SITAR: GUI Test Script Repair

Zebao Gao and Zhenyu Chen, IEEE Member
Yunxiao Zou and Atif M. Memon, IEEE Member

Abstract—System testing of a GUI-based application requires that test cases, consisting of sequences of user actions/events, be executed and the software's output be verified. To enable automated re-testing, such test cases are increasingly being coded as low-level test scripts, to be replayed automatically using test harnesses. Whenever the GUI changes—widgets get moved around, windows get merged—some scripts become unusable because they no longer encode valid input sequences. Moreover, because the software's output may have changed, their *test oracles*—assertions and checkpoints—encoded in the scripts may no longer correctly check the intended GUI objects. We present *ScrIpT repAireR* (*SITAR*), a technique to automatically *repair* unusable low-level test scripts. *SITAR* uses reverse engineering techniques to create an abstract test for each script, maps it to an annotated event-flow graph (EFG), uses repairing transformations and human input to repair the test, and synthesizes a new "repaired" test script. During this process, *SITAR* also repairs the reference to the GUI objects used in the checkpoints yielding a final test script that can be executed automatically to validate the revised software. *SITAR* amortizes the cost of human intervention across multiple scripts by accumulating the human knowledge as annotations on the EFG. An experiment using QTP test scripts suggests that *SITAR* is effective in that 41–89% unusable test scripts were repaired. Annotations significantly reduced human cost after 20% test scripts had been repaired.

Index Terms—GUI testing, GUI test script, test script repair, human knowledge accumulation

1 Introduction

A significant problem with test automation of software applications with a Graphical-User Interface (GUI) front-end is that an unacceptably large number of tests may become unusable each time the software is modified [1], [2], [3]. Yet, *GUI testing* is unavoidable: when the only means of interaction with a software is its GUI, *system testing*, i.e., testing the software as a whole [4], requires it be tested with its GUI. A *GUI test case* consists of sequences of *user actions/events* that are executed on the software via its GUI widgets, and *checkpoints* that determine whether the software executed correctly. Such system tests are tightly coupled with the GUI's structure, i.e., they refer to specific widgets in the GUI and encode sequences ("*First click on the File menu, then click on Open menu-item, then select a file, and click on the Open button*") that are allowed by the GUI's workflow.

- *Zebao Gao was previously at the State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China. He is currently a PhD student at the Department of Computer Science, University of Maryland, College Park, MD, USA. E-mail: gaozebao@cs.umd.edu*
- *Zhenyu Chen, the corresponding author, is with the State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China. E-mail: zychen@nju.edu.cn*
- *Yunxiao Zou was previously at the State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China. He is currently a PhD student at the Department of Computer Science, Purdue University, West Lafayette, IN, USA. E-mail: zou41@purdue.edu*
- *Atif M. Memon is with the Department of Computer Science, University of Maryland, College Park, MD, USA. E-mail: atif@cs.umd.edu*

During maintenance, if the GUI changes, some tests become *unusable* either because (1) the event sequences they encode are no longer allowed on the modified GUI or (2) their checkpoints (assertions) no longer correctly check the intended GUI objects.

Unusable test cases are not problematic when GUI testing is done manually. Human testers execute the test cases as per a test plan and manually verify the correctness of the output [5]. If what they see on the GUI differs from what the test plan describes, they are often able to use common sense about simple changes, deduce whether they have encountered a bug or a deliberate change, and, if needed, revise the test plan.

In contrast, unusable test cases are a significant problem for automation. If an automated test harness encounters anything (an unexpected screen, a widget) different from what it expects, it simply fails or hangs. While automation is desirable because scripted tests can be run many times, its benefits are quickly dwarfed by high maintenance costs when large numbers of tests become unusable and require re-coding or re-recording [1] each time the GUI is modified.

In previous work, we partially dealt with the problem of unusable test cases. However, we only focused on high-level model-based test cases, represented as abstract events, not as scripts. Originally generated automatically using model-based approaches [6], some of these test cases in our previous work became unusable due to software modifications. We *repaired* these model-based test cases [1] by developing a new technique based on "repairing transformations" that matched unusable model-level test cases and transformed them into new usable model-level test cases.

Repairing model-based test cases has limited prac-

tical value in an Industry that relies largely on manually scripted and captured test cases. Most GUI tests used in Industry are coded as scripts (e.g., VBScript) or manually recorded to be automatically replayed by test harnesses/tools (e.g., HP QuickTest Professional (QTP), Selenium [7], [8]). These scripts suffer from maintenance problems and need to be repaired. Repairing scripted test cases is very important and relevant because they are usually based on carefully crafted use cases and functional requirements. Human testers invest valuable time transcribing their domain knowledge and experience into the test scripts and checkpoints to comprehensively cover business logic, making the scripts complex and valuable.

Unfortunately, our previous technique [1] does not directly apply to manually scripted test cases for a number of reasons. First, the model—event-flow graph (EFG) [6]—that formed the basis for the repair was inadequate as it lacked vital information (e.g., state, annotations) needed for script repair. Second, the way we obtained the EFGs from the GUIs (using reverse engineering via *GUI Ripping* [9] based on depth-first traversal) was too limiting. GUI Ripping is based on dynamic analysis, and hence suffers from incompleteness inherent in such analyses, leading to partial EFGs. We cannot use partial EFGs for script repair because many times, these scripts contain events absent from partial EFGs. Our previous work did not suffer from this limitation because our model-based test cases themselves were obtained from these partial models [1] – there was no possibility of them containing events absent from the models. Third, we had assumed perfect knowledge of the GUI's changes. In practice, such knowledge is not always available, making our previous "perfect repairing transformations" unusable. Fourth, we had completely ignored the test oracles – assertions and checkpoints, that form a vital part of test scripts as they determine whether a test case passed or failed. Fifth, we had developed only 4 transformations to address a limited number of GUI changes. Finally, our prior tools worked only at model level, not the script level. GUI test scripts are coded using very low-level method-like calls, e.g., *Window("PMS").Button("Add").Click* to click on the "Add" button in the window entitled "PMS." We lacked mechanisms to automatically abstract such scripts to the model level required by our transformations and synthesize low-level repaired scripts from the models.

In this paper, we present *ScrIpT repAireR* (*SITAR*), a technique to *repair* unusable low-level *test scripts*. *SITAR* works by elevating the level of abstraction of unusable test scripts—from script language level to model level—applies model-based repairing transformations to obtain model-level usable test cases, and then synthesizes new low-level usable scripts. To perform the repair, it relies on imperfect and inaccurate EFG models of the GUIs, as well as human input.

More precisely *SITAR* takes as input an application under test (AUT) A_0 with its incomplete and imprecise automatically reverse engineered EFG G_0, the AUT's modified version A_1 with its incomplete and imprecise EFG G_1, and test suite TS_0 (including assertions/checkpoints of graphical outputs) created for A_0. *SITAR* identifies $ts_0 \subseteq TS_0$ of test cases that are no longer usable for A_1, and hence are candidates for repair. *SITAR* uses repairing transformations and human input (as annotations) to repair test cases in ts_0 that can execute and satisfy assertions on A_1.

