Automatically Repairing Broken Workflows for Evolving GUI Applications

Sai Zhang   Hao Lü   Michael D. Ernst
Department of Computer Science & Engineering
University of Washington, USA
{szhang, hlv, mernst}@cs.washington.edu

ABSTRACT

A workflow is a sequence of UI actions to complete a specific task. In the course of a GUI application's evolution, changes ranging from a simple GUI refactoring to a complete rearchitecture can break an end-user's well-established workflow. It can be challenging to find a replacement workflow. To address this problem, we present a technique (and its tool implementation, called FlowFixer) that repairs a broken workflow. FlowFixer uses dynamic profiling, static analysis, and random testing to suggest a replacement UI action that fixes a broken workflow.

We evaluated FlowFixer on 16 broken workflows from 5 real-world GUI applications written in Java. In 13 workflows, the correct replacement action was FlowFixer's first suggestion. In 2 workflows, the correct replacement action was FlowFixer's second suggestion. The remaining workflow was un-repairable. Overall, FlowFixer produced significantly better results than two alternative approaches.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging.
General Terms: Reliability, Experimentation.
Keywords: Dynamic analysis, GUI applications, workflows.

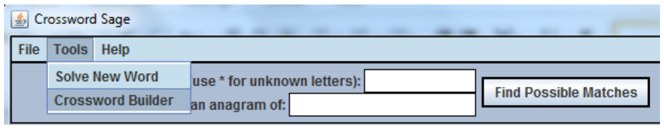
1. INTRODUCTION

Most users interact with a software application through its Graphical User Interface (GUI). The software developers evolve the GUI over time to improve the user experience. Many popular software systems, from web-based systems like social media to desktop applications like office suites, routinely revamp their GUIs. Such GUI changes can be even more frequent than changes to the core domain logic.

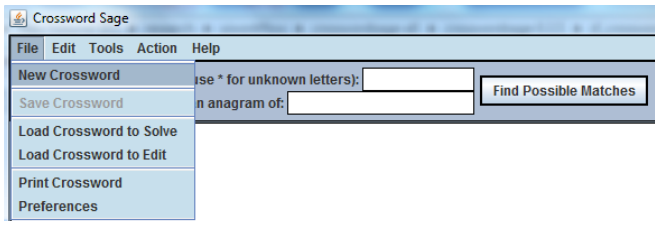
1.1 The Broken Workflow Problem

Although application evolution is generally beneficial, it can create usability problems for end-users. Changes ranging from a simple GUI refactoring to a complete rearchitecture can break an end-user's workflow — a sequence of UI actions to complete a specific task. To recover from a broken workflow due to GUI evolution, end-users often need to seek information from online help forums, the software user manual, or experts. This process can be tedious, laborious, and frustrating.

ISSTA'13, July 15–20, 2013, Lugano, Switzerland
Copyright 2013 ACM 978-1-4503-2159-4/13/07...$15.00
http://dx.doi.org/10.1145/2483760.2483775



(a) Crossword version 0.3.0



(b) Crossword version 0.3.5

Figure 1: Evolution of the Crossword [4] GUI breaks an existing workflow: building a new crossword puzzle. In version 0.3, a user clicks "Tools → Crossword Builder" to create a new puzzle. In version 0.35, this menu item has been removed and a functionally equivalent one, "File → New Crossword" is added. Our technique FlowFixer automatically recommends "click menu item: File → New Crossword" as a replacement UI action to repair this broken workflow in version 0.35.

As an example of real GUI application evolution, Figure 1 shows screenshots corresponding to versions 0.3 and 0.35 of the Crossword project [4]. The GUI changes break an end-user's workflow. In version 0.3, a user clicks the menu item "Tools → Crossword Builder" to create a new crossword puzzle; but in version 0.35, this menu item has been removed, and the user must instead click the menu item "Files → New Crossword".

The broken workflow problem is not rare in practice. For example, our examination of Microsoft Office user forums [31] indicates that broken workflows are serious issues. For *every* release of *each* Microsoft Office product in the past 5 years, end-users complained about their broken workflows due to UI changes and sought information to repair them. These users had already tried but failed to find the new workflow before they wrote the forum posting. Writing a good forum post is harder than reproducing an existing workflow (since the post has to clearly describe the existing workflow).

GUI evolution presents problems not just for end-users, but also for software developers. To automate testing of a GUI application, test engineers often write test scripts to mimic end-user workflows by performing actions on GUI elements. Such test scripts are fragile to UI changes. According to an empirical study [28], as many as 74% of test scripts become unusable between successive releases of a typical GUI application. An internal evaluation of automated testing in Accenture showed that even simple modifications to GUIs resulted in 30% to 70% changes to test scripts [14].

1.2 Repairing Broken Workflows

Manually repairing every broken workflow is tedious and frustrating, since a GUI application like Microsoft Word contains hundreds of GUI screens and thousands of UI actions. It is infeasible for a software user to explore this space to choose replacement actions.

Another possible approach is to programmatically compare the GUIs of two versions, identify changed GUI elements, and then locate UI actions that reference these modified GUI elements [7,14,26]. This is a blackbox impact analysis that does *not* require analysis of the GUI application's source code. Section 4.3.3 empirically demonstrates that existing GUI-comparison-based approaches only repair a small number of broken workflows. There are two fundamental limitations that cause this poor performance. (1) For some UI actions (e.g., filling a blank textbox), their effects cannot be observed by merely comparing GUIs statically without actually executing actions. Existing blackbox GUI-comparison-based approaches fail to distinguish UI actions that look similar but result in different consequences. (2) During software evolution, the GUI can change substantially: a GUI element may be moved from one screen to another, the label of a GUI element may be modified, and a GUI element may even be replaced by another GUI element with a different type (i.e., replacing a menu item with a toolbar button). Therefore, it is generally impractical to find a replacement GUI element and a suitable action on it from the new application version without knowing the precise "action semantics". For these reasons, a GUI-comparison-based approach like [14] only warns about affected workflow steps (i.e., test script statements) that should be modified, but does not indicate how to fix the broken workflow.

Our approach: a program-analysis-based solution. This paper uses program analysis to repair broken workflows that are affected by GUI evolution. Our technique, called FlowFixer, recommends replacement UI actions in the updated GUI application to complete the same workflow.

The key observation behind FlowFixer is that during the evolution of a GUI application, the underlying system that implements a given functionality often stays relatively the same between versions, even when its GUI evolves rapidly. To repair a broken workflow, FlowFixer analyzes how the methods invoked by a workflow in the old version were changed during application evolution. Specifically, if a UI action (e.g., click a menu item) is removed from the application, FlowFixer first identifies methods invoked when performing that UI action in the old version, then analyzes how these invoked methods differ in the new version, and finally reasons about which UI actions in the new version can trigger the updated methods.

FlowFixer works in four steps (illustrated in Figure 2):

1. **Dynamic Workflow Profiling.** FlowFixer instruments the old application version, so that executing the instrumented application produces an execution trace. FlowFixer asks the user to demonstrate a workflow on its GUI. This is an easy, fast task for the user. FlowFixer analyzes the obtained execution trace to extract all invoked methods.
2. **Static Method Matching.** For each method invoked by the workflow in the old version, FlowFixer identifies its corresponding, possibly updated method in the new version.
3. **Random Action Execution.** FlowFixer instruments the new application version. Then, it randomly executes UI actions on the instrumented version and observes which methods get invoked. Based on the action execution and method invocation information, FlowFixer builds a map from each executed UI action to its invoked methods. This step can be performed ahead of time, by the software vendor.
4. **Replacement Action Recommendation.** For each matched method (from step #2) in the new version, FlowFixer queries the built map (from step #3, mapping each executed UI action to its invoked methods) to find UI actions that can trigger it. FlowFixer ranks these found UI actions, in terms of adapting the broken workflow to the new application.

