9**	**10**	**11**	**12**	-	-	-	10.5
		C	4	12	12	4	-	-	-	
MKT	72	L	**7**	**8**	**9**	**10**	**11**	**12**	-	9.5
		C	8	12	12	20	16	4	-	
IT M	3	L	**8**	-	-	-	-	-	-	8.6
		C	3	-	-	-	-	-	-	
MK	12	L	**6**	**7**	**8**	-	-	-	-	7
		C	3	6	3	-	-	-	-	
MKT	36	L	**5**	**6**	**7**	**8**	-	-	-	6.3
		C	3	15	12	3	-	-	-	
CI	16	L	**3**	**4**	**5**	**6**	**7**	**8**	-	5.5
		C	2	2	4	4	2	2	-	
SO	64	L	**7**	**8**	**9**	**10**	**11**	-	-	8.6
		C	2	8	18	24	12	-	-	
BC	8	L	**8**	**9**	**11**	**12**	-	-	-	10
		C	2	2	2	2	-	-	-	
CF	4	L	**2**	**3**	**4**	-	-	-	-	3
		C	1	2	1	-	-	-	-	
CT TC	26	L	**5**	**7**	**8**	**9**	**10**	-	-	8.6
		C	2	1	5	10	6	-	-	
BG	26	L	**5**	**6**	**9**	**10**	**11**	**12**	-	10.1
		C	2	2	1	5	10	6	-	
		L	**5**	**6**	**8**	**9**	**10**	**11**	**12**	
FULL	200	C	2	2	1	7	15	15	6	14.2
			13	**14**	**15**	**16**	**17**	-	-	
			3	16	50	59	24	-	-	
FP W	114	L	**9**	**13**	**15**	**16**	**18**	-	-	15.9
		C	6	6	48	6	48	-	-	
MD	36	L	**11**	**19**	**20**	**25**	**26**	-	-	21.4
		C	4	4	12	4	12	-	-	

We can see that the total running time of HPRT is as high as 11,257 seconds (over 3 hours) for the Format Text MKT task, yet it takes only 19 seconds using EventFlowSlicer. We see as high as a 99.8% reduction in runtimes. Most of the tasks have reductions higher than 60 percent. On two

Table 4: Types and Counts of Constraint Rules Uses

Application	Category	Rules HPRT	Rules EFS	EFS Rule Types REQUIRE	EXCLUDE	ORDER	REPEAT	ATOMIC	Widgets
Format Text	**M**	4	7	4	-	2	-	-	9
	MK	7	9	4	2	2	1	-	12
	MKT	7	10	4	3	2	1	-	15
Insert Hyperlink	**M**	6	7	5	-	2	-	-	9
	MK	8	9	5	2	2	-	-	11
	MKT	8	9	5	2	2	-	-	11
Absolute Value	**M**	6	8	5	-	2	1	-	11
	MK	8	10	5	2	2	1	-	14
	MKT	9	11	5	3	2	1	-	16
Insert Table	**M**	4	6	4	-	1	1	-	7
	MK	6	8	4	2	1	1	-	9
	MKT	7	9	4	3	1	1	-	11
Comment Indent		5	8	3	3	1	1	-	9
Search Options		6	7	5	1	1	-	-	11
Bold Center		6	8	4	2	1	1	-	12
Compile File		NA	6	2	2	1	1	-	9
Commented Text	**TC**	NA	2	1	1	-	-	-	11
	BG	NA	3	1	1	-	-	1	12
	FULL	NA	7	1	-	-	4	2	13
Four Paragraphs	**W**	NA	12	4	3	1	-	4	12
	MD	NA	16	6	3	2	1	4	12

of the smaller tasks (Format Text using only menus and Insert Table using only menus), EventFlowSlicer is slower. These tasks are relatively simple with only 4 test cases. In this situation the overhead of EventFlowSlicer loading and reducing graphs seems to be an overriding factor, yet both of the generators take less than 10 seconds to complete the generation. The tasks from HPRT are limited to lengths of at most 12 (after this point, the exhaustive generation technique does not allow us to complete), however, we have generated tests with lengths as high at 26 using EFS (last task of Table 3).