Our exemplar system for experimentation with *SITAR* is QTP [7] that provides functional and regression test automation. Our experiment involved 370 QTP test scripts containing a total of 13,043 events and 1,224 checkpoints obtained by more than 200 testers on 3 software applications. We used 2 versions of each application. The changes to the software made all test scripts unusable for 2 applications. We show that up to 89% of the scripts were repaired by *SITAR*. The resulting line coverage of the scripts for one application went up from 0% to 68.3%. We show that annotations help to reduce human involvement dramatically after 20% test scripts have been repaired.

Our study also serves to illustrate 3 important points: (1) manually created test cases often have considerably more coverage over the applications than automated reverse engineering techniques (Table 10), (2) *SITAR* successfully repairs test cases (tables 11 and 12) without breaking the business logic of the original test cases (Table 13), and (3) by accumulating human input as assertions in the model, *SITAR* achieves better automation over the lifetime of the overall repair process (figures 10 and 11).

In designing *SITAR*, we make the following research contributions.

- Mechanisms to use an incomplete/inaccurate automatically reverse-engineered EFG together with human input to repair complex test scripts while retaining the scripts' ability to test the AUT's business logic.
- Mechanisms to handle repairs without perfect knowledge of the GUI and its changes. Based on the existing EFG approximation, the repair suggestions are automatically given to human testers who then select the most applicable repair transformation.
- Mechanisms to incorporate and accumulate human input into the overall process which accelerates the process and produces a more accurate EFG model.
- Mechanisms to repair object references in checkpoints. Most of the checkpoints remain valid in the repaired scripts, which shows the business logic encoded in the original test scripts persists after the repairs.
- New annotations in the EFG to facilitate repairs. Specifically, we introduce a new `dominates` an-

notation on edges.
- Mapping between code and model level to realize translation from code to model and vice versa.
- Output an annotated EFG model of the AUT that is more accurate than the one obtained automatically via reverse engineering. These annotations take the form of events/edges confirmed/added by a human tester. The accurate EFG accelerates the repairing process as well as has the potential to improve test generation and repair for later versions of the AUT.

2 Related Work

Our work on *SITAR* has roots in two previously reported techniques [1], [5] that laid the foundations for GUI test repair and script maintenance. The idea of test repair was originally developed by Memon et al. [10]. The approach reported therein fully automatically classifies usable and unusable test cases. The unusable test cases are fully automatically further classified as repairable or not. Repairing transformations are developed to fully automatically repair the test cases. Because of their focus on full automation, the work made a number of assumptions about the EFG models, test cases, and modifications. The EFGs were assumed to be complete and precise; the test cases were assumed to be composed of high-level events only from the set in the EFGs, without checkpoints or assertions; and all modifications were assumed known. These assumptions do not hold for test scripts that are created manually. The scripted test cases may contain certain events absent from the EFGs, which are prone to incompleteness. Test scripts have checkpoints useful for functionality verification. And GUI modifications from one version of a software application to its next version are rarely fully documented precisely. Our current work, *SITAR*, sacrifices full automation but works in a more realistic setting by eliminating these assumptions.

The work on test script repair was done by Grechanik et al. [5], in which they automatically identify changes between GUI objects and locate test script statements that reference the modified GUI objects [5]. Their tool pops up warnings that enable testers to fix errors in test scripts manually. However, the repairs are done manually.

Several researchers build upon the work on GUI test repair. We ourselves have extended the overall repair process to one driven by combinatorial coverage and genetic algorithms to yield GUI specifications [11]. Daniel at al. [3] propose a white-box approach where GUI changes, specifically GUI refactorings, are recorded and later used to repair the test cases. They envision the use of a smart IDE that will record these changes precisely as they happen and will use them to change the GUI code and to repair test cases. Fu et al. [12] proposed a type-inference tool, named TIGOR, for GUI test scripts, which can work as an assistant of testers to determine type errors of GUI scripts by static analysis. The big difference between our work and theirs is that we do not purely focus on type of objects, but also various kinds of errors like deletion, addition or modification of widgets to fix various kinds of errors in regression GUI testing.

Zhang et al. [13] develop a technique to automatically repair broken workflows for evolving GUI applications. They focus on replacement of invalid user interface actions in the original user interface workflows by dynamic profiling, static analysis, and random testing. They evaluate their technique on 15 workflows from 5 applications.

Choudhary et al. propose WATER to suggest repairs for broken test scripts of web applications [14]. Their technique is based on differential testing; the behavior of test cases is used to suggest the location of the repairs. Leotta et al. [15] present an industrial case study of web test suite repair in which they compare the maintainability of selenium WebDriver test suites by employing different locators, specifically ID or XPath. Such an approach reflects the lack of comprehensive approach for test repair in industrial practice. Alshahwan et al. [16] have presented an approach to repair user session data that has become obsolete due to modification of web pages. Using white-box techniques, they map and locate changes between the two versions and take repairing operations based on the changes detected.

Mirzaaghaei et al. propose TestCareAssistant to automatically repair tests because of changes in method declarations [17]. Their technique combines data-flow analysis with program diffs. In later work, they coin the term "test case adaptation" to support test suite evolution [18] to repair test cases that do not compile due to changes in the code and to generate new test cases for evolved software. Mirzaaghaei has also observed, and experimentally verified, that software developers follow common patterns to identify changes and adapt test cases [19]. Mirzaaghaei envisions the development of an automated approach that generalizes these patterns into a set of test adaptation patterns that can automatically evolve existing test cases and generate new ones.

Daniel et al. develop ReAssert [20] for test case maintenance for unit tests. ReAssert maintains broken unit tests by changing assertions and updating obsolete literal values by analyzing the run time environment of running the failing test case. In later work, they "improve ReAssert for programs with more complex control flow or operation by using symbolic execution"[2].

Because of the difficulties that surround test repair, Evans et al. propose a technique named differential testing to alleviate the test repair problem and to reveal behavior changes during software evolution [21]. Differential testing creates test suites for the original

as well as the modified program and contrasts both versions with the two suites. Other research that may also aid in test repair is automated detection of API refactorings in libraries [22] that automatically detects refactoring between two versions of libraries based on syntax analysis. Similarly, work on repairing defects in the program, not test cases is also relevant [23].

Our work is unique in the sense that we also repair the expected output (i.e., the test oracle) part of the test cases. We leverage our earlier work [24], [25] on test oracles to form the basis for the repair. Also related is the work by Xie et al. who augment automatically generated unit-test suites with regression oracle checking [26]. They show that test oracles from previous version of software can be valuable in detecting faults after software evolution. They collect object states by executing test cases on the original software, and use this information as oracles for the augmented unit-test suites for the modified software.

Yang et al. [27] study the difference between random testing and functional testing in GUI applications. They show the importance of human knowledge in testing large applications via creating more meaningful functional tests. Tests generated by manual functional testing bring more challenges to test repair, and at the same time, make our technique that focuses on functional tests more valuable.