1.3 Evaluation

FlowFixer works in an automatic manner and scales to realistic programs. We evaluated FlowFixer on 16 broken workflows from 5 non-trivial Swing [19] GUI applications, described in Figure 5. FlowFixer successfully repaired 15 broken workflows. In 13 workflows, the correct replacement action was FlowFixer's first suggestion. In 2 workflows, the correct replacement action was FlowFixer's second suggestion. The remaining workflow was unrepairable (the functionality had been removed from the application), but FlowFixer did not identify that fact and produced a list of possible replacement actions. FlowFixer is fast enough for practical use, taking less than 4 minutes to repair one broken workflow, on average. FlowFixer's accuracy and speed make it a promising technique.

We compared FlowFixer to an existing technique [14], which uses GUI comparison to recommend replacement UI actions. The existing technique can only repair 6 out of 16 broken workflows.

We also compared FlowFixer to an alternative technique, which uses static analysis to recommend replacement UI actions for a broken workflow. FlowFixer produced significantly better results for all workflows.

1.4 Contributions

This paper makes the following contributions:

- **Technique.** We present a technique to repair broken workflows. Our technique uses dynamic profiling, static analysis, and random testing to suggest fixes to a broken workflow (Section 2).
- **Implementation.** We implemented our technique in a tool, called FlowFixer, for Java GUI software (Section 3). Our implementation is publicly available at `http://workflow-repairer.googlecode.com`.
- **Evaluation.** We applied FlowFixer to 16 broken workflows from 5 real-world GUI applications, comprising 244580 LOC. The results show the accuracy and efficiency of FlowFixer (Section 4).

2. TECHNIQUE

This section explains FlowFixer's 4 steps, as introduced in Section 1.2 and illustrated in Figure 2.

FlowFixer takes as input an old workflow (that are valid on the old version but broken on the new version), the source code of two versions of a GUI application, and the version history between these two versions.

2.1 Profiling a Broken Workflow

FlowFixer first instruments the old version of the tested application offline by inserting code to monitor each method's execution at run time. Then, FlowFixer asks the end-user to demonstrate a workflow on the instrumented version up to the broken action. Demonstration is one of the simplest ways for an end-user to describe a workflow; it is easier than writing specifications or scripts of any form.

During instrumentation, FlowFixer distinguishes event handlers (i.e., event-handling methods) from other methods, and records both of them in the execution trace. An event handler is a special method in a GUI application that is called back by the GUI framework. In a Java application, each feasible UI action is handled by an event handler (possibly shared with other UI actions). The event handler further calls other methods to realize the desired functionality.

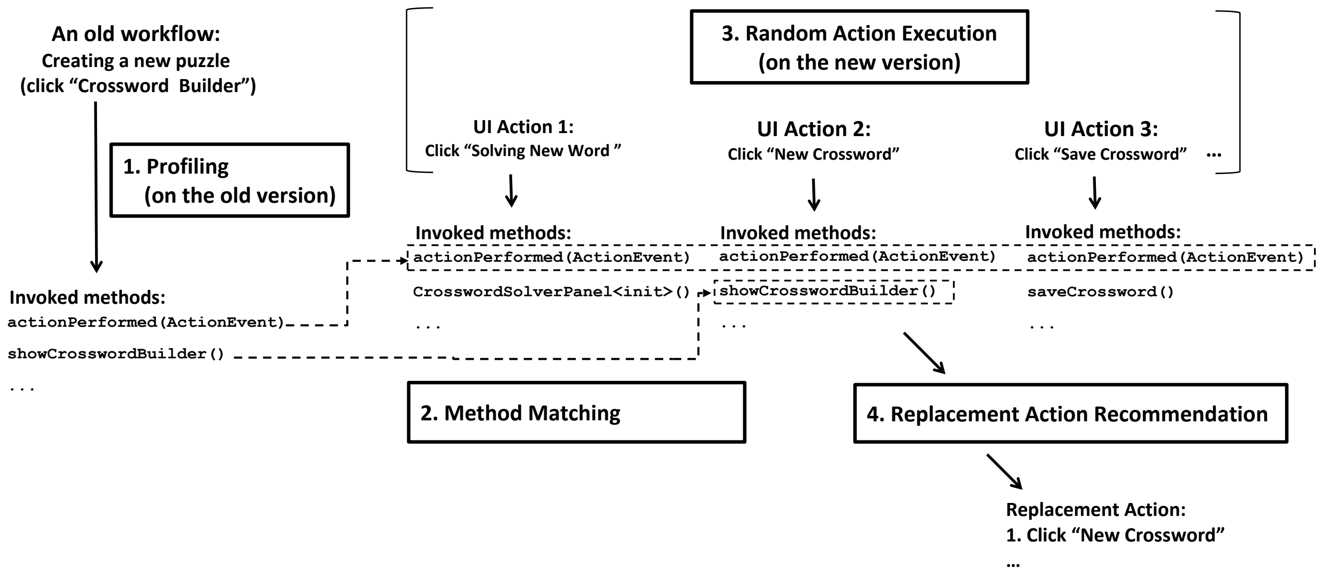


Figure 2: Illustration of FlowFixer's 4 steps in repairing a broken workflow using the Crossword example from Figure 1.
1. **The "Profiling" step (Section 2.1) determines which methods are invoked by an old workflow on the old version of the application.**
2. **The "Method Matching" step (Section 2.2) finds correspondences between methods in the old and new versions, as shown by dotted lines.**
3. **The "Random Action Execution" step (Section 2.3) generates and executes random UI actions on the new version, in order to determine which methods each UI action invokes. Step 3 is independent of steps 1 and 2 and can be performed as a preprocessing step.**
4. **The "Replacement Action Recommender" step (Section 2.4) takes the method matching and random UI execution information to output a ranked list of replacement UI actions to repair the old workflow.**

For the Crossword example, FlowFixer's top recommendation to repair the broken workflow is the UI action: click "New Crossword".

Executing the instrumented application produces an *execution trace*, which consists of a sequence of invoked methods. Take the Crossword program in Figure 1 as an example: an end-user clicks the menu item "Tools → Crossword Builder" to create a new puzzle. (Here, clicking "Tools → Crossword Builder" is a single GUI action, not two actions.) As shown in Figure 2, this old workflow invokes an event handler called `MenuListener.actionPerformed(ActionEvent)` (for short, `actionPerformed(ActionEvent)`), which further calls `showCrosswordBuilder` and other methods (omitted in Figure 2). FlowFixer records those invoked methods in an execution trace file.

2.2 Matching Methods across Versions

For each invoked event handler and other methods invoked by the broken workflow in the old version, FlowFixer finds its corresponding, possibly updated method in the new version.

Previous work has described how to match program elements across program versions [5,9,21,29,33]. We were not able to use an existing tool since the tool implementation is either unavailable [33] or does not support the latest Java version [5], or the technique itself focuses on identifying changes by a small set of refactorings [9,21] rather than finding the counterpart of an arbitrary program element. Thus, we created our own tool to perform method matching. If other techniques or tools improve, we could integrate them into FlowFixer.

The key to our approach is our observation that during a GUI application's evolution, code implementing the *same* functionality across two versions often stays relatively the same. This is particularly true for event handlers. When developers revamp a GUI, they often *reuse* an existing event handler and make it respond to a different UI event rather than re-writing the same event-handling logic from scratch. Based on this observation, FlowFixer employs three simple heuristics for method matching, and uses the first one that succeeds.

1. **Identical Method heuristic.** Return a method with the identical fully-qualified name in the new version.
2. **Similar Name heuristic.** Return all methods with a fully-qualified name whose Levenshtein string similarity [34] to the original method name is within a pre-defined threshold (default: 0.9). This heuristic handles code evolution such as refactoring (e.g., method renaming, and method pull-up).
3. **Co-evolving heuristic.** If a method is replaced by another method in the new version, the deleted method and the replacement methods are often committed in the same revision. This heuristic leverages this observation and parses the application evolution history to identify co-changed methods with the deleted one.

In Figure 2, the Identical Method Heuristic works for the Crossword program. Section 4 gives examples where the other heuristics are necessary.