Overall we see an average of 63.1% reduction of runtime over HPRT. If we only consider test cases of length 9 and above then EventFlowSlicer shows an improvement of 92.5% over HPRT.

We next turn to Table 4 which shows the types and counts of constraint rules used by each technique. First we show the total number of constraints (rules) used for runs on either generator. We see that there is a slight increase in rules from HPRT to EFS, however we never increase constraints by more than 3. We do see several tasks where the atomic rule was used (this does not exist within HPRT). Not shown in the table is that typically the extra rules came from the need to define additional *(mutually) requires* rules, to ensure that the generator finds the correct number of widgets before accepting a test case as valid for output.

Summary of RQ1. We conclude that EventFlowSlicer is more efficient timewise than HPRT. Although we see some increase in constraints it is small.

To answer RQ2 we examine the tasks that we have defined for this study. We include tasks from the original HPRT work, ones that have functional differences only and ones that are abstract. For each of these tasks we are able to create rules and generate the tests using EventFlowSlicer. The test cases generated are then run using a modified version of the GUITAR Replayer and we observed them as they run to confirm that they perform the required tasks. We were able to re-create the HPRT tests successfully (we confirmed both the lengths of tests, and also performed a diff of the test cases). We found 3 test cases where the actual events within the HPRT generator differed from ours, but after examination we believe that these were actually incorrect (we believe that the wrong test cases were posted online).

We next examine the number of test cases generated for the various tasks in Table 3. We see as many as 200 test cases for one task and on average 38. This also shows that EventFlowSlicer is effective in that it automates a difficult manual process.

Summary of RQ2. We conclude that EventFlowSlicer is effective at generating test cases for all three types of tasks that we have identified.

6. RELATED WORK

There has been a large body of work on GUI testing [2, 3, 5–8, 10, 12, 17]. We focus here on model-based and other goal based techniques. In model-based testing, the most widely used technique reverse engineers the interface to create an event flow graph and then generates all tests to satisfy some coverage criteria such as all length 2 test cases, random of some length, or using combinatorial coverage [2, 8, 10, 17]. We use model-based testing and event flow graphs in this work and have built our techniques on top of the GUITAR framework, however, in general this work aims to cover events in a graph and has no notion of a test goal.

In early work by Memon et al. [9] they proposed the use of AI planning to generate tasks (such as ours), however in that work they do not aim to generate all tasks and do not differentiate between structural events (e.g. bold is bold whether it uses a menu or a toolbar).

In the work of Moriera et al. they propose pattern based testing [11, 12]. That work is similar in that they define abstract patterns that can occur on a GUI and then generate test cases to perform that task. In their work they focus only on the abstract tasks and do not attempt to perform a complete generation. Furthermore their aim is primarily for functional fault detection and do not consider the user performance aspects.

Several dynamic, state-based tools such as TESTAR [4,16] and GUI Driver [1] have been proposed. TESTAR dynamically produces long random test cases from a given starting point on the application while GuiDriver [1] is built on GUITAR. While these approaches scale to longer test cases they do not solve the problem of generating tests for a particular goal.

Finally, our work is most closely related to that of the HPRT [15] research where the notion of a goal is for usability testing [14]. We have built our generator on top of that work but have extended its functionality. First we add different types of tasks that HPRT does not directly support. Second we perform an EFG edge reduction and then generate only the tests that satisfy rules via a depth-first search with pruning.

7. CONCLUSIONS AND FUTURE WORK

In this paper we have presented EventFlowSlicer, a technique for goal based GUI test generation. EventFlowSlicer uses a four step process to generate all test cases that perform a particular task. The test cases can be structurally different, have different functionality, or be abstract. We have compared EventFlowSlicer with the HPRT test case generator and find that it is 92 percent more efficient on average when the test case lengths are above 9 steps and 63 percent more efficient overall. We also have shown that we can successfully generate tasks for all three types, and that the number of constraints does not grow too large.

In future work we will incorporate this into the HPRT tool, will perform user studies to evaluate the ease of task definition and will run larger empirical studies on more tasks across more applications. We will also carry out testing for both functional and human performance faults.

8. ACKNOWLEDGMENTS

This work was supported in part by NSF grants CCF-1161767 and CNS-1205472.

9. REFERENCES

[1] P. Aho, N. Menz, T. Räty, and I. Schieferdecker. Automated Java GUI Modeling for Model-Based Testing Purposes. In *2011 Eighth International Conference on Information Technology: New Generations*, pages 268–273. IEEE, Apr. 2011.

[2] S. Arlt, A. Podelski, I. Banerjee, A. M. Memon, and M. Schaf. *Lightweight Static Analysis for GUI Testing.* IEEE International Symposium on Software Reliability Engineering, 2012.

[3] G. Bae, G. Rothermel, and D.-H. Bae. On the Relative Strengths of Model-Based and Dynamic Event Extraction-Based GUI Testing Techniques: An Empirical Study. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*, pages 181–190, Nov. 2012.

[4] S. Bauersfeld and T. E. J. Vos. GUITest: A Java Library for Fully Automated GUI Robustness Testing. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 330–333, New York, NY, USA, 2012. ACM.

[5] S. Carino and J. H. Andrews. Dynamically Testing GUIs Using Ant Colony Optimization (T). In *Automated Software Engineering (ASE), IEEE/ACM International Conference on*, pages 138–148. IEEE, Nov. 2015.

[6] F. Gross, G. Fraser, and A. Zeller. Search-based system testing: High coverage no false alarms. *Proceedings of the International Symposium on Software Testing and Analysis*, 2012:67–77, 2012.

[7] L. Mariani, M. Pezze, O. Riganelli, and M. Santoro. AutoBlackTest: Automatic Black-Box Testing of Interactive Applications. In *IEEE International Conference on Software Testing, Verification and Validation*, pages 81–90, Apr. 2012.

[8] A. M. Memon. An event-flow model of GUI-based applications for testing. *Journal of Software Testing, Verification and Reliability*, 17:137–157, 2007.

[9] 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.

[10] A. M. Memon and Q. Xie. Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software. *Software Engineering, IEEE Transactions on*, 31(10):884–896, Oct. 2005.

[11] R. M. L. M. Moreira and A. C. R. Paiva. PBGT tool: An integrated modeling and testing environment for pattern-based gui testing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 863–866, New York, NY, USA, 2014. ACM.

[12] R. M. L. M. Moreira, A. C. R. Paiva, and A. Memon. A pattern-based approach for GUI modeling and testing. In *Proceedings of the 24th annual International Symposium on Software Reliability Engineering (ISSRE 2013)*, 2013.

[13] B. N. Nguyen, B. Robbins, I. Banerjee, and A. Memon. Guitar: an innovative tool for automated testing of GUI-driven software. *Automated Software Engineering: An International Journal*, 21:65–105, 2013.

[14] A. Swearngin, M. Cohen, B. John, and R. Bellamy. Easing the generation of predictive human performance models from legacy systems. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI, pages 2489–2498, 2012.

[15] A. Swearngin, M. B. Cohen, B. E. John, and R. K. Bellamy. Human performance regression testing. *Proceedings of the 2013 International Conference on Software Engineering*, pages 152–161, 2013.

[16] T. E. Vos, P. M. Kruse, N. Condori-Fernández, S. Bauersfeld, and J. Wegener. Testar: Tool support for test automation at the user interface level. *Int. J. Inf. Syst. Model. Des.*, 6(3):46–83, July 2015.

[17] X. Yuan, M. Cohen, and A. Memon. GUI interaction testing: Incorporating event context. *IEEE Transactions on Software Engineering*, 37(4):559 –574, 2011.