Finally, Pinto et al. present an excellent treatment of the realities surrounding test-suite evolution and maintenence [28]. They discuss various realistic use cases in which test cases are added, removed, and refactored in practice. They also point out that, different from previous cases, test repair is more complex and hard-to-automate and existing test-repair techniques focusing on assertions may be inapplicable in practice. This motivates us to repair real test scripts which involves different types of changes and requires domain knowledge to repair. We enhance the widely used EFG model by storing human actions as new nodes/edges/labels in the model to accelerate the semi-automatic repair process.

3 Overview

We provide an overview of *SITAR* via a simple homegrown example application called *Project Management System* (PMS), used to create and manage projects; project members may be added, edited and removed. We show 3 dialogs of *PMS version 1.0* in Figure 1. The (leftmost) main window, entitled *PMS*, has two views: (1) a default start screen (not shown) and (2) "General Information of Project" screen (shown) that requires the event sequence ⟨*File*, *Open Project*⟩ and selection of a project from a dialog. In this view, a user may add a new project member or remove an existing one. Clicking on *Add Member* opens the (center) dialog. *PMS* also contains a pull-down menu used to launch other dialogs. For example, ⟨*File*, *Create Project*⟩ opens the (rightmost) *Create Project* dialog.

The GUI enforces certain constraints. First, on the main window, the button *Remove Member* is enabled only when a project member (row) is selected. Second, the *Finish Member* button in the *Add Member* dialog remains disabled (Figure 1) until valid values are entered for the *Name* and *E-mail* text fields.

Our test scripts for version 1.0 are shown in Figure 2. Test script TS_1 creates a new project, inputs values for *Title* and *Description*, and clicks on the *Finish Project* button. TS_2 creates a new project member, inputs *Name*, selects *Character* and inputs *E-mail*, finally clicking on the *Finish Member* button. TS_3 opens a project, removes the 2nd project member, and selects the 1st member for further actions. TS_4 opens a project, checks (cp1) whether it has 4 members, removes the first member, adds information for a new member, but before clicking on *Finish Member*, checks (cp2) if 3 members are displayed; and confirms (cp3) that 4 members are displayed after the button is clicked. TS_5 creates a new project entitled "newProject," and checks (cp1) whether the title was reflected correctly on the GUI; it then adds a new member, checks (cp2) whether the *Character* was reflected correctly on the GUI, and checks (cp3) whether the number of members is 1.

For ease of presentation, we have shown simple (yet illustrative) test scripts. In practice, test scripts may look more complicated. Instead of referencing widgets by their names, widgets can be referenced by their IDs or sequence numbers. For example, two button with exactly the same label in the same window will need to be distinguished by their sequence numbers. In our examples, we avoid such complications; during our mapping and repairing procedure, without loss of generality, we address each widget only by its label. We later generalize to addressing a widget by its *signature*.

Our version 2.0 of *PMS* has two changes. First, the *Remove Member* button now pops up a new confirmation dialog (Figure 3); clicking on *Yes* is required to actually delete the project member. Second, the data field *Character* is renamed to *Role*. This changes the column header in the leftmost window of Figure 1 and the text-label in the center dialog. It also changes the way the *Character* field is accessed in the test scripts. For example, in TS_2, Line 4 refers to the *Field* as "Character," which is no longer correct; it needs to be "Role."

Our regression testing scenario is now ready. If we execute all 5 test scripts on PMS version 2.0, TS_1 will execute correctly (provided that nothing else is broken in version 2.0), TS_2 will stop prematurely because the *Character* field no longer exists. TS_3 will hang because it does not know how to handle the unexpected confirmation dialog. TS_4 will have problems because of both changes, and TS_5 accesses the *Character* field,

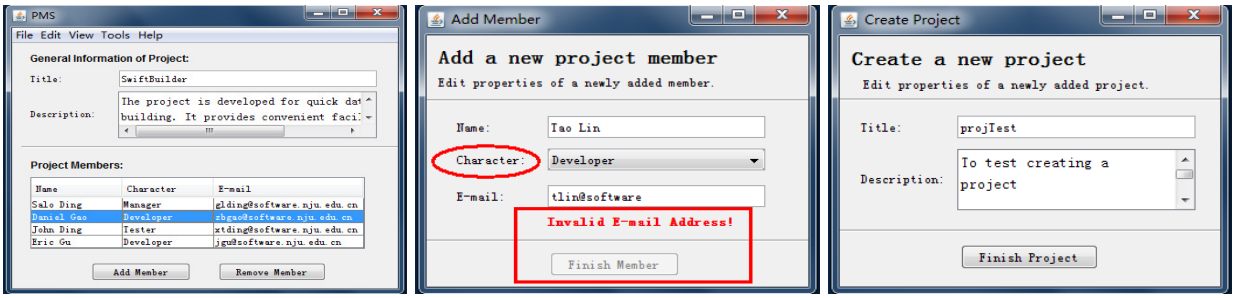


Fig. 1. 3 Windows in Version 1.0.

TS_1	1	Window("PMS").Menu("File").Menu("Create Project").Select
	2	Window("PMS").Dialog("Create Project").Field("Title").Set "projTest"
	3	Window("PMS").Dialog("Create Project").Field("Description").Set "To test creating a project"
	4	Window("PMS").Dialog("Create Project").Button("Finish Project").Click
TS_2	1	Window("PMS").Menu("File").Menu("Open Project").Select
	2	Window("PMS").Button("Add Member").Click
	3	Window("PMS").Dialog("Add Member").Field("Name").Set "Tao Lin"
	4	Window("PMS").Dialog("Add Member").Field("Character").Set "Developer"
	5	Window("PMS").Dialog("Add Member").Field("E-mail").Set "tlin@software.nju.edu.cn"
	6	Window("PMS").Dialog("Add Member").Button("Finish Member").Click
TS_3	1	Window("PMS").Menu("File").Menu("Open Project").Select
	2	Window("PMS").Table("Project Members").SelectRow #2
	3	Window("PMS").Button("Remove Member").Click
	4	Window("PMS").Table("Project Members").SelectRow #1
TS_4	1	Window("PMS").Menu("File").Menu("Open Project").Select
	cp1	Window("PMS").Table("Project Members").CheckProperty("size", 4)
	2	Window("PMS").Table("Project Members").SelectRow #1
	3	Window("PMS").Button("Remove Member").Click
	4	Window("PMS").Table("Project Members").SelectRow #1
	5	Window("PMS").Button("Add Member").Click
	6	Window("PMS").Dialog("Add Member").Field("Character").Set "Tester"
	7	Window("PMS").Dialog("Add Member").Field("Name").Set "John Ding"
	8	Window("PMS").Dialog("Add Member").Field("E-mail").Set "johnding@cs.umd.edu"
	cp2	Window("PMS").Table("Project Members").CheckProperty("size", 3)
	9	Window("PMS").Dialog("Add Member").Button("Finish Member").Click
	cp3	Window("PMS").Table("Project Members").CheckProperty("size", 4)
TS_5	1	Window("PMS").Menu("File").Menu("Create Project").Select
	2	Window("PMS").Dialog("Create Project").Field("Title").Set "newProject"
	3	Window("PMS").Dialog("Create Project").Button("Finish Project").Click
	cp1	Window("PMS").Field("Name").CheckPropery("text", "newProject")
	4	Window("PMS").Button("Add Member").Click
	5	Window("PMS").Dialog("Add Member").Field("Name").Set "Eric Gu"
	6	Window("PMS").Dialog("Add Member").Field("Character").Set "Developer"
	7	Window("PMS").Dialog("Add Member").Field("E-mail").Set "ericgu@cs.umd.edu"
	cp2	Window("PMS").Dialog("Add Member").Field("Character").CheckPropery("value", "Developer")
	8	Window("PMS").Dialog("Add Member").Button("Finish Member").Click
	cp3	Window("PMS").Table("Project Members").CheckProperty("size", 1)

Fig. 2. Five Test Scripts for Version 1.0.

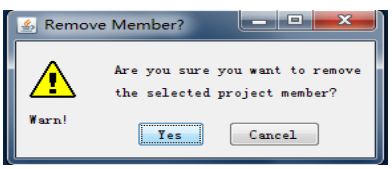


Fig. 3. New Dialog Opens after Clicking *Remove Member*

and hence will fail to execute.

Prior to the development of *SITAR*, the only way to recognize that these low level test scripts will not run is by (1) manual examination of their code, or (2) execution followed by manual examination of the result logs. Needless to say, both are expensive human-intensive processes. It is worth mentioning that techniques such as type-error checking [12] would not help in this case as there are no type modifications in our example.

Our tool *SITAR* **starts** by obtaining, fully automatically, an *approximation* of a model called an event-flow graph (EFG) for version 2.0 (Figure 4). It uses a process called *GUI Ripping* [9], which we very briefly describe now. During ripping, the application is executed; its main window is opened, the list of all events is extracted and stored in a queue; the events are executed by the ripper one-by-one from this queue. Buttons are clicked, radio buttons are selected, etc., thereby mimicking a human user. If a text field is encountered, then a manually predetermined value is entered; if none is provided, one is generated randomly. As events execute, they open new windows. This causes the current window to change. The previous queue is pushed onto a stack, to be used later once the new current window is closed.

One output of the GUI ripper is a model called the event-flow graph (EFG). The EFG is a directed graph model of the GUI. Its vertices represent events; edges represent either the `may-follow` or `dominates` relationship. A `may-follow` edge from n_x to n_y means that the event represented by n_y *may immediately* follow the event represented by n_x *along some execution paths*. For example, because an end user may execute *Yes* in the confirmation dialog of Figure 3, immediately after clicking on *Remove Member* of the main *PMS* window, there is an edge from *Remove Member* to *Yes* in this application's EFG. Similarly, there is an edge from *Remove Member* to *Cancel*. A `dominates` edge, introduced for the first time in this research, forces the occurrence of a preceeding event – we give an example later in this section and formally define it in the next section.

The EFG obtained by the ripping process is only an approximation because the dynamic approach of the ripper has limitations that stem from the paths it follows, and the data it provides during ripping. For example, in its default fully automatic mode, it cannot generate a valid e-mail; *Finish Member* remains disabled. Consequently, the EFG model does not contain any of *Finish Member*'s incident edges. Repairing in the face of an imperfect EFG requires manual intervention and annotations that we introduce in this paper.

The second output of ripping is a *signature* for each widget, i.e., a unique way to address and access each widget. The signature typically consists of the

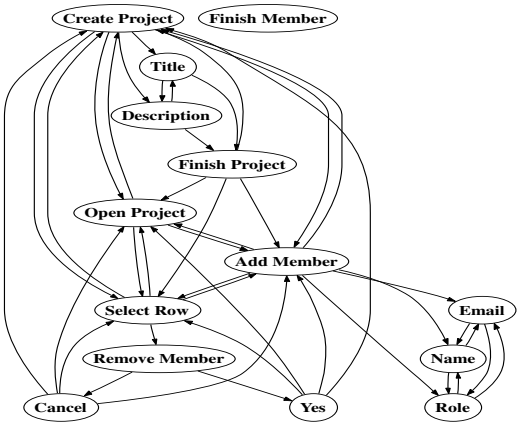


Fig. 4. Approximate Ripped EFG for Version 2.0

container (Window or Dialog), the class that was instantiated to create the widget (e.g., Button, Field, Menu, Table), and some properties of the widget, such as *location*, *height*, *width*, and the default action used to execute the widget. For example, the signature for the *Create Project* menu item contains information such as its class (*Menu*), and the containing *PMS* window and the *File* menu. The signature enables *SITAR* to map QTP statements to logical event names in the EFGs; and ultimately test scripts to sequences of logical event names. It is these logical sequences that are repaired. A part of the mapping is shown in Table 1. We note that because *Character* is not in version 2.0 of *PMS*, it has no equivalent logical name and is marked as NULL. An reasonable alternative to creating logical names for each event is to generate the corresponding QTP statement to access the widget. However, a logical name provides us with a concise way to address the event in our models.

TABLE 1
Partial Mapping QTP Statements to Logical Names

QTP Access	Logical Name
Window("PMS").Menu("File").Menu("Create Project")	Create Project
Window("PMS").Dialog("Create Project").Field("Title")	Title
Window("PMS").Dialog("Create Project").Field("Description")	Description
Window("PMS").Dialog("Create Project").Button("Finish Project")	Finish Project
Window("PMS").Menu("File").Menu("Open Project")	Open Project
Window("PMS").Button("Add Member")	Add Member
Window("PMS").Button("Remove Member")	Remove Member
Window("PMS").Dialog("Add Member").Field("Name")	Name
Window("PMS").Dialog("Add Member").Field("Character")	NULL
Window("PMS").Dialog("Add Member").Field("E-mail")	Email
Window("PMS").Dialog("Add Member").Button("Finish Member")	Finish Member

Our repairing process starts by *checking* all test scripts, i.e., ensuring that event sequences (from the original scripts) map to paths in version 2.0's EFG. As expected, TS_1 checks out fine; there is a valid path ⟨Create Project, Title, Description, Finish Project⟩ in the EFG. All other test scripts fail.

TS_2 maps to the sequence ⟨Open Project, Add Member, Name, NULL, Email, Finish Member⟩. NULL needs to be resolved. *SITAR* uses a shortest-path algorithm to determine possible substitutes for NULL. Because there is a direct edge from *Name* to *Email*, one possible choice is that there is no longer a need for an intermediate event. *SITAR* also looks for the next shortest path because the original test script contained an intermediate event, and hence an alternative event might be available. It determines that *Role* may be such an event. A human tester is notified of the problem with the two possible solutions to replace NULL (1) no event, i.e., delete NULL or (2) *Role*; the tester manually sets the original *Character* event as *Role*. This information is stored in the mapping (Table 1) to be reused for subsequent repairs. TS_2 is still unusable because the EFG model does not contain the edge (*Email*, *Finish Member*). The human tester is asked to confirm the existence of this edge in version 2.0. The EFG is updated (Figure 5) and TS_2 is repaired as shown in Figure 6.

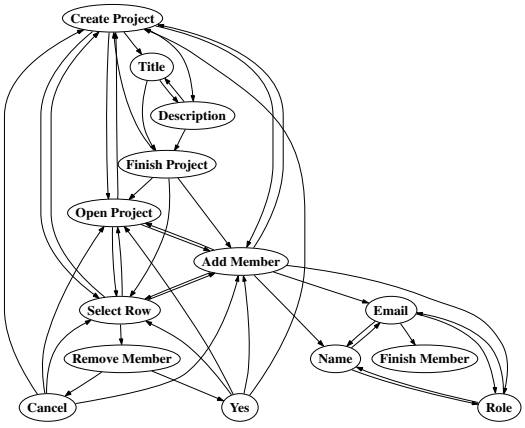


Fig. 5. EFG of version 2.0 after Repairing TS_2

TS_3, ⟨Open Project, Select Row, Remove Member, Select Row⟩, has several problems. First, the edge *(Remove Member, Select Row)* no longer exists. Second, the actual removal of a member is now done when the user clicks on *Yes*; in some sense, the old *Remove Member* is now equivalent to the new *Yes*. These two problems are handled via two mechanisms. The first requires a tester to map the original *Remove Member* to *Yes*. The second problem is handled by manually annotating the edge *(Remove Member, Yes)* in version 2.0's EFG as a `dominates` edge (Figure 7). This means that *Remove Member must always* be executed immediately before *Yes* for all executions. Supplied with this new information, *SITAR* uses a 2-step process to repair TS_3. It first replaces *Remove Member* with *Yes*, obtaining ⟨Open Project, Select Row, **Yes**, Select Row⟩; it then uses the `dominates` edge to add the new *Remove Member* before *Yes* to the repaired test script, yielding ⟨Open Project, Select Row, Remove Member, Yes, Select Row⟩. The final repaired script is shown in Figure 6.

The information supplied thus far by the tester,

TS_2	1	Window("PMS").Menu("File").Menu("Open Project").Select
	2	Window("PMS").Button("Add Member").Click
	3	Window("PMS").Dialog("Add Member").Field("Name").Set "Tao Lin"
	4	Window("PMS").Dialog("Add Member").Field("Role").Set "Developer"
	5	Window("PMS").Dialog("Add Member").Field("E-mail").Set "tlin@software.nju.edu.cn"
	6	Window("PMS").Dialog("Add Member").Button("Finish Member").Click
TS_3	1	Window("PMS").Menu("File").Menu("Open Project").Select
	2	Window("PMS").Table("Project Members").SelectRow #2
	3	Window("PMS").Button("Remove Member").Click
	4	Window("PMS").Dialog("Remove Member?").Button("Yes").Click
	5	Window("PMS").Table("Project Members").SelectRow #1
TS_4	1	Window("PMS").Menu("File").Menu("Open Project").Select
	cp1	Window("PMS").Table("Project Members").CheckProperty("size", 4)
	2	Window("PMS").Table("Project Members").SelectRow #1
	3	Window("PMS").Button("Remove Member").Click
	4	Window("PMS").Dialog("Remove Member?").Button("Yes").Click
	5	Window("PMS").Table("Project Members").SelectRow #1
	6	Window("PMS").Button("Add Member").Click
	7	Window("PMS").Dialog("Add Member").Field("Role").Set "Tester"
	8	Window("PMS").Dialog("Add Member").Field("Name").Set "John Ding"
	9	Window("PMS").Dialog("Add Member").Field("E-mail").Set "johnding@cs.umd.edu"
	cp2	Window("PMS").Table("Project Members").CheckProperty("size", 3)
	10	Window("PMS").Dialog("Add Member").Button("Finish Member").Click
	cp3	Window("PMS").Table("Project Members").CheckProperty("size", 4)
TS_5	1	Window("PMS").Menu("File").Menu("Create Project").Select
	2	Window("PMS").Dialog("Create Project").Field("Title").Set "newProject"
	3	Window("PMS").Dialog("Create Project").Button("Finish Project").Click
	cp1	Window("PMS").Field("Name").CheckPropery("text", "newProject")
	4	Window("PMS").Button("Add Member").Click
	5	Window("PMS").Dialog("Add Member").Field("Name").Set "Eric Gu"
	6	Window("PMS").Dialog("Add Member").Field("Role").Set "Developer"
	7	Window("PMS").Dialog("Add Member").Field("E-mail").Set "ericgu@cs.umd.edu"
	cp2	Window("PMS").Dialog("Add Member").Field("Role").CheckPropery("value", "Developer")
	8	Window("PMS").Dialog("Add Member").Button("Finish Member").Click
	cp3	Window("PMS").Table("Project Members").CheckProperty("size", 1)

Fig. 6. Repaired Test Scripts for Version 2.0.

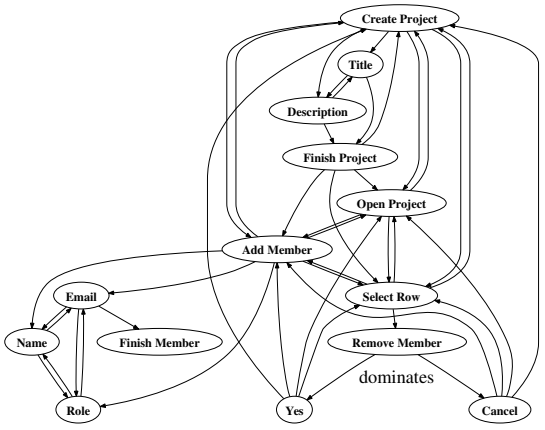


Fig. 7. Annotated EFG of Version 2.0 After Repairing TS_3

cached in the mapping table and annotated EFG is now sufficient to fully automatically repair TS_4 and TS_5. TS_4 uses *Remove Member* and accesses the *Character* field. The modified mapping and `dominates` relationship is used to repair TS_4; the new test script is seen in Figure 6. Note that the checkpoints (cp1, cp2, cp3) in the script remain unchanged.

TS_5 uses the *Character* field in an input statement and a checkpoint. *SITAR* already knows how to repair the script using the earlier modified mapping. For the checkpoint (cp2), only the GUI object is updated; the *expected value* "Developer" remains unchanged. The final test script is in Figure 6.

4 Models & Algorithms

A logical perspective of the components of *SITAR* is shown in Figure 8. *SITAR* deals with three distinct

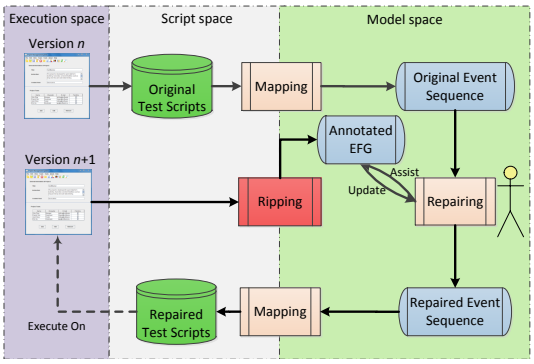


Fig. 8. Logical View of *SITAR*'s Components

spaces (levels of abstraction): (1) execution, (2) script, and (3) model. *Ripping* raises the level from execution to model. And *mapping* first raises the level of abstraction from script (for version n) to model and then, once the scripts have been repaired, from the model back to script (version n+1). We now discuss the designs of these components and the models and algorithms they employ to realize *SITAR*.

4.1 Ripping

Ripping aims to raise the level of abstraction of the implemented GUI (version n+1) to an abstract model that allows repairs. During ripping, the GUI application is executed automatically; the application's windows are opened in a depth-first manner. The GUI Ripper extracts all the widgets and their properties from the GUI. During the reverse engineering process, in addition to widget properties, additional key attributes of each widget are recovered (e.g., whether it is enabled, it opens a modal/modeless[1] window, it opens a menu, it closes a window, it is a button, it is an editable text-field). These attributes are used to construct the EFG.

A *modal window* is a GUI window that, once invoked, monopolizes the GUI interaction, restricting the focus of the user to a specific range of events within the window, until the window is explicitly or implicitly terminated. Other windows in the GUI that do not restrict the user's focus are called *modeless windows*; they merely expand the set of GUI events available to the user. We define the term *modal dialog* as a group of windows consisting of a modal window X and a set of modeless windows that have been invoked, either directly or indirectly from X. The modal dialog remains in place until X is explicitly terminated. By definition, a GUI user cannot interleave events of one modal dialog with events of other modal dialogs; the user must either explicitly terminate the currently active modal dialog or invoke

1. Standard GUI terminology; see detailed explanations at msdn.microsoft.com/library/en-us/vbcon/html/vbtskdisplayingmodelessform.asp and documents.wolfram.com/v4/AddOns/JLink/1.2.7.3.html.

another modal dialog to execute events in different dialogs. At all times during interaction with the GUI, the user interacts with events within the modal dialog.

Ripping outputs an EFG model that represents all possible event interactions in the GUI. Modeled as a directed graph, each of its vertices represents an event (e.g., click-on-Edit, click-on-Paste)[2] and each edge represents a `may-follow` relationship between two events. A `may-follow` edge from vertex x to vertex y shows that an event y *may* be performed immediately after event x (i.e., y `may-follow` x).

The construction of the EFG is based on the identification of modal dialogs, and hence the identification of modal and modeless windows. A classification of GUI events is used to identify modal and modeless windows. For example, *Restricted-focus events* open *modal windows*. If **v** is a restricted-focus event, then only the events of the invoked modal dialog are available. *Unrestricted-focus events* open *modeless windows*. If **v** is an unrestricted-focus event, then the available events are all top-level events of the invoked modal dialog available as well as all events of the invoking modal dialog. *Termination events* close modal windows. If **v** is a termination event, then `may-follow`(**v**) consists of all the top-level events of the invoking modal dialog.

In our work to date, all EFG edges are initially `may-follow` after the ripping stage. In this paper, we introduce a new type of edge, called a `dominates` edge, to enhance the EFG. A `dominates` edge from vertex x to vertex y shows that event y must be preceded by event x (i.e., x `dominates` y); in this case, the vertex representing y has a single incoming edge (from x). Some edges may be manually updated to `dominates` edges by a tester in the process of repair.

Definition: An *annotated EFG* (or simply EFG[3]) is a 4-tuple $\langle V, I, E_{may}, E_{dom} \rangle$, where:

1. V is a set of vertices representing all events of objects.
2. $I \subseteq V$ is a set of initial vertices. The events in I are available to the user when the application is first invoked.
3. $E_{may} \subseteq V \times V$ is a set of `may-follows` edges between vertices. $(v_i, v_j) \in E_{may}$ iff v_j may be executed immediately after v_i.
4. $E_{dom} \subseteq V \times V$ is a set of `dominates` edges between vertices. $(v_i, v_j) \in E_{dom}$ iff v_j has a single incoming edge from v_i; and v_j must be preceded by v_i in all executions. And that $E_{may} \cap E_{dom} = \phi$.

From our discussion so far, it is evident that an EFG is stateless. It is merely a flow graph that encodes the different sequences of events that may be executed on the GUI.

2. In subsequent discussion, for brevity, the names of events will be abbreviated, e.g., Edit and Paste.

3. Note that this is different from our previous work [6].

4.2 Mapping

The mapping between low level GUI objects/widgets and logical events used in the model raises the level of low-level scripts to that of the model so that sequences may be mapped to EFG paths and subsequently repaired.

During ripping, we know the location of each widget, its *container* (e.g., Dialog and Window), and the class that was used to create it. These attributes are also used by QTP to address each widget. As discussed previously, we explicitly store the mapping between the QTP names and our logical event names. We parse each script statement and extract its object type and title. For example, the statement *JavaWindow("PMS").JavaButton("Add").click* is parsed to obtain two object types: *JavaWindow* and *JavaButton*, and their corresponding title: *PMS* and *Add*. These are then searched in the mapping from the ripper and represented using their logical form. If a match is not found that a NULL entry is created.

More formally, our mapping is a 2-column lookup table: the first column is the addressing mechanism used by the scripting system for a widget; the second column is the logical model-level string label that we assign to the widget. To create the mapping, we start with an empty mapping, *Map*, adding entries to it as we iteratively examine each script using the following algorithm, which takes 2 inputs (1) a test script *TS* consisting of a sequence of script statements $\langle s_1, s_2, ..., s_n \rangle$ created on the original version of the AUT A_0, and (2) G_1, the EFG of the modified AUT A_1. The output is the modified *Map*. The pseudo-code of our technique is shown in Figure 9. As shown in line 5-8, for each script statements s_i, do:

(1) If s_i already in *Map*, then skip to next script statement; else
(2) if $\exists e \in G_1$ such that e's GUI properties (determined from its signature described earlier) match with GUI properties from the GUI element in statement s_i, then add new entry (s_i, e) to *Map*; else
(3) add new entry (s_i, *NULL*) to *Map*.

The second of the above 3 alternatives is desirable, and hence, we call it a *successful* mapping.

As mentioned earlier, the only feasible way to obtain the EFG G_1 is by using automatic reverse engineering techniques, which may miss a considerable portion of windows/widgets of A_1. We show later in our empirical study on three open source Java applications that only 12-65% of the events (widgets) are obtained by our GUI Ripper and successfully mapped. The rest are all mapped to *NULL*. We also show that at least one unsuccessfully mapped script statement occurs in most of the scripts created by test engineers. This result reinforces our original intuition that we cannot use our previous fully automatic technqiues for test script repair. Next we describe a more realistic

human-assisted repair approach.

4.3 Repairing

By this point, after the mapping has been obtained, each script statement is either mapped to a high-level event in our EFG model or *NULL*. Similarly, each GUI widget referenced in checkpoints in the scripts is also mapped to a model-level widget (equivalently an event because of how we synonymously model events and widgets) or *NULL*. *SITAR* is ready to start repairing.

The ideal case for the repairer is one in which the sequence has a non-NULL entry in the mapping for each of its events and there is an edge in the EFG for each adjacent event pair. Such a sequence is considered to be valid and the mapping is used to synthesize a low-level script.

A mapped script that has at least one *NULL* (event/checkpoint) must be repaired. Moreover, even if there are no *NULL* events in the script, an invalid *flow* of execution of events still warrants a repair. That is, the sequence of events performed by the script needs to be allowed by the GUI's workflow. More formally, we formulate the problem of test repair as follows. Given an EFG G_1 and a mapped script, represented as a sequence of events $\langle e_1, e_2, ..., e_n \rangle$ and checkpoint $\langle c_1, c_2, ..., c_m \rangle$ where $e_i \in G_1 \cup \{NULL\}$ and $c_i \in G_1 \cup \{NULL\}$, the script needs repair if one of the following conditions is satisfied:

- Case 1, *Missing event:* At least one of e_i or c_j in the sequence is *NULL* where $1 \leq i \leq n$ or $1 \leq j \leq m$;
- Case 2, *Missing edge:* At least one pair (e_i, e_{i+1}), $1 \leq i \leq n-1$, of adjacent events in the sequence is not a valid edge in E_{G_1}.

The output of the repair is a sequence of events $\langle e'_1, e'_2, \ldots, e'_n \rangle$ and repaired check points $\langle c'_1, c'_2, ..., c'_m \rangle$ that do not contain any missing widgets or missing edges.

Repairing starts with 4 inputs: (1) a set of logical event sequences, each corresponding to a test script, (2) the EFG model in which all the edges are marked as `may-follow`, (3) the initial mapping, and (4) an empty table of *approved* paths between each pair of events. These 4 inputs are also seen in the pseudo-code of the *repair* algorithm shown in Figure 9: input test suite TS_0 created on A_0, automatically obtained EFG G_1 for application A_1, initial mapping *mappingTable* and an *approvedTable*. We use this pseudo-code to explain the two cases of the repair process.

• *Case 1: Repair missing event when there is a NULL event in the sequence (lines 10-13).* In this case, the QTP statement could not be mapped to a logical event in the EFG. This may happen for a number of reasons. First, the event may no longer exist in the GUI of A_1. In this case the tester needs to delete the event from the script. And a may-follow relationship will be added from the event prior to current event to

```
1   Initialization of global variables:
        G1 = (V, I, E), mappingTable = ∅, approvedTable = ∅
2   Output: TS1, updated G1, mappingTable and approvedTable

3   Procedure repair(TestScript TS0):
4       TS1 ← ∅
5       For all test cases TC= ⟨s1, s2, ··· , sn⟩ ∈ TS0
6           EvtSet ← ∅; CptSet ← ∅
7           For all statements or checkpoints si ∈ TC
8               Map si to event ei or checkpoint ci
9               EvtSet.add(ei); CptSet.add(ci)
10          For all events ei and checkpoints ci ∈ EvtSet ∪ CptSet
11              If ei = NULL or ci = NULL
12                  repairedSeg ← repairEventAndUpdateModel(si, ei−1, ei+1)
13                  (EvtSet ∪ CptSet).replace(⟨ei−1, ei, ei+1⟩, repairedSeg)
14          For all ⟨ei, ei+1⟩ where ei and ei+1 ∈ EvtSet
15              If ⟨ei, ei+1⟩ ∉ E
16                  repairedSeg ← repairEdgeAndUpdateModel(ei, ei+1)
17                  (EvtSet ∪ CptSet).replace(⟨ei, ei+1⟩, repairedSeg)
18          TS1.add((EvtSet ∪ CptSet).mapToTestScript())
19      Return TS1

20  Procedure repairEventAndUpdateModel(si, ei−1, ei+1):
21      If confirm script si or remap to ei′:
22          add si or ei′ to V as ϵi
23          add ⟨ei−1, ϵi⟩ and ⟨ϵi, ei+1⟩ to Emay
24          If remap: mappingTable.add(si → e′i)
25          Return ⟨ei−1, ϵi, ei+1⟩
26      If deleteNode:
27          add ⟨ei−1, ei+1⟩ to Emay if not exist
28          Return ⟨ei−1, ei+1⟩
29      Return repairEdgeAndUpdateModel(ei−1, ei+1)

30  Procedure repairEdgeAndUpdateModel(x, y):
31      If approvedTable.lookup(⟨x, y⟩) → ξ
32          Return ⟨x, ξ, y⟩
33      If (a, y) ∈ Edom
34          Return repairEdgeAndUpdateModel(x, a) ∪ y
35      search the shortest path ⟨x, ξi, y⟩ ∈ E
36      tester confirm ⟨x, ξi, y⟩ from suggested paths and
        may make manual modifications to the best suggested paths:
37          approvedTable.add(⟨x, y⟩ → ξi)
38          Return ⟨x, ξi, y⟩
```

Fig. 9. Pseudo-code of Repairer

the event after current event (lines 26-28). Second, the event may still exist in A_1, but may have been missed during automatic creation of G_1. In this case, the tester needs to add the missed event. The new event will be added to the vertex set V and a pair of new edges related to this event will be added to the may-follow edge set E_{may} (lines 21-23). Third, certain attributes of the event's widget may have changed, e.g., new label, widget type, or it may have moved to a different location in the GUI, making an automatic equivalence determination by matching signatures impossible. In this case the tester can remap the missed event to its new counterpart in A_1, and this information will be added to a mapping table which can be referenced by future repairs (line 24). The repairing algorithm uses information, e.g., the sequence it is repairing, to assist the human tester to make the above decisions. It uses the events immediately *before* and *after* NULL to search for possible alternatives via a shortest path algorithm on the EFG. The alternatives are presented to a tester, who may *confirm* one of the alternatives for current and future repairs. In our implementation, we prioritize the alternatives (by the length of the resulting paths) and present only the most important alternative to the tester. Note that deletion and addition of events also impact edges in the EFG. These

modifications are similar to those discussed in Case 2 next.

• *Case 2: Repair missing edge when no EFG edge is found for two consecutive events x and y in the sequence (lines 14-17).* The repairer uses one of several mechanisms to repair this sequence. First, it examines the *approved* paths table declared as *approvedTable* in the pseudocode; if an entry of the form $[(x, y) \longrightarrow \xi]$ is found, then a human tester had, during a previous script repair, declared that the subsequence $\langle x, y \rangle$ may *always* be replaced with $\langle x, \xi, y \rangle$, where ξ is an event sequence. No additional confirmation is needed (lines 31-32). Second, if there is a `dominates` edge (a, y) in the annotated EFG, then a is inserted into the sequence immediately before y and if there is no EFG edge (x, a), then *Case 2* is recursively applied to (x, a). Again, no additional confirmation is needed (lines 33-34). Third, the repairer employs the shortest path algorithm to find a possible path between x and y (line 35). Please note that only the paths from the *approvedTable* or recursively found by tracing back from `dominate` edges will be automatically applied as a repair. The domain knowledge obtained from testers' previous operations works well when there is a corresponding relationship between event sequences from the old and new versions, but cannot fully solve the problem of lacking of context information because our technique is based on stateless models. Whereas the shortest path algorithm may be able to find only paths containing `may-follow` edges, then confirmation is sought from a human tester who may (1) select one of the suggested paths, (2) create a new path in the EFG by adding new events and edges, (3) join x and y with a new direct edge, or (4) declare that y is not reachable from x in this context and hence the test script cannot be repaired. If no path is found, then the tester needs to manually specify a path, adding events, edges, etc., if needed (line 36).

During manual intervention, the tester may also at any time, reclassify `may-follow` edges as `dominates` edges. Or, the tester may create new entries for the *approved* paths table (line 37). Each new entry (i, j) causes the automatic creation of additional new entries. For instance, assume that the tester adds an entry $[(i, j) \longrightarrow \gamma]$. *SITAR* searches for all existing entries of the form $[(b, i) \longrightarrow \delta]$, and, for each found, adds a new entry $[(b, j) \longrightarrow \langle \delta, i, \gamma \rangle]$. Intuitively, this means that if there is a new approved path from i to j, and there was an existing approved path from any other event b to i, then the concatenated path with i in between is automatically considered approved from b to j.

The reader will note from the above discussion that *SITAR* does not unilaterally make repair decisions. All repairs are authorized by the tester. We feel that this is important so that the tester can verify that the modifications do not cause the test scripts from deviating from the business logic that they are testing. That being said, *SITAR* does use algorithms to come up with repair suggestions from which a tester can examine and select. Currently we offer the following 3 options to the tester, depending on the nature of the repair.

(1) *Confirmation:* the user may confirm the correctness of the current event which means no repair is needed but just the models need to be updated; or the user may confirm the path suggested by *SITAR* when repairing an edge.
(2) *Modification:* the user may repair the current event by re-mapping it to another event; or modify the path suggested by *SITAR* when repairing an edge;
(3) *Addition:* the user may manually add a sequence of events to repair a missing edge.

5 Empirical Study

We now empirically study GUI script maintenance issues and evaluate the effectiveness of *SITAR*.[4] Specifically, we address the following research questions.

RQ1: What fraction of the original test scripts become unusable after GUI modifications? What is the nature of GUI modifications that make test scripts unusable?
RQ2: What fraction of the GUI is ripped automatically?
RQ3: How many test scripts (including checkpoints) are repaired by *SITAR*? How well do the repaired test scripts cover the same percentage of code and events as the original?
RQ4: What is the cost of repairing?
RQ5: How effective are the mappings and annotations?
RQ6: What fraction of test scripts are not repaired and why?

Metrics: We employ several metrics for the above questions. For **RQ1**, we count the test scripts that become unusable. We also classify GUI modifications and count the number of impacted test scripts for each modification class. For **RQ2**, we compare the number of events obtained automatically by the Ripper against those that we obtain by careful manual examination of the GUIs. For **RQ3**, we count the number of scripts successfully repaired, and compute their code and event coverage. We manually study the checkpoints that we fail to repair and classify the reasons. For **RQ4**, we measure the cost of repairs. For **RQ5**, we measure the decline in manual cost over the sequence of repairs. We compute the *operation ratio* (the ratio between number of manual operations and lines of code of repaired test scripts) and *time cost per line of code* (measured in seconds) over the entire repair process. For **RQ6**, we count and classify the test scripts that were not repaired.

4. We provide all data used in this study in downloadable form at http://www.cs.umd.edu/users/atif/SITAR-TSE-Data

Study Process: We start by selecting 3 study subjects, each with 2 versions. We then assemble a team of testers to manually create a number of test scripts using QTP version 11.00. A different team of testers then uses *SITAR* to repair the original test scripts. During this process, we compute all the metrics discussed earlier so that we can address each of our RQs.

Study Subjects: We select three applications – Crossword Sage[5] (versions 0.3.3 and 0.3.4), PDFsam[6] (versions 1.2.0 and 2.0.0), and OmegaT[7](versions 1.8.1_07 and 2.0.5_03). Some of their characteristics are shown in Table 2. For brevity, we refer to Crossword Sage versions 0.3.3 and 0.3.4 as **CS1** and **CS2**, respectively; PDFsam versions 1.2.0 and 2.0.0 as **PDF1** and **PDF2**, respectively; and OmegaT versions 1.8.1_07 and 2.0.5_03 as **OT1** and **OT2**, respectively.

TABLE 2
Characteristics of Study Subjects

Study Subject	All Widgets	Modified Widgets			Events	LOC
		Changed	Added	Deleted		
CS1	59				82	1419
CS2	75	5	16	0	102	1587
PDF1	117				147	8372
PDF2	143	2	26	0	176	11795
OT1	337				376	15474
OT2	348	21	11	0	389	15341

Crossword Sage is used to build and solve crossword puzzles. The major change in CS2 is the data format of stored crosswords. Consequently, crosswords created in CS1 cannot be opened in CS2. Another change is that the size of the crossword must be input before its creation. Finally, there are small cosmetic changes to a menu item and a message box, and the *Save* dialog's title is changed from *Open* to *Save*. These changes led to the addition and modification of some GUI widgets (Table 2). None were deleted.

PDFsam is used to split and merge pdf files. From our perspective, the major changes from PDF1 to PDF2 are renaming of the main window's title and the root item of a menu tree. Although these are more or less cosmetic changes, all test scripts will be impacted because the main window must be opened to navigate to other parts of the GUI.

OmegaT is a computer aided translation tool. The major changes from OT1 to OT2 are the renaming of the main window's title and change of labels of 21 menu-items and buttons (Table 2); their functionality remains unchanged.

These 3 subjects are appropriate for our study because we have used them in previous work [1], [29] and understand their functionality well. We also know them, especially CrosswordSage to implement a variety of changes, e.g., functional changes, GUI changes, additions and re-organization of GUI components, thereby making them suitable for test repair. Moreover, they provide us a variety in terms of size (from thousands of lines of code to tens of thousands). Finally, our GUITAR tool works very well with them. The reader will also note that we did not use the most recent versions of the applications because the newest releases have stabilized their GUIs over time, providing us with very little GUI level script repair opportunities.

Original Test Suites[8]: More than 200 undergraduate students in the software engineering major were employed to create the original test scripts. The students were given clear instructions on how to create the test scripts. They were first trained to use QTP for 3 hours, after which they developed a preliminary set of scripts that were examined for quality and discarded. Only after we were confident in their ability to create the scripts did we employ them to create the scripts that we used for our study. We simultