# EventFlowSlicer: A Tool for Generating Realistic Goal-Driven GUI Tests

Jonathan A. Saddler, Myra B. Cohen
Department of Computer Science & Engineering
University of Nebraska-Lincoln
Lincoln, NE 68588-0115, USA
saddler@huskers.unl.edu, myra@cse.unl.edu

***Abstract*—Most automated testing techniques for graphical user interfaces (GUIs) produce test cases that are only concerned with covering the elements (widgets, menus, etc.) on the interface, or the underlying program code, with little consideration of test case semantics. This is effective for functional testing where the aim is to find as many faults as possible. However, when one wants to mimic a real user for evaluating usability, or when it is necessary to extensively test important end-user tasks of a system, or to generate examples of how to use an interface, this generation approach fails. Capture and replay techniques can be used, however there are often multiple ways to achieve a particular goal, and capturing all of these is usually too time consuming and unrealistic. Prior work on human performance regression testing introduced a constraint based method to filter test cases created by a functional test case generator, however that work did not capture the specifications, or directly generate only the required tests and considered only a single type of test goal. In this paper we present EventFlowSlicer, a tool that allows the GUI tester to specify and generate all realistic test cases relevant to achieve a stated goal. The user first captures relevant events on the interface, then adds constraints to provide restrictions on the task. An event flow graph is extracted containing only the widgets of interest for that goal. Next all test cases are generated for edges in the graph which respect the constraints. The test cases can then be replayed using a modified version of GUITAR. A video demonstration of EventFlowSlicer can be found at https://youtu.be/hw7WYz8WYVU.**

***Index Terms*—Software test generation, graphical user interfaces, goal-based testing**

## I. Introduction

Automated test generation for Graphical User Interfaces (GUIs) has been the subject of a large body of research [1]–[8]. Approaches include model-based techniques, which have led to tools such as GUITAR [9], random techniques, or search-based techniques that systematically explore the application event-space to increase code coverage [3], [6]. However, all of these methods for test generation, focus primarily on functionality with the aim of exercising as many events (and event sequences) on the interface as possible. While this is effective for some types of testing, it fails to mimic real tasks that a user might perform. Recent work on usability testing, Human Performance Regression Testing (HPRT), proposed a technique to automate test case generation that mimics the various ways an expert user can perform some task based on a stated goal [10]. This allows the interface designer to find problematic paths in the interface. In HPRT, a set of widgets on the interface that participate in the given task are first enumerated and a set of constraints are added to ensure that the task is realistic (i.e. it doesn't open and close a window before a useful action is performed). That work, however, has several limitations. First, the types of tasks allowed are limited to those which can be performed on the interface in different ways, but with only minor structural differences (e.g. menus versus buttons). Second, the user has to specify widgets and constraints manually in a text file (with no tool support). Third, the tests are not directly generated, but rather, are filtered from the set of all possible functional tests on an event flow graph which limits scalability.

Other approaches for realistic user testing include pattern based testing [4], [11], however, that work differs in that it uses a template for a specific goal and does not provide the flexibility of generating tests for any type of test goal. Memon et al. [12] used AI planning, however they generate a single test case and require the user to specify the exact orders of sequences of actions in the path. Zhang et al. [13] use static analysis to extract the set of actions to perform a workflow on the interface, but they do not generate replayable tests and require access to the source code. There has also been some work on generation of realistic test cases for web applications [14]–[16], however they do not generate event sequences for desktop GUI applications.

In this paper we present our tool, EventFlowSlicer (EFS for short), that solves the stated challenges of HPRT. EFS was first described in [17], [18]. It allows users to specify and generate test cases for a variety of goal types. The first type of test goal is the most stringent and expects that the test cases differ only in slight structural ways (such as by using a menu versus a keyboard, or by switching the order of a task). Our example task in this paper is structural. EFS also supports goals that are functionally different. These types of goals have at least two different ways to perform the same task that use very different methods (such as using entirely different menus and steps). An example of such a task used in our prior work is that of search and replace in DrJava. The third type of goal that is supported is an abstract goal. A user might specify a general goal – that they want to change the look of some text. This might include making the text a different font, or a different color, or italicizing, etc. We can define such a test goal using EventFlowSlicer and generate all of the test cases constrained

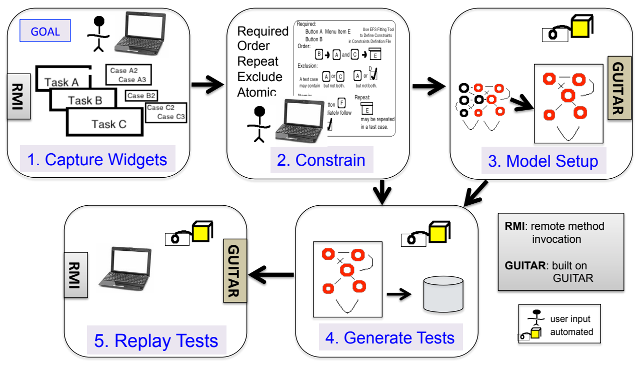


Fig. 1. Overview of EventFlowSlicer

by a set of user defined rules. The task used in our running example can be made into an abstract goal, by removing some of our constraints. We used an abstract version of this goal which generated 200 test cases in our prior work [17].

In EventFlowSlicer, the user is provided both with a capture mechanism and a graphical constraint interface that walks them through each part of the process. They then push a button to extract a small event flow graph relevant only to their task and to generate tests that satisfy their goal. Once generated, the tests can be replayed from within EventFlowSlicer. EventFlowSlicer can also read inputs from prior steps, so that the process can be started at any point and artifacts reused. EventFlowSlicer utilizes the model-based approach of the GUITAR framework for its ripping process and replay [9].

We begin next with an overview of the tool architecture and discuss the targeted users. We then present a running example of the use of each of the modules of the tool in Section III. We follow with a discussion of our validation of EFS (Section IV). Then we conclude and present future work in Section V.

## II. System Architecture

Figure 1 shows an overview of EventFlowSlicer. It has five primary modules labeled 1 to 5. The first module allows the user to provide input via a capture tool that creates an initial part of the goal specification. This part of the process uses Java remote method invocation (RMI) to allow the application to open and close in a separate process. Once the widgets are captured, the second module provides a graphical interface for the user that walks them through the process of defining the sets of constraints for the given goal. There are five types of constraints (described later), *Requires*, *Order*, *Repeat*, *Exclude* and *Atomic*. Once the constraints are defined, the next phase is to perform the model setup which creates an event flow graph (a graph model of the events and their follows edges which are dynamically extracted from an interface [19]). containing only those widgets that are captured. This module is built on top of the GUITAR ripper [9]. Once the model has been created, the next step is to generate tests. The event flow graph is explored in a depth-first manner, pruning edges that violate constraints, and recording paths. All paths that satisfy all the constraints are output as the set of tests. Last, the tests can be replayed using a modified version of the GUITAR relayer. We have retrofitted this module to use RMI. We will discuss each module in more detail in the next section.

**Envisioned Users** EventFlowSlicer is developed both for experienced testers who can use the command-line input and directly modify the constraints files, as well as for less experienced testers such as user interface designers, who want to explore different ways to perform the same task on the interface. We expect that the output of the test case replay can be used in some existing performance prediction tools as was done in HPRT [10]. We also envision that EFS can be used to help discover ways to perform actions on a task for a tester who iteratively interacts with the tool, relaxing and adding constraints in the process.

## III. Example Tool Walkthrough

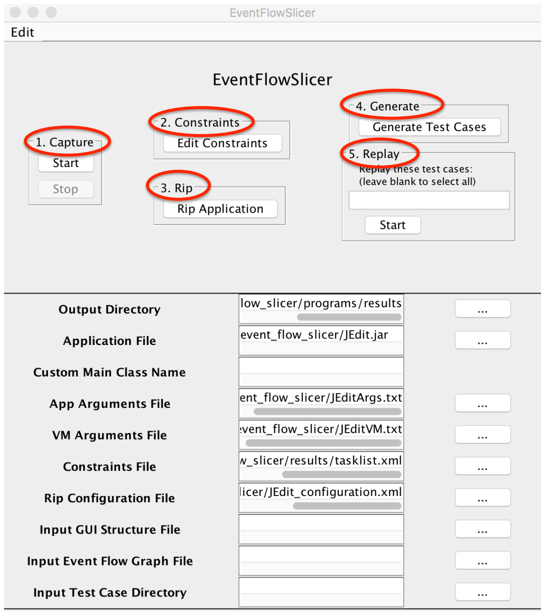
We use a small task (modified from [17]) based on a Stack Overflow question. The user asked how to change the stylization (specifically background color) of text typed into a Java source file as a comment [20].

We have defined our task to satisfy a specific (non-trivial) goal that can be achieved many ways. The user wants to make the text of the Java comment in jEdit to be bold and italic, and to have a new background color. As part of our goal we will allow the user to make these changes in any order and to use both the mouse and keyboard to achieve the background color changes. Each valid test case must have all three changes made. Our task in [17] called Commented Text (BG) is similar to this one, however it is abstract in that it did not require that all three changes happen, but only at least one (hence there were more ways to perform the task). More formally our goal in this demonstration will be to *enumerate all possible ways the jEdit Style Editor can be opened and used to change the commented text so that it is italicized, bold, and has a gray background color*. There are 12 ways to achieve the goal we have stated. This task will be run on jEdit Version 5.1.0.

In order to make this task realistic when creating test cases, we follow a few principles.

- The test case must not repeat any states or move to unnecessary states leading to the goal state.
- The tester is always actively exploring *new states of the editor* when they click the OK button.
- The test case must always make forward progress towards achieving the goal.

To begin, we start EventFlowSlicer. We show the main screen in Figure 2. Notice that some of the text fields are filled in already for demonstration. At startup these are empty, but can be auto-filled by using command line arguments and passing the `-gui` flag. EventFlowSlicer has five steps (#1-5, circled in red) that are performed in order. Most of these steps can be skipped by providing input files such as a constraints file which will skip step #1 and #2, or a set of tests which will allow one to move directly to step #5. As input to start the application, the tester provides an output directory and location of the application. This can be a .jar file (shown in the figure) or can be a class file. Optional virtual machine and



| | |
|---|---|
| Output Directory | low_slicer/programs/results |
| Application File | event_flow_slicer/JEdit.jar |
| Custom Main Class Name | |
| App Arguments File | ent_flow_slicer/JEditArgs.txt |
| VM Arguments File | vent_flow_slicer/JEditVM.txt |
| Constraints File | v_slicer/results/tasklist.xml |
| Rip Configuration File | licer/JEdit_configuration.xml |
| Input GUI Structure File | |
| Input Event Flow Graph File | |
| Input Test Case Directory | |

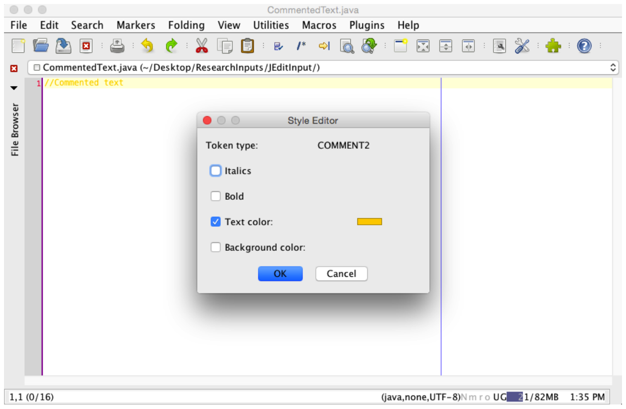
Fig. 2. EventFlowSlicer main screen

application specific arguments can be provided as input files. No other input is needed to begin step #1.

## A. Capture

Once the initial information is provided to EFS, the capture button is clicked and the application under test will open. We show jEdit open during the capture phase in Figure 3. The jEdit Style Editor is not initially visible to the tester from the main editor window — it must be revealed by clicking a menu item in the main window's [Utilities] menu. The tester opens this window via the menu (which is being captured by EFS) and upon entering this window they click on each of the widgets that are involved in the task. In this case they will click on the Italics, Bold and Background color checkboxes. This can be done in any order. They will ignore the text color picker, since this is not needed for the task and should not be captured. Once the background color checkbox is selected, another color picker becomes accessible. They will click on that and they are taken to a color window where they can select the background color either by clicking on a cell with the mouse or by using the keyboard. The tester will capture both actions (again in either order). The following widgets will be captured by EventFlowSlicer:

- "Utilities" menu
- "Quick Settings" submenu
- "Edit syntax style" menu item
- "Italics" check box
- "Bold" check box
- "Background color" check box and push button
- the "Light-gray " swatch button using mouse
- the "Light-gray " swatch button using the arrow keys + space bar


File Edit Search Markers Folding View Utilities Macros Plugins Help
File Browser
Style Editor
OK
Cancel

CommentedText.java

CommentedText.java (~/Desktop/ResearchInputs/JEditInput/)

```
//Commented text
```

Token type: COMMENT2

☐ Italics

☐ Bold

✓ Text color:

☐ Background color:

1,1 (0/16) (java,none,UTF-8)Nmro UG 1/82MB 1:35 PM

Fig. 3. The JEdit Style Editor Window, Nothing Selected

- Pick a Color "OK" button
- Style Editor "OK" button

During capture the tester visually selects the widgets and the labels are printed to the console to help identify what actions EFS is capturing. When done, he or she hits the Stop Capture button and the widgets are recorded to a file. The file containing all these widgets is automatically used in the next phase (and added to the interface in the field labeled Constraints file). The tester can manually edit this XML file if some incorrect widgets were selected.

## B. Constrain

This step specifies how these widgets work together to achieve the goal. It will add constraints related to the captured widgets. During this phase a GUI walks the tester through each constraint in order, and provides a list of the possible widgets which can participate in this rule to make the tester's job easier. Any of the captured widgets are allowed to be added into constraints groups, and test case generation will enforce that the test cases adhere to these rules.

To briefly review from our previous work [17], the five constraints the tester can work with in the currently implemented version of EventFlowSlicer are shown below. We assume that $W = \{w_1, w_2, ..., w_n\}$ is the set of captured widgets.

- **Requires**: Given a set $R \subseteq W, \exists w_i \in R$ appearing in each generated test case.
- **Exclusion**: Given the set $E \subseteq W$, $\forall \{w_i, w_j\} \in E, w_i \neq w_j$, at most one of $w_i$ and $w_j$ can appear in any generated test case.
- **Repeat**: Given a set $P \subseteq W$ and a *minBound* and *maxBound*, $\forall w_i \in P$, $w_i$ may occur up to $r$ times in a generated test case where $minBound \leq r \leq maxBound$[1].
- **Order**: Given a sequence, of $n$ sets $S$=<$O_1, O_2, ..O_n$>, $O_i \subset W$ and $\forall \{O_i, O_j\}, O_i \cap O_j = \emptyset$, and $i < j$, $\forall w_i \in O_i, w_i$ appears before all $w_j \in O_j$ in every test case. This is a partial order on widgets.

[1]Without specifying any constraint, *minBound* = 0, *maxBound* = 1

- **Atomic**: A sequence, $S = <w_i, w_j, ..w_n>$ where $w_i \in W$. This is an exact order of widgets.

Each of these rules can be used repeatedly. For instance, we can require three separate groups of widgets by adding 3 requires rules. In our running example, the following constraints were employed:

1) Three *Requires* constraints, one for each of the "Italics", "Bold", and "Background Color" checkboxes. This will force all three to appear in a test case for it to be valid. (Note: We can also add all three to a single requires constraint, meaning that at least one must be present in a test case — creating a more abstract goal.)
2) One *Exclusion* constraint containing both the keystroke and click option for selecting a color. This means test cases will only use one of these methods to change the background color.
3) An *Atomic* constraint that states that the "Background color checkbox" is clicked before its selector button is clicked. If we were to change this to an order constraint stating the "checkbox" appear only at *some time prior* to selecting a color, we would more than double the number of test cases generated. We bind these two actions together for the sake of an understandable working example. EFS supports both options.

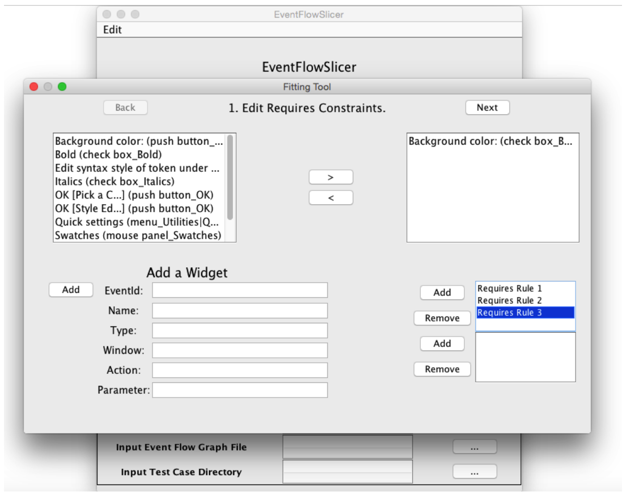
An example constraint tool window is show in Figure 4. The possible widgets are on the left, the current constraint in the display is the *Requires* constraint. As each widget is selected for this rule it will be added on the right. We require only a few *Requires* rules to obtain a suite that is focused on just the three formatting widgets we mentioned. In the jEdit application, the "Italics", "Bold", and "Background color" buttons can only be found in the *Style Editor* window, shown in Figure 3, and they are also among the few widgets we captured in the first step. Thus, having required that each test case use one of these widgets, we force every test case to first navigate down the menu path to the Style editor (the one denoted by *"Utilities", "Quick Settings", and "Edit Syntax Style"*.

The space of test cases generated is limited to realistic, expert-user test cases, based on global constraints (see [10]) which will hold as long as constraints specified in the Constraints Tool don't override them. Test cases will not open a window, or click to enter a tab in a tabbed selection panel, and then immediately close the window or panel to perform actions elsewhere. Each event will be used only once in each test case (in absence of overriding *Repeat* rules), and all tasks will end and begin in the main window the application opened when launched. Without global constraints, the test case might continually check and uncheck the italic or bold option, or leave the Style Editor without changing the state at all.

We are now done specifying our test suite to the generator. EventFlowSlicer allows us to immediately transition into letting the program handle the rest of the work.

## C. Model Setup

The next step is to create the models from which to derive test cases. The generator takes the captured output and the



Background color: (push button_...
Bold (check box_Bold)
Edit syntax style of token under ...
Italics (check box_Italics)
OK [Pick a C...] (push button_OK)
OK [Style Ed...] (push button_OK)
Quick settings (menu_Utilities|Q...
Swatches (mouse panel_Swatches)

Background color: (check box_B...

Requires Rule 1
Requires Rule 2
Requires Rule 3

Fig. 4. The EventFlowSlicer Constraints Tool

output of the constraints selection, both combined into one file, and gathers necessary data from the interface using a GUITAR *rip operation* and creates an event flow graph (EFG), a graph where nodes represent the events in the application and edges indicate a node can follow another one. The rip operation is explained in detail in [19]. The resulting EFG does not provide a complete graph of the interface, but instead only rips the widgets that were captured and are contained in the constraints file [19]. We show the full EFG for jEdit in Figure 5(a). This has hundreds of nodes and thousands of edges. By contrast, the EFG that is ripped using EventFlowSlicer is shown in Figure 5(b). This is further reduced in the first part of test generation (Shown as Figure 5(c)).

EventFlowSlicer uses a GUITAR filter to prune widgets that are not captured. If a widget sits under a menu item that wasn't explicitly captured, the path to it is left in the EFG to ensure proper test cases can be generated. The result of "extra hidden" widgets is reported by EventFlowSlicer at the end of the Model Setup step.

## D. Generation

Once the model is created, the tester can select Generate Test Cases. The first step makes sure that the Constraints file and EFG file were generated for the same application. Each main element in the constraints file is checked for a unique mapping to an event in the EFG. The second step is to reduce the EFG to remove some edges based on the global rules (see [10], [17] for more information). In particular the NoRepeats rule implies that widgets should by default not be repeated in test cases, so the reduction removes any self edges from the EFG (and other edges which traversing would automatically imply illegal breaking of global rules in a test case). There are three such reductions we make, RepeatSelf, WindowOpencloseCannotHappen, and ExpandToChild. They are described in [17], [18]. We have experimented with various orderings of these three reductions (since the last reduction is impacted by the former) and have found that the different

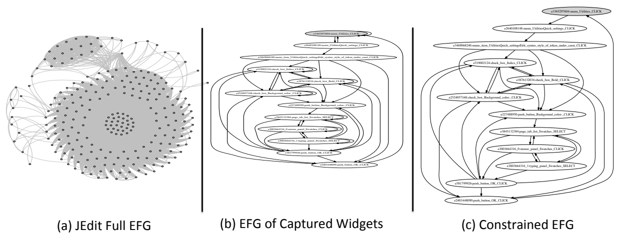


Fig. 5. 3 phases of the EFG reduction. Unconstrained ripping to obtain jEdit's full EFG is shown in (a). EventFlowSlicer will produce a reduced EFG from its rip shown in (b). The test generation phase will reduce the EFG further by removing edges for the global rules, just before generation (c)

orderings of reductions have an insignificant effect, but that reductions overall are useful in reducing graphs to nearly half their size on average, which reduces test generation time. The resulting event flow graph after the global rules are removed in our example is shown in Figure 5(c).

Once these reductions and mappings are made, the generator then starts systematically traversing the EFG from the widgets that exist on the main window. When a test case containing all *Requires* widgets is found, (and that does not violate other constraints) it is returned as a test case. The traversal then continues, avoiding branches that violate constraints as it goes. In the end, all specified test cases are generated.

## *E. Test Cases*

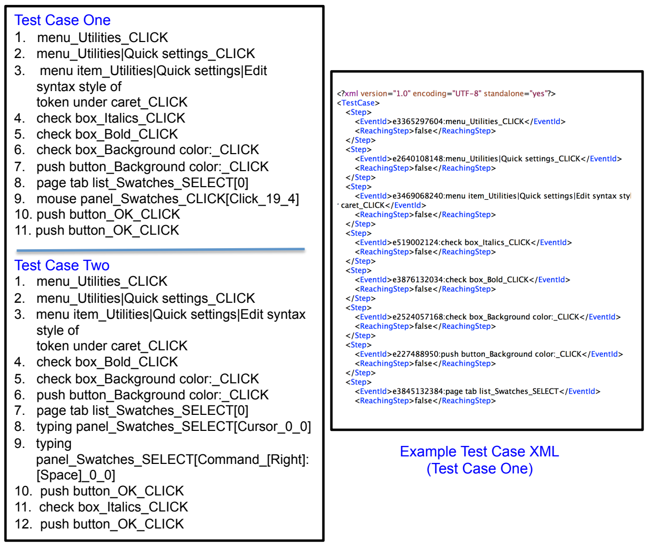
Our working example will result in 12 test cases. There are six different tests that account for each of the possible orders for Italic, Bold and Background Color, and for each of these, there are two ways to change the background color (mouse or keyboard).

We have verified that the test cases generated from EventFlowSlicer for this instance achieve the desired result, by manually examining each test case. We present examples of two test cases in Figure 6. The left side lists the steps for both of the test cases while the right side shows a portion of the actual test case XML for the first test case which is in a modified GUITAR format. The GUITAR generator creates widget IDs using a 10-digit hash value. As a design decision, EventFlowSlicer test cases are relabeled with English language descriptions of the widgets, so that the tester can readily see what the test cases are doing, before having to replay the test case. This is useful for a number of reasons:

- The tester need not run the test cases in order to understand what it is doing.
- Scripts can utilize this information to mine data about the widgets in each test case.
- Reasoning about the validity of each test case to achieve the desired goal is possible by looking at a visualization of the EFG relabeled in the same manner and studying paths taken to achieve the generated test cases.

## *F. Replay*

The last phase of EventFlowSlicer is test case replay. The tester can replay all tests in the test suite, or select a subset by


Test Case One
1. menu_Utilities_CLICK
2. menu_Utilities|Quick settings_CLICK
3. menu item_Utilities|Quick settings|Edit syntax style of token under caret_CLICK
4. check box_Italics_CLICK
5. check box_Bold_CLICK
6. check box_Background color:_CLICK
7. push button_Background color:_CLICK
8. page tab list_Swatches_SELECT[0]
9. mouse panel_Swatches_CLICK[Click_19_4]
10. push button_OK_CLICK
11. push button_OK_CLICK

Test Case Two
1. menu_Utilities_CLICK
2. menu_Utilities|Quick settings_CLICK
3. menu item_Utilities|Quick settings|Edit syntax style of token under caret_CLICK
4. check box_Bold_CLICK
5. check box_Background color:_CLICK
6. push button_Background color:_CLICK
7. page tab list_Swatches_SELECT[0]
8. typing panel_Swatches_SELECT[Cursor_0_0]
9. typing panel_Swatches_SELECT[Command_[Right]:[Space]_0_0]
10. push button_OK_CLICK
11. check box_Italics_CLICK
12. push button_OK_CLICK

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<TestCase>
  <Step>
    <EventId>e3365297604:menu_Utilities_CLICK</EventId>
    <ReachingStep>false</ReachingStep>
  </Step>
  <Step>
    <EventId>e2640108148:menu_Utilities|Quick settings_CLICK</EventId>
    <ReachingStep>false</ReachingStep>
  </Step>
  <Step>
    <EventId>e3469068240:menu item_Utilities|Quick settings|Edit syntax sty
· caret_CLICK</EventId>
    <ReachingStep>false</ReachingStep>
  </Step>
  <Step>
    <EventId>e519002124:check box_Italics_CLICK</EventId>
    <ReachingStep>false</ReachingStep>
  </Step>
  <Step>
    <EventId>e3876132034:check box_Bold_CLICK</EventId>
    <ReachingStep>false</ReachingStep>
  </Step>
  <Step>
    <EventId>e2524057168:check box_Background color:_CLICK</EventId>
    <ReachingStep>false</ReachingStep>
  </Step>
  <Step>
    <EventId>e227488950:push button_Background color:_CLICK</EventId>
    <ReachingStep>false</ReachingStep>
  </Step>
  <Step>
    <EventId>e3845132384:page tab list_Swatches_SELECT</EventId>
    <ReachingStep>false</ReachingStep>
```

Example Test Case XML
(Test Case One)

Fig. 6. Our running example produces test cases that bold, italicize, and change the background color of text, in any order. The atomic constraint ensures that the background color activation checkbox is checked immediately prior to the color selection window being opened in every test case. On the left we show two abstract test cases consisting of just the steps. On the right we show the partial XML for the first test case.

number. The output from the generation step is a set of test case files that can be executed on the application. In this step, the application is opened by EFS, and sent events mapping to the descriptions given in the test case file. Some Java Swing applications are intolerant to "soft closing" of an application's windows by test automators. (Applications may block the user from doing so at certain stages where other forms of input like text input are required, or the app may hold up the system in order to write files to the file system). Choi et al. [21] point this out as a problem when testing GUI's. We use remote method invocation (RMI) to avoid this problem. Since EFS opens the application in a subprocess, it has control over restarting the application for the next test case, and prevents the underlying system under test from interfering with this process.

The output from this operation is a set of snapshots that demonstrate all of EventFlowSlicer's actions. Borrowing from techniques used in [10], EFS executes an action, and then takes a visual snapshot to provide a graphical story of what happened during the test case. Images from a variant of our running example, jEdit Commented Text, were used to help verify the over 200 test cases we got back from one of our tasks [17].

## IV. Validation

In our initial work [17] we validated EventFlowSlicer on a set of tasks comprising 21 user goals on four different applications. These include both structural only goals, as well as those which have functional differences and which are abstract. The first twelve (structural) were taken from prior work on human performance regression testing [10] for LibreOffice. There were four core tasks (two on Writer, one for Calc and one for Impress). Each of these was performed on three different versions of the system, one that had only menus, one that had menus and keyboard shortcuts, and one with menus, keyboard shortcuts and toolbar buttons). Using this study, we were able to confirm that we generated the exact same test cases that the HPRT generator would produce when fed similar inputs.

Furthermore, we evaluated the generation of nine newer goal-based test suites (six that were based on Stack Overflow questions and three that we created for other applications). We used three applications: jEdit, DrJava and TerpWord. For each of these tasks we performed the full workflow of EventFlowSlicer, beginning with capture. The test case count of these test suites ranged from 3 to 200 (average of 38). Test generation times (after capture and constraints) ranged from 5 seconds (4 test cases) to 19.6 seconds (200 test cases). For this study, we validated each test case against the test cases generated via the HPRT technique (in the 12 instances where we mimicked their goal). For the other test cases we manually examined the test cases. The runtime for test generation of EventFlowSlicer showed a 63.1% reduction over the HPRT test generation technique. The goal used in the running example here is a restricted version of the Commented Text, BG task[2].

## V. Conclusions and Future Work

This paper has presented EventFlowSlicer, a goal-based tool for test specification and generation. EventFlowSlicer allows the tester to capture relevant widgets for a task, specify constraints and then automatically extract a model, generate tests and replay them. All intermediate artifacts are saved so the tester can start at any point in the process, and/or edit them manually. We have examined the use of EventFlowSlicer, in helping with the task of exploring the space of all possibilities in a goal (changing commented text's background color, and making it bold and italic in jEdit). The user performs the manual steps of *capture* and defining *constraints* as input to the generator, which then creates a test suite of 12 test cases covering all intended manners of accomplishing our goal.

In future work we will make EventFlowSlicer available for others to use and extend. We also plan to develop a method to construct a graph of the outcome of a test suite replay operation. One use of this graph is to help interface designers weed out pathologically long scenarios that users would have trouble executing. A second use when paired with our snapshots, would be to visualize where bugs appear in an interface, and to help localize faults to certain usage scenarios. We also intend to evaluate the usability of the tool and evaluate the difficulty of defining constraints.

## Acknowledgments

This work was supported in part by National Science Foundation grants CCF-1161767, CNS-1205472 and CCF-1745775.

[2]Artifacts from our validation study [17] can be found at: http://cse.unl.edu/∼myra/artifacts/EventFlowSlicer/

## References

[1] A. M. Memon and Q. Xie, "Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software," *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 884–896, Oct. 2005.

[2] 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 *International Symposium on Software Reliability Engineering (ISSRE)*, Nov. 2012, pp. 181–190.

[3] F. Gross, G. Fraser, and A. Zeller, "Search-based system testing: High coverage no false alarms," *International Symposium on Software Testing and Analysis (ISSTA)*, pp. 67–77, 2012.

[4] T. Monteiro and A. C. R. Paiva, "Pattern Based GUI Testing Modeling Environment," in *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, Mar. 2013, pp. 140–143.

[5] X. Yuan, M. Cohen, and A. Memon, "GUI interaction testing: Incorporating event context," *IEEE Transactions on Software Engineering*, vol. 37, no. 4, pp. 559 –574, 2011.

[6] S. Carino and J. H. Andrews, "Dynamically Testing GUIs Using Ant Colony Optimization," in *International Conference on Automated Software Engineering (ASE)*, Nov. 2015, pp. 138–148.

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

[8] S. Bauersfeld and T. E. J. Vos, "GUITest: A Java Library for Fully Automated GUI Robustness Testing," in *International Conference on Automated Software Engineering (ASE)*, 2012, pp. 330–333.

[9] B. N. Nguyen, B. Robbins, I. Banerjee, and A. Memon, "GUITAR: an innovative tool for automated testing of GUI-driven software," *Automated Software Engineering*, vol. 21, pp. 65–105, 2013.

[10] A. Swearngin, M. B. Cohen, B. E. John, and R. K. Bellamy, "Human performance regression testing," *International Conference on Software Engineering (ICSE)*, pp. 152–161, 2013.

[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 *International Conference on Automated Software Engineering (ASE)*, 2014, pp. 863–866.

[12] A. M. Memon, M. E. Pollack, and M. L. Soffa, "Hierarchical GUI test case generation using automated planning," *IEEE Transactions on Software Engineering*, vol. 27, no. 2, pp. 144–155, Feb. 2001.

[13] S. Zhang, H. Lü, and M. D. Ernst, "Automatically Repairing Broken Workflows for Evolving GUI Applications," in *International Symposium on Software Testing and Analysis (ISSTA)*, 2013, pp. 45–55.

[14] M. Bozkurt and M. Harman, "Automatically generating realistic test input from web services," in *International Symposium on Service Oriented System (SOSE)*, Dec. 2011, pp. 13–24.

[15] P. McMinn, M. Shahbaz, and M. Stevenson, "Search-Based Test Input Generation for String Data Types Using the Results of Web Queries," in *International Conference on Software Testing, Verification and Validation*, Apr. 2012, pp. 141–150.

[16] L. Mariani, M. Pezzè, O. Riganelli, and M. Santoro, "Link: Exploiting the Web of Data to Generate Test Inputs," in *International Symposium on Software Testing and Analysis (ISSTA)*, New York, NY, USA, 2014, pp. 373–384.

[17] J. Saddler and M. B. Cohen, "EventFlowSlicer: Goal based test generation for graphical user interfaces," in *International Workshop on Automating Test Case Design, Selection, and Evaluation (A-TEST)*, 2016, pp. 8–15.

[18] J. A. Saddler, "EventFlowSlicer, a goal-based test case generation strategy for graphical user interfaces," Master's thesis, University of Nebraska, Lincoln, 2016.

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

[20] (2014, April) Highlight comments background in Jedit. [Online]. Available: https://stackoverflow.com/questions/27180136/highlight-comments-background-in-jedit

[21] W. Choi, G. Necula, and K. Sen, "Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning," *SIGPLAN Not.*, vol. 48, no. 10, pp. 623–640, Oct. 2013.