Given a method in the old version, it is possible that this step will return multiple methods in the new version as the *potentially* matched ones. This does not present problems for FlowFixer, since many of the matched methods identified by heuristics are not invoked by any UI actions and thus are automatically ignored. Section 2.4 describes the detailed action recommendation algorithm.

2.3 Executing Random UI Actions

After identifying the matched methods in the new application version, FlowFixer needs to find UI actions that can trigger those matched methods. The new application's GUI can be dramatically different than the old one; thus, we cannot assume the end-users are familiar with it nor ask an end-user to explore every possible UI action on it.

To obtain information about the new application version, FlowFixer employs *random testing* to generate and execute *random UI actions* on the new version, and observes the invoked event handlers and methods in the background.

FlowFixer's random testing monitors the GUI application's state and repeatedly adds newly-available UI actions to the action queue. When random testing a GUI application, the UI action space to be

```
Input: a Java GUI application $P$
Output: a map from UI action to its invoked methods
exploreGuiApplication($P$)
 1: return exploreUIScreen(getInitialScreen($P$))
```

```
Input: a screen $S$
Output: a map from UI action to its invoked methods
exploreUIScreen($S$)
 1: $actionMap \leftarrow$ new Map⟨Action, Methods⟩
 2: $actionList \leftarrow$ getAvailableActions($S$)
 3: while $actionList$ is not empty do
 4:     $action \leftarrow$ removeRandomElement($actionList$)
 5:     if $action$ is not applicable then
 6:         continue {$action$ is disabled or on a different UI element}
 7:     end if
 8:     $methods \leftarrow$ executeAction($action$)
 9:     $actionMap$.put($action$, $methods$)
10:     if popUpModalDialog($S$) then
11:         $dialog \leftarrow$ getModalDialog($action$)
12:         $nextMap \leftarrow$ exploreUIScreen($dialog$)
13:         $actionMap \leftarrow actionMap \cup nextMap$
14:     end if
15:     $newActions \leftarrow$ getNewActions($S$)
16:     $actionList$.addAll($newActions$)
17: end while
18: dispose($S$)
19: return $actionMap$
```

Figure 3: Algorithm for randomly executing UI actions on a GUI application.

explored can be large. For the sake of efficiency, our algorithm approximates the exploration by executing each UI action (at most) once. This is because, in most cases, a UI action invokes the same event handler even when it exhibits different behaviors in different contexts.

Figure 3 sketches the random execution algorithm. The main procedure `exploreGuiApplication` starts processing at the initial screen of a GUI application. The `exploreUIScreen` procedure creates a worklist of all available UI actions (line 2). For each action that is applicable (i.e., not disabled) to the current application state, the algorithm executes it and records its invoked methods (lines 8–9). For each applicable action that requires user inputs, the `executeAction` procedure randomly chooses values from a pre-defined value pool. If the executed action creates a modal dialog, the algorithm recursively explores all available actions on the new modal dialog (lines 10–14). After executing each UI action, the algorithm adds newly-available actions (including actions that were ignored on line 5 if they become applicable again) to the worklist (lines 15–16), since executing a UI action may change the program state and enable other actions. For example, in the Crossword program, executing the action of creating a new puzzle enables a new action of saving a puzzle. After executing all applicable actions on a screen, the algorithm disposes that screen (line 18).

Our implementation of `getAvailableActions` ignores several kinds of UI events, such as key pressing, mouse moving, and window disposing. This was not a problem in our experiments (Section 4), for three reasons. First, some ignored events such as key pressing often serve as a shortcut for other actions supported by FlowFixer. Second, some ignored events such as mouse moving are usually not essential in a workflow, or if essential are not replaceable by other events. Third, other events such as window disposing are performed by the GUI framework; an end-user rarely uses them in their own workflows.

Auxiliary methods:

getBrokenHandler(*trace*): returns the first event handler recorded in the execution *trace* that is broken in the new application

getMatchedMethods(V_{old}, V_{new}, *method*): returns a list of methods (including handlers) that match *method* (Section 2.2)

getActions(*actionMap*, *methods*): reverse query on *actionMap* (produced by the algorithm in Figure 3); returns actions that invoke any one of the *methods*

getInvokedMethods(*handler*, *trace*): returns a list of methods that are invoked by *handler* in the execution *trace*; that is, the methods invoked after *handler* but before the next handler in the trace

```
Input: two versions of the GUI application: $V_{old}$ and $V_{new}$;
  the execution trace $T$ of a workflow on $V_{old}$; and $actionMap$,
  the precomputed result of exploreGuiApplication($V_{new}$)
Output: a ranked list of replacement UI actions in $V_{new}$
recommendReplacementActions($V_{old}$, $V_{new}$, $T$, $actionMap$)
 1: $handler \leftarrow$ getBrokenHandler($T$)
 2: $matchedHandlers \leftarrow$ getMatchedMethods($V_{old}$, $V_{new}$, $handler$)
 3: $handlerActions \leftarrow$ getActions($actionMap$, $matchedHandlers$)
 4: if $|handlerActions| = 0$ then
 5:     return $handlerActions$
 6: end if
 7: $weightMap \leftarrow$ new Map⟨Action, Float⟩
 8: for each $action$ in $handlerActions$ do
 9:     $weightMap[action] = \frac{1}{|handlerActions|}$
10: end for
11: $invokedMethods \leftarrow$ getInvokedMethods($handler$, $T$)
12: for each $method$ in $invokedMethods$ do
13:     $matchedMethods \leftarrow$ getMatchedMethods($V_{old}$, $V_{new}$, $method$)
14:     $actions \leftarrow$ getActions($actionMap$, $matchedMethods$)
15:     for each $action$ in $actions$ do
16:         $weightMap[action] = \max(weightMap[action], \frac{1}{|actions|})$
17:     end for
18: end for
19: return $weightMap$.sortedKeys()
```

Figure 4: Algorithm for recommending replacement UI actions for a broken workflow. The exploreGuiApplication procedure is defined in Figure 3.

Our implementation of `executeAction`, bounds each action execution to 5 seconds; thus, our random testing algorithm always terminates.

Random testing is easy to implement, scalable to large programs, and remarkably effective in practice, including in our context. However, it has no coverage guarantee. Executing random UI actions is not the only way to identify invoked methods for each UI action. Another possible way is to use static analysis to find all reachable methods for a UI action. Compared with random testing, static analysis can be sound but suffers from several limitations. First, it is hard for a static analysis to distinguish UI actions that share the same event handler. For example, in the Crossword program [4], 12 menu items share the same event handler `MenuListener.actionPerformed`. Thus, it is hard for a static analysis to decide which action has been performed when it gets invoked. Second, static analysis is conservative: it may include methods that are actually unreachable in practice and thus introduce noises. Section 4.3.1 measures the coverage achieved by random testing in our evaluation; and Section 4.3.4 empirically compares FlowFixer with an alternative approach based on static analysis, and shows that FlowFixer yields significantly better results.

2.4 Recommending Replacement UI Actions

For a broken UI action in a workflow, FlowFixer recommends a list of replacement UI actions that may complete the same workflow in the new version. The basic idea of FlowFixer's replacement UI action recommendation algorithm is to check each matched method in the new version and then infer which UI action is most likely to invoke it. The algorithm described in this section suggests one fix action for one broken action in a workflow rather than for a sequence of actions. If multiple UI actions in a workflow are broken caused by the GUI evolution, the algorithm can be used iteratively. This was not necessary in our experiments.

Figure 4 sketches the recommendation algorithm. The algorithm first analyzes a recorded execution trace to extract the event handler invoked by the broken UI action in the broken workflow (line 1). In a Java application, each UI action is handled by an event handler. Since the input execution trace is produced by the user after demonstrating the workflow up to the broken action (Section 2.1), the last recorded event handler is the event handler invoked by the broken UI action. Then, the algorithm identifies a list of UI actions in the new version that can trigger the matched event handlers (line 3). If there is no action in the action list, the algorithm concludes that the broken workflow cannot be repaired (lines 4–6). Otherwise, at least one UI action triggers the matched event handler; and the rest of algorithm deals with this case.

To estimate the likelihood of each UI action being a replacement action, the algorithm associates each UI action with a weight (line 7). The larger the weight, the more likely the action is the desired one. The algorithm identifies program elements relevant to the broken workflow (handlers on line 2, methods on line 11). The weight of a UI action is inversely proportional to the number of UI actions that trigger the program element. If a relevant method is *uniquely* invoked by one action, that action is more likely to be the desired action. On the other hand, if a relevant method is invoked by multiple actions, the likelihood of each action being the desired action decreases. Finally, the algorithm ranks the UI actions by their weights in a decreasing order and returns them (line 19). If two actions have the same weight, the algorithm ranks the action invoking more methods higher in the output.

For example, in Figure 2, the matched event handler `actionPerformed` is invoked by three UI actions on the new version, so each action's weight is 1/3. The matched method `showCrosswordBuilder` is uniquely invoked by the UI action "Click New Crossword", so its weight is 1. Thus, the algorithm ranks the action "Click New Crossword" highest.

If a replacement UI action is applicable on a window other than the current one, FlowFixer's recommendation also shows the action that triggers the other window.

Our algorithm focuses on the "uniqueness" of a method being invoked by UI actions. An alternative is to rank each replacement action based on coverage: compute the methods covered by the broken UI action, then rank replacements according to what fraction of those methods each replacement covers. Section 4.3.1 empirically compares this approach with FlowFixer's algorithm, and finds that FlowFixer's algorithm yields better results.

2.5 Discussion

We next discuss some design issues in FlowFixer.

Soundness and completeness. The FlowFixer technique is neither sound nor complete. The primary reason is that both the method matching algorithm and the random testing technique used in FlowFixer are based on heuristics. The method matching algorithm cannot guarantee to identify the counterpart of each invoked method; and the random testing algorithm cannot guarantee to find all invoked methods by a UI action. Despite such limitations, as demonstrated in Section 4, FlowFixer is still useful in repairing many real-world broken workflows.

Repairing broken GUI test scripts. Repairing broken GUI test scripts is related to, but more challenging than, repairing broken workflows. A test script can be written in a programming language. This gives it more flexibility than a workflow, for instance, permitting it to perform an action on a disabled GUI element by directly calling its underlying event handler. In addition, a test script may include other computations besides UI actions, and repairing a broken test script requires updating computations as well as UI actions. A GUI test script may be more fragile to software evolution, if it precisely encodes the position of each target GUI element. Therefore, a tiny UI change like switching the positions of two neighboring menu items can break a GUI test script, but won't affect a workflow from an end-user's perspective. Due to these difficulties, existing test script repairing techniques [14, 17, 26] fix "execution errors": they identify statements affected by GUI changes or delete unusable actions to make an obsolete test script executable, while ignoring its original semantics. By contrast, FlowFixer aims to preserve the semantics of a broken workflow. FlowFixer can be viewed as a first step toward solving the more general GUI test script repair problem.

3. IMPLEMENTATION

We implemented FlowFixer using the WALA framework [36] and the UISpec4J library [35]. FlowFixer uses WALA to perform offline instrumentation of Java bytecode. To execute UI actions and gather method invocation information, FlowFixer employs UISpec4J to transparently intercept windows in the background without actually rendering the GUI. Currently, FlowFixer supports GUI applications using the Swing framework [19], and supports UI actions on most of the built-in Swing GUI elements, such as button, menu, checkbox, tree structure, listbox, etc. When executing UI actions, FlowFixer employs several simple heuristics to skip actions on GUI elements whose label contains "Exit" or "Quit", since such UI actions often cause the whole application to abort. Although our current implementation only supports GUI applications using the Swing framework, there are no fundamental limitations that prevent the technique itself from being extended to other frameworks.

To increase robustness, when executing a UI action, FlowFixer spawns a UI action execution thread. If the thread hangs before a user-specified time limit (default: 5 seconds), FlowFixer terminates the thread and executes the next UI action.

The FlowFixer implementation is publicly available at `http://workflow-repairer.googlecode.com`.

4. EVALUATION

We evaluated four aspects of FlowFixer's effectiveness, answering the following research questions:

1. How accurate is FlowFixer in repairing broken workflows for real-world GUI applications? That is, what is the rank of the actual replacement UI actions in FlowFixer's output (Section 4.3.1)?
2. How long does it take for FlowFixer to repair a broken workflow (Section 4.3.2)?
3. How does FlowFixer's effectiveness compare to an existing approach based on GUI comparison (Section 4.3.3)?
4. How does FlowFixer's effectiveness compare to an alternative approach based on static analysis (Section 4.3.4)?

4.1 Subject Programs and Broken Workflows

We evaluated FlowFixer on 5 Java programs shown in Figure 5. We selected these programs because they are popular open-source,

Subject program	Description	Size of new version			Versions	ΔLOC	Broken workflows
		LOC	#Class	#Method			
Crossword [4]	A tool for solving crosswords	3087	19	193	0.3 → 0.35	1386	1. Create a new puzzle
JEdit [20]	A cross-platform text editor	32607	275	1382	2.5 → 2.6	5017	2. Show the previous/next file
Gantt Project [13]	A tool for project scheduling and management	55009	771	3852	2.0.1 → 2.5.4	3777	3. Zoom in/out the current chart 4. Go to the previous/next date 5. Save the current chart 6. Show critical path 7. View chart options
JabRef [18]	A tool for bibliography reference management	83447	636	3340	2.0 → 2.8.1	38992	8. Search for a record 9. Import records into a new database 10. Export records from the current database
Freemind [10]	A tool for writing mind maps	70430	354	3930	0.71 → 0.80	10757	11. Export a map as HTML 12. Export all map branches as HTML 13. Show the previous/next graph 14. Zoom in/out the current graph 15. Change a graph node property 16. Change a graph edge property

Figure 5: Subject programs and broken workflows used to evaluate FlowFixer. Column "ΔLOC" shows the number of changed lines of code between the old and versions, as counted by UNIX diff. Column "Broken workflows" lists unique broken workflows.

GUI-based applications available at SourceForge.net. Each program contains multiple GUI screens and many GUI elements, and has evolved over a long period of time (5–12 years). In addition, three of the programs (Crossword, Freemind, and Gantt Project) have been used in the GUI testing literature [26, 30].

For each application, we performed a retrospective analysis to select two versions that contain significant GUI changes. We collected all documented workflows from the old version's user manual. Workflows documented in the user manual, though by no means completely covering an application's functionality, are often among the most useful and important ones. Then, we tried to perform each collected workflow on the new version to check whether the workflow broke or not. We considered a workflow broken if a GUI element used in the old workflow did not appear in the new version on the same panel (e.g., the same menu, or dialog). Location changes, such as swapping the order of two menu items, do not count as breaking a workflow: even though they may affect a GUI test script [7], they are unlikely to confuse an end-user. We evaluated every broken workflow we found; we did not select only workflows on which FlowFixer works well. The workflows are of reasonable complexity: each workflow contains 1 to 8 steps with one broken action, and some are non-trivial for a human to fix (see Figure 7 for an example).

Overall, there were 356 workflows in the user manuals, and 70 of these (20%) were broken in the new version. Some of the broken workflows had the same root cause: (1) Two workflows might use the same UI element that was changed. (2) If all items of one menu are moved to another, only one broken workflow related to those two menus is listed. (3) A change in the way zooming is done appears only once per application, even though zooming in and out are distinct UI actions.

There were 16 root causes in total. Figure 5 lists one representative broken workflow for each root cause.

4.2 Evaluation Procedure

For each subject program, we first used FlowFixer to instrument both old and new versions based on the strategy described in Section 2.1. We demonstrated each broken workflow on the instrumented old version to obtain an execution trace. After that, FlowFixer analyzed the obtained execution trace and two program versions. FlowFixer works in a fully automatic way to repair a given workflow, and has two possible outcomes: it either recommends a ranked list of replacement actions or concludes that the broken

Program	Workflow ID	FlowFixer Rank	Root Cause
Crossword	1	1	A menu item is re-named and moved to another menu
JEdit	2	1	A menu item is moved to another menu
Gantt Project	3	1	A toolbar icon is replaced by button
	4	1	A toolbar icon is replaced by button
	5	2	The original UI action is replaced by a different action (see Figure 7 for details)
	6	1	An icon is replaced by a button
	7	X	This functionality is removed
JabRef	8	1	A menu item is moved to another menu
	9	1	A menu item is re-named
	10	2	A menu item is re-named
Freemind	11	1	A menu item is re-named and moved to another menu
	12	1	A menu item is re-named and moved to another menu
	13	1	A menu item is moved to another menu
	14	1	A menu item is moved to another menu
	15	1	A menu item is moved to another menu
	16	1	A menu item is moved to another menu

Figure 6: Experimental results in repairing broken workflows. Column "Rank" shows the absolute rank of the actual replacement UI action in FlowFixer's output (lower is better), in which "X" means FlowFixer output a list of replacement UI actions but the workflow is actually unrepairable. Column "Root Cause" shows the root cause of the broken workflow.

workflow cannot be repaired. For both cases, we manually examined FlowFixer's output to determine its correctness.

We made a simple change to each subject program by removing the splash screen. This is a pure implementation consideration, because the UISpec4J library cannot handle splash screens before a GUI application launches. The code edit affected 10 lines of code in total across the 5 applications, and it did not modify any GUI functionality.

Our experiments were run on a 2.67GHz Intel Core PC with 4GB physical memory, running Windows 7.

4.3 Results

4.3.1 *Accuracy*

As shown in Figure 6, FlowFixer is highly effective in repairing broken workflows. FlowFixer successfully repaired 15 broken work-

Number of workflows that FlowFixer can repair		
Identical Method	Identical Method + Similar Name	Identical Method + Similar Name + Co-evolving
11	13	15

Figure 8: Number of broken workflows that FlowFixer can repair using the three heuristics in Section 2.2.

Program	Random UI Action Execution	
	Event Handler Coverage	Overall Method Coverage
Crossword	80%	27%
JEdit	52%	42%
Gantt Project	24%	26%
JabRef	10%	20%
Freemind	42%	26%
Average	42%	28%

Figure 9: Experimental results in executing random UI actions. Columns "Event Handler Coverage" and "Overall Method Coverage" show the coverage of event handlers and methods (including event handlers) in random UI action execution, respectively.

flows. For 13 workflows, the correct action was FlowFixer's first suggestion. For 2 workflows, the correct action was FlowFixer's second suggestion. For 1 workflow, no repair was possible (the functionality had been removed from the application), but FlowFixer incorrectly output a list of possible replacement UI actions.

We use a broken workflow from the Gantt Project program as an example to illustrate FlowFixer's effectiveness. As shown in Figure 7, there are significant GUI changes between Gantt Project versions 2.0.1 and 2.5.4. After the GUI evolution, the toolbar button used to save the current chart state in version 2.0.1 (Figure 7(a)) disappeared in version 2.5.4 (Figure 7(b)). In version 2.5.4, the UI action to save the current chart state is changed from "click a button" to "fill a table cell". The difficulty of finding this replacement action is further exacerbated by the fact that the table to receive the desired UI action is in a different dialog that only becomes visible after users click another button. For this example, FlowFixer outputs the desired UI action as the second suggestion in its report. The key fact explored by FlowFixer during repair is that a method named `UndoableEditImpl.createTemporaryFile`, invoked by the broken workflow on the old version, exists on the new version and is uniquely invoked by the "table cell filling" UI action.

One broken workflow in Gantt Project is not repairable in the new version. In version 2.0.1, Gantt Project provided a menu item called "Chart options" as a shortcut to configure the current Gantt chart, but this menu item was removed in version 2.5.4. We checked the source code of version 2.5.4, and confirmed that code implementing this shortcut was no longer reachable by any event handler. For this workflow, FlowFixer produces a list of replacement UI actions because the Method Matching step (Section 2.2) returns a matched method in the new version that is not actually relevant. Future work should remedy this problem. One possible way is to improve the precision of the method matching algorithm by further analyzing a method's calling context [5].

Figure 8 shows the effects of using different method matching heuristics (Section 2.2) in repairing broken workflows. Our experimental results confirm the observation that the underlying code implementing the same functionality often stays relatively the same between versions, even when the codebase evolves substantially. For more than 75% of the workflows in our experiment, two method-name-based matching heuristics are sufficient. Figure 2 shows an example of method matching using the Identical Method heuristic. The Similar Name heuristic handles a change in the Gantt Project program, when the method to handle panel scrolling event is changed from `ScrollingManagerImpl.scrollRight` to `ScrollingManagerImpl.scrollBy`. The Co-evolving heuristic permits FlowFixer to identify a matching in the Freemind [10] program, when the event handler for editing a graph edge is changed from `EdgeStyleAction.actionPerformed(ActionEvent)` to `EdgeStyleAction.apply(MapAdapter, MindMapNode)`.

Figure 9 shows the experimental results of executing random UI actions. On average, the random UI action execution algorithm described in Section 2.3 achieved 42% and 28% coverage for event handlers and overall methods, respectively. The coverage is not high, but still useful for recommending replacement UI actions. The low coverage is because most of the uncovered event handlers and methods require a UI action to satisfy some pre-condition, such as executing on a specific application state, and the random testing algorithm was not able to satisfy the pre-condition. Future work should remedy this problem. One possible approach is to integrate FlowFixer with existing symbolic analyses [16] to construct the required desired application states. Despite this limitation, methods invoked by a UI action even without satisfying all pre-conditions often give enough information to disambiguate different UI actions. Take the Crossword program from Figure 2 as an example. The `showCrossworldBuilder` method contains code to perform pre-condition checking. Even though a randomly-generated UI action may fail to bypass the pre-condition checking code and miss other methods called by `showCrossworldBuilder`, knowing an action can invoke method `showCrossworldBuilder` is sufficient to disambiguate it from others. This is because the `showCrossworldBuilder` method is uniquely invoked by the action of clicking "New Crossword".

We next evaluate an alternative approach to recommend replacement actions. This approach replaces FlowFixer's algorithm in Section 2.4 with a ranking heuristic based on the proportion of invoked methods in response to the broken UI action that are covered by each alternative replacement UI action, with higher ranks assigned to actions that cover more methods. This approach degraded the accuracy of FlowFixer for every broken workflow shown in Figure 6. The primary reason is that each UI action can invoke many utility methods, so two UI actions sharing a large proportion of (utility) methods do not necessarily indicate that they perform similar tasks. By contrast, FlowFixer's heuristic focuses on the "uniqueness" of a method being invoked by UI actions, and yields better results.

Summary. FlowFixer repairs realistic broken workflows with high accuracy for evolving GUI applications with non-trivial GUI changes.

4.3.2 Time Cost

We measured FlowFixer's performance in two ways: the time cost in executing random UI actions for each subject program and the time cost in recommending replacement UI actions for each broken workflow. Figure 10 shows the results.

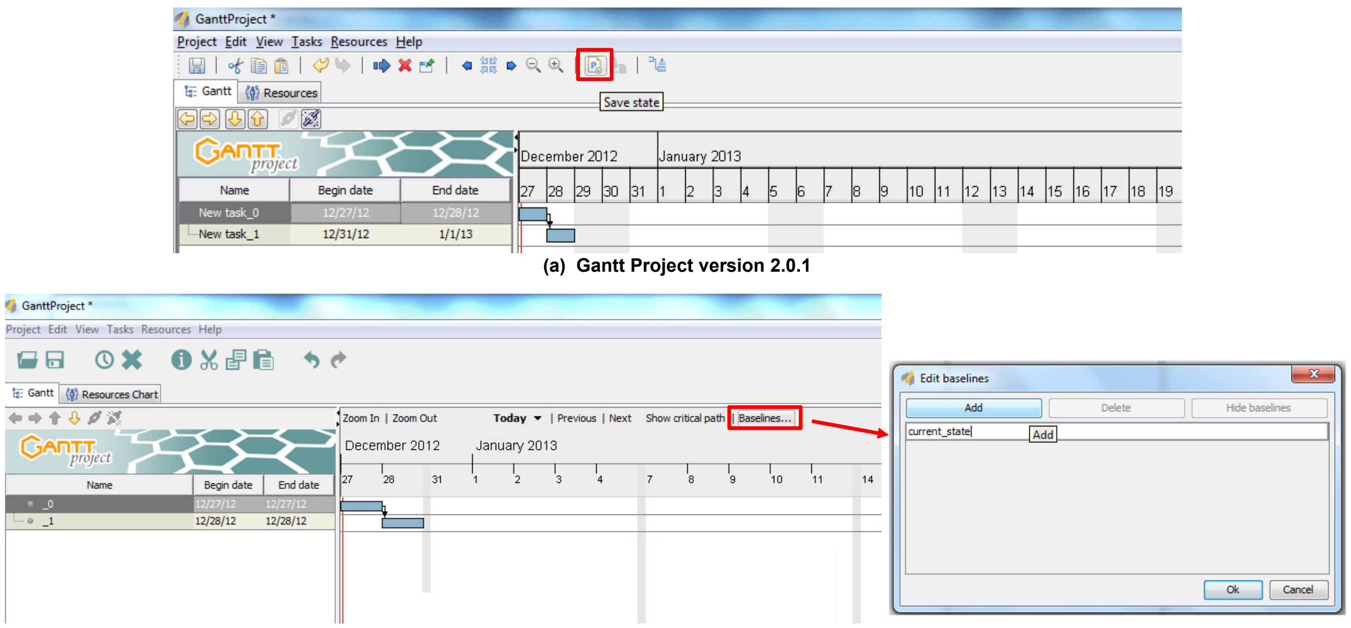
On average, FlowFixer uses 27 minutes to execute random UI actions on each subject program. However, random testing is a one-time cost per program and the computed results can be cached to share across workflows.

FlowFixer spends an average of 3.2 minutes to recommend replacement UI actions for one workflow. The time used to repair a workflow is roughly proportional to the number of methods invoked by a workflow, rather than the size of the subject program.

Summary. FlowFixer repairs realistic broken workflows with acceptable time cost.

4.3.3 Comparison with a GUI-Comparison Approach

We compared FlowFixer with an existing approach called REST [14]. We chose REST because it is a recent technique targeting a



(a) Gantt Project version 2.0.1

(b) Gantt Project version 2.5.4

Figure 7: The GUI change between two Gantt Project versions breaks workflow #5: saving the current chart state. In version 2.0.1, users click a toolbar button called "Save state" (highlighted in (a)) to complete this workflow. However, in version 2.5.4, the GUI change forces users to first click a toolbar button "Baselines..." (highlighted in (b)), then input a name in the table cell in the popup modal dialog to save the current state. Our FlowFixer technique can suggest this replacement action to repair the broken workflow.

Program	Workflow ID	Time (seconds) Action Execution	Time (seconds) Recommendation
Crossword	1	22	37
JEdit	2	3611	47
Gantt Project	3	2664	205
	4		324
	5		353
	6		299
	7		319
JabRef	8	335	189
	9		118
	10		402
Freemind	11	700	65
	12		112
	13		150
	14		192
	15		110
	16		122
Average		1642	190

Figure 10: FlowFixer's performance in repairing broken workflows. The time cost has been divided into two parts: executing random UI actions for each subject program (column "Action Execution") and recommending replacement UI actions for each workflow (column "Recommendation").

similar problem. REST is a technique to maintain GUI test scripts. It first compares the GUIs of two application versions to identify changed GUI elements. Then, it localizes GUI test script statements that are affected by the changed GUI elements. REST has a different focus than FlowFixer: it aims to identify affected test script statements rather than repairing an affected test script.

We extended REST to support repairing a broken workflow as follows. For each affected UI action in a broken workflow, we use REST to identify the same GUI element in the new version, and then recommend UI actions on this GUI element as the replacement actions. For instance, if a menu item used in the broken workflow disappears in the new version, we use REST to find a menu item with the same label in the new version as the replacement UI action.

Program	Workflow ID	Repairable?	Can repair the workflow? FlowFixer	Can repair the workflow? REST
Crossword	1	Yes	Yes	No
JEdit	2	Yes	Yes	Yes
Gantt Project	3	Yes	Yes	No
	4	Yes	Yes	No
	5	Yes	Yes	No
	6	Yes	Yes	No
	7	No	No	No
JabRef	8	Yes	Yes	Yes
	9	Yes	Yes	No
	10	Yes	Yes	No
Freemind	11	Yes	Yes	No
	12	Yes	Yes	No
	13	Yes	Yes	Yes
	14	Yes	Yes	Yes
	15	Yes	Yes	Yes
	16	Yes	Yes	Yes

Figure 11: Experimental results in comparing FlowFixer with a REST-based technique [14]. Each cell in columns "FlowFixer" and "REST" shows whether the corresponding technique can repair a broken workflow or not. FlowFixer repairs 15 workflows in total. By contrast, the REST-based technique repairs 6 workflows in total and incorrectly states that 9 workflows are un-repairable. For workflow 7 that is actually un-repairable, FlowFixer incorrectly outputs a list of replacement UI actions, while the REST-based technique correctly states that workflow cannot be repaired. The time cost of the REST-based technique is slightly lower than FlowFixer and is omitted for brevity.

Figure 11 shows the experimental results. FlowFixer repaired 16 broken workflows, while the REST-based technique only repaired 6 broken workflows. The primary reason for REST's poor performance is that a GUI can change substantially between two application versions and such non-trivial change is often beyond REST's ability to identify the replacement UI actions in the new

Program	Workflow ID	FlowFixer		Static Analysis	
		Rank	Time (s)	Rank	Time (s)
Crossword	1	1	59	7	194
JEdit	2	1	3658	70	939
Gantt Project	3	1	2869	13	1951
	4	1	2988	11	2402
	5	2	3017	25	3078
	6	1	2963	25	2414
	7	X	2983	X	2546
JabRef	8	1	544	N	4040
	9	1	473	22	4013
	10	2	737	N	4134
Freemind	11	1	765	20	978
	12	1	812	19	1132
	13	1	850	21	1226
	14	1	892	22	1255
	15	1	810	14	1116
	16	1	822	13	1198
Average		1.1	1682	22	2039

Figure 12: Comparison of FlowFixer with a static analysis-based approach described in Section 4.3.4. Column "Rank" shows the absolute rank of the actual replacement UI action in the output. "N" represents that the actual replacement UI action is not in the output. "X" represents that the workflow is un-repairable but the tool suggests a replacement UI action. Both "N" and "X" are dropped when averaging ranks. The "Rank" and "Time (s)" columns for FlowFixer's results are taken from Figure 6 and Figure 10, respectively. For fair comparison, the time cost of FlowFixer includes both the random UI execution time and the replacement action recommendation time.

version. For example, the text on a GUI element can be modified (e.g., the Crossword example in Figure 1), a GUI element can be replaced by a different GUI element (e.g., workflows 3, 4, and 6), and the action on a GUI element to complete the same workflow can be changed to a different action (e.g., the Gantt Project example in Figure 7). The 6 workflows that REST can repair all share the same root cause: a menu item is moved from one menu to another without changing its label.

We did not compare FlowFixer with other related GUI testing [27, 39, 40] and GUI test script repairing approaches [17, 26] because these approaches target a rather different problem than FlowFixer. Automated GUI testing frameworks like GUITAR [27, 39, 40] aim to systematically validate a GUI application's functionality. Test script repairing approaches like [17, 26] aim to repair *unusable* test scripts by simply removing or updating affected statements without preserving the original testing semantics, and thus are inapplicable to broken workflow repair. By contrast, FlowFixer recommends replacement UI actions to complete the *same* workflow. In addition, some techniques [17, 26] only work for GUI tests generated by the techniques themselves. It is unknown whether such techniques can be generalized to support repairing realistic broken workflows.

Summary. Comparing GUIs of two application versions without analyzing the program is insufficient in repairing realistic broken workflows. A program-analysis-based approach like FlowFixer can produce significantly better results.

4.3.4 Comparison with Static Analysis

As described in Section 2.3, FlowFixer uses random testing, a typical dynamic analysis, to identify methods that can be invoked by a UI action. Another possible way to do so is to statically examine the source code to identify reachable methods for each UI action. Compared with random testing, using static analysis is conservative and sound: it outputs results that describe the program's behavior, no matter on what inputs or in what environment the program is run. However, a conservative static analysis may output methods that are actually unreachable by any UI action.

To investigate the trade-offs between using dynamic and static analyses, we compared FlowFixer with an alternative approach based on static analysis. The alternative approach replaces the random UI action execution step (Section 2.3) with the following static analysis. The static analysis examines the source code to build a call graph for the program and identifies possible event handlers for each GUI element declaration. After that, the static analysis follows the call graph to identify all reachable methods from each event handler and treats them as reachable methods by a UI action.

Figure 12 shows the experimental results. The static analysis-based approach produces less accurate results. There are two primary reasons for this. First, an event handler may be shared by many GUI elements. For example, in Crossword, 12 UI actions share the same event handler `MenuListener.actionPerformed`. Thus, without executing the program, it is hard for a static analysis to precisely reason about the correct UI action from an invoked event handler. Second, static analysis is conservative and can include many methods that are actually unreachable by UI actions. This causes the recommendation algorithm (Figure 3) to recommend UI actions that cannot actually invoke a matched method at runtime.

As shown in Figure 12, the cost of static analysis is comparable to FlowFixer's. The majority of the time was spent traversing the call graph to find reachable nodes for every UI action. The static call graph contains 26639–44407 nodes when using the RTA call graph construction algorithm [2]. We did not use more expensive algorithms like k-CFA ($k > 1$), because they do not scale to our subject programs.

Summary. Random testing, though neither sound nor complete, can provide more accurate and useful results than static analysis.

4.4 Discussion

Threats to validity. There are several threats to the validity of our evaluation.

The 5 Java programs and their changes might not be representative, though some were used in previous research. Likewise, the 16 broken workflows might not be representative, even though we evaluated every broken workflow we found. For example, each broken workflow we found contains only one one broken action. Furthermore, FlowFixer ignores some UI actions, such as mouse moving and key pressing. Thus, we do not claim the results will apply for every workflow and GUI change.

FlowFixer's effectiveness depends on the effectiveness of the method matching algorithm and random testing. It may yield less useful results for GUI programs with significant code changes, or for programs with constrained interfaces (that random testing fails to provide useful results). However, different algorithms can be plugged into FlowFixer.

Our evaluation only compared FlowFixer with two other approaches. Comparing with other analyses or tools might yield different observations.

Our evaluation focuses on FlowFixer's algorithm for workflow repairing. A future user study should evaluate whether FlowFixer helps users.

Experimental conclusions. We have three chief findings. (1) FlowFixer is useful in recommending replacement UI actions to repair broken workflows. (2) Comparing GUIs of versions to identify replacement actions is insufficient in practice. (3) Compared with static analysis, random testing yields more accurate and useful results for the broken workflow repair problem.

5. RELATED WORK

Work related to this paper falls into three main categories; (1) analyzing and testing GUI applications; (2) techniques for supporting GUI software evolution; and (3) automated software error patching techniques.

5.1 Analyzing and Testing GUI Applications

Various techniques have been developed to automate GUI testing including model creation [12], test generation [1,44], test execution [39], and test oracle selection [27]. For example, GUITAR [27,39,40] is a GUI testing framework for Java and Microsoft Windows applications; it generates event sequence-based test cases for GUI programs using a structural event generation graph. Gross et al. [15] proposed a search-based test generator for GUI applications. Their approach maximizes coverage metrics, and avoids false failures by constructing tests at the system level. The UI action execution algorithm implemented in FlowFixer (Section 2.3) can be treated as a simple way to exercise a GUI application with the goal of increasing UI action coverage. However, FlowFixer's UI action execution algorithm does not aim to increase source code coverage or find new bugs. As demonstrated in the experiments, this simple algorithm works well for our problem. More sophisticated testing algorithms [15,39] could be incorporated if necessary.

Stabilizer, proposed by Michail and Xie [30], is a tool that helps users avoid bugs in GUI applications. Stabilizer monitors a user's actions in the background, and gives a warning as well as the opportunity to abort the action, when a user attempts an action that has led to problems in the past. Stabilizer aims to *prevent* an existing bug from happening again. By contrast, FlowFixer aims to *repair* a broken workflow.

5.2 Supporting Software Evolution

As software evolves, its behavior must be validated. Regression test selection and prioritization [38] indicate which tests need to be executed for a changed program. When a regression test fails, developers need to understand its root cause [23]. To address this problem, many debugging techniques are developed for evolving software. For example, Delta Debugging aims to find a minimal subset of changes that still makes the test fail [41,43]. Test minimization techniques [22,42] simplify the failed test to ease comprehension for developers. Recently, Qi et al. [33] proposed a symbolic execution-based debugging approach to synthesize new inputs that differ marginally from the failing input in their control flow behavior, then compare the execution traces of the failing input and the new inputs to obtain critical clues to the root-cause of the failure. All these existing techniques focus on helping software *developers* understand divergent program behaviors between versions, rather than giving a suggestion to software *end-users* about a broken workflow.

The evolution of software (in particular a framework) can break client programs. Finding suitable replacements for software elements that were accessed by a client program and deleted as part of the software's evolution is an important task. Several approaches have been proposed to address this problem. For example, SemDiff [5] is a technique to recommend adaptive changes for clients of framework code that has evolved in a way that is not backward-compatible. The key idea of SemDiff is analyzing how the framework adapts to its own changes, and recommending similar adaptations. Compared with existing matching techniques, the method matching heuristics used in FlowFixer (Section 2.2) are specifically designed for the workflow repair problem. It uses the observation that underlying code implementing the same functionality often stays the same between versions. Investigating the benefits of more sophisticated matching techniques [5,29] is future work.

There is some research on maintaining GUI code. Li and Wohlstadter [24] used dynamic information to visualize generated GUI elements in a GUI editor and to help GUI editors to map source code to GUI views when code changes. Recently, Wang et al. [37] proposed an approach to automating presentation changes in dynamic web applications. Their approach aims to help a developer perform presentation changes in a dynamic web application in a way similar to performing presentation changes in static web pages. The proposed technique recommends GUI-related code changes by mapping changes from the program output to its source code using dynamic analysis and check the safety of the changes through static and dynamic analysis. Compared to FlowFixer, these techniques work in a different granularity with a rather different goal. They aim to identify code fragment, or GUI elements that are affected during GUI application evolution, but are not applicable in repairing broken workflows encountered by software end-users.

A number of approaches have been proposed to automatically repair unit tests [6] and GUI test scripts [3,11,14,17,26]. Memon et al. proposed a technique that models GUIs with event-flow graphs and brings obsolete GUI tests back in sync with the updated application version [26]. To help in understanding and maintaining GUI test scripts, Fu et al. presented a technique to infer GUI widget types in a test script and map them to code fragments [11]. Grechanik et al. proposed a GUI-comparison-based approach to identifying GUI changes between two application versions and automatically evolving test scripts [14]. Recently, Huang et al. employed a genetic algorithm to repair broken GUI test scripts [17]; and Brett et al. [7] introduced a set of GUI refactoring types to help developers accommodate the changes both in GUI views and their corresponding controllers. However, there are several significant difference between existing work and FlowFixer. First, most approaches [7,11,14] only remind users of the affected program elements or test scripts during software evolution, and do not give any repair suggestion. Furthermore, as we show in Section 4.3.3, extending such approaches can only repair a small number of real broken workflows. Second, some approaches [6] repair a test by updating the oracle part through monitoring and recording the test execution results. Those approaches are not applicable to the problem of repairing broken workflows, since a workflow breaks because some GUI elements disappear in the new version rather than violating some pre-defined oracles. Third, other approaches [17,26] repair GUI test scripts by deleting or updating the obsoleted test suites without preserving the test intentions, while FlowFixer recommends replacement UI actions in the new version to complete the same workflow. In addition, some approaches work on the test suites that are generated by the tool itself. It is unclear whether they can be extended to repair real-world broken workflows as FlowFixer does.

5.3 Patching Software Errors

Automatically repairing broken workflows can be viewed as a special case of automated error patching. Research in the area of automated software error patching has ranged from using formal specifications and repair templates to patch errors [8]; to automatically inserting code to handle overflow errors using genetic programming [17]; to using similar context information to repair runtime exceptions [25]; to enforcing software behaviors at runtime [32].

Compared to FlowFixer, existing techniques usually search for good patches using specifications or test oracles as the criterion, and are concerned with bugs in the program that result in erroneous program states. By contrast, FlowFixer does not try to fix the program itself. Instead, it repairs a broken workflow caused by UI changes that are not related to any erroneous program states.

6. CONCLUSION

This paper presented a practical technique (and its tool implementation, called FlowFixer) for repairing broken workflows for evolving GUI applications. Our experimental results show that FlowFixer is accurate and efficient. The source code of FlowFixer is publicly available at: `http://workflow-repairer.googlecode.com`.

We consider FlowFixer as one step towards improving the usability of GUI applications. As future work, we plan a user study to evaluate FlowFixer's usefulness to end-users. A challenge will be finding study participants who are familiar with an old version of a subject program. We are also interested in investigating some possible downstream applications of the FlowFixer technique. As an example, FlowFixer could automatically update the documented workflows in the user manual as a GUI application evolves.

7. ACKNOWLEDGEMENTS

We thank Forian Gross for discussion about UISpec4J. This work was supported in part by ABB Corporation and NSF grants CCF-1016701 and CCF-0963757.

8. REFERENCES

[1] S. Arlt, A. Podelski, C. Bertolini, M. Schaf, I. Banerjee, and A. Memon. Lightweight static analysis for GUI testing. In *ISSRE*, 2012.

[2] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proc. OOPSLA*, 1996.

[3] S. R. Choudhary, D. Zhao, H. Versee, and A. Orso. WATER: Web Application TEst Repair. In *ETSE*, 2011.

[4] Crossword. `http://sourceforge.net/projects/crosswordsage/`.

[5] B. Dagenais and M. P. Robillard. Recommending adaptive changes for framework evolution. *TOSEM*, 20(4), 9 2011.

[6] B. Daniel, D. Dig, T. Gvero, V. Jagannath, J. Jiaa, D. Mitchell, J. Nogiec, S. H. Tan, and D. Marinov. ReAssert: a tool for repairing broken unit tests. In *ICSE*, 2011.

[7] B. Daniel, Q. Luo, M. Mirzaaghaei, D. Dig, D. Marinov, and M. Pezzè. Automated GUI refactoring and test script repair. In *ETSE*, 2011.

[8] B. Demsky, M. D. Ernst, P. J. Guo, S. McCamant, J. H. Perkins, and M. Rinard. Inference and enforcement of data structure consistency specifications. In *ISSTA*, 2006.

[9] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings in evolving components. In *ECOOP*, 2006.

[10] Freemind. `http://freemind.sourceforge.net`.

[11] C. Fu, M. Grechanik, and Q. Xie. Inferring types of references to GUI objects in test scripts. In *ICST*, 2009.

[12] S. Ganov, C. Killmar, S. Khurshid, and D. E. Perry. Event listener analysis and symbolic execution for testing GUI applications. In *ICFEM*, 2009.

[13] Gantt Project. `http://www.ganttproject.biz/`.

[14] M. Grechanik, Q. Xie, and C. Fu. Maintaining and evolving GUI-directed test scripts. In *ICSE*, 2009.

[15] F. Gross, G. Fraser, and A. Zeller. Search-based system testing: high coverage, no false alarms. In *ISSTA*, 2012.

[16] W. Halfond, S. Anand, and A. Orso. Precise interface identification to improve testing and analysis of web applications. In *ISSTA*, 2009.

[17] S. Huang, M. B. Cohen, and A. M. Memon. Repairing GUI test suites using a genetic algorithm. In *Proc. ICST*, 2010.

[18] JabRef. `http://jabref.sourceforge.net/`.

[19] JDK Swing Framework. `http://docs.oracle.com/javase/6/docs/technotes/guides/swing/`.

[20] JEdit. `http://www.jedit.org/`.

[21] M. Kim, D. Notkin, and D. Grossman. Automatic inference of structural changes for matching across program versions. In *ICSE*, 2007.

[22] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer. Efficient unit test case minimization. In *ASE*, 2007.

[23] B. Li, X. Sun, H. Leung, and S. Zhang. A survey of code-based change impact analysis techniques. In *STVR*, 2012.

[24] P. Li and E. Wohlstadter. View-based maintenance of graphical user interfaces. In *AOSD*, 2008.

[25] S. Mani, V. S. Sinha, P. Dhoolia, and S. Sinha. Automated support for repairing input-model faults. In *ASE*, 2010.

[26] A. M. Memon. Automatically repairing event sequence-based GUI test suites for regression testing. *ACM Trans. Softw. Eng. Methodol.*, 18(2):4:1–4:36, Nov. 2008.

[27] A. M. Memon, M. E. Pollack, and M. L. Soffa. Automated test oracles for GUIs. In *Proc. FSE*, 2000.

[28] A. M. Memon and M. L. Soffa. Regression testing of GUIs. In *FSE*, 2003.

[29] S. Meng, X. Wang, L. Zhang, and H. Mei. A history-based matching approach to identification of framework evolution. In *ICSE*, 2012.

[30] A. Michail and T. Xie. Helping users avoid bugs in GUI applications. In *ICSE*, 2005.

[31] Microsoft Office Community Forums. `http://support.microsoft.com/gp/gp_newsgroups_master`.

[32] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *SOSP*, 2009.

[33] D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani. Darwin: an approach for debugging evolving programs. In *FSE*, 2009.

[34] The SimMetrics String Similarity Metric Library. `http://sourceforge.net/projects/simmetrics/`.

[35] UISpec4J. `http://www.uispec4j.org/`.

[36] WALA. `http://wala.sourceforge.net`.

[37] X. Wang, L. Zhang, T. Xie, Y. Xiong, and H. Mei. Automating presentation changes in dynamic web applications via collaborative hybrid analysis. In *FSE*, 2012.

[38] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Softw. Test. Verif. Reliab.*, 22(2):67–120, Mar. 2012.

[39] X. Yuan, M. B. Cohen, and A. M. Memon. GUI interaction testing: Incorporating event context. *IEEE TSE*, 37(4), 2011.

[40] X. Yuan and A. M. Memon. Using GUI run-time state as feedback to generate test cases. In *Proc. ICSE*, 2007.

[41] A. Zeller. Yesterday, my program worked. today, it does not. why? *SIGSOFT Softw. Eng. Notes*, 24(6):253–267, 1999.

[42] S. Zhang. Practical semantic test simplification. In *Proc. ICSE (NIER)*, 2013.

[43] S. Zhang, Y. Lin, Z. Gu, and J. Zhao. Effective identification of failure-inducing changes: A hybrid approach. In *PASTE*, 2008.

[44] S. Zhang, D. Saff, Y. Bu, and M. D. Ernst. Combined static and dynamic automated test generation. In *Proc. ISSTA*, 2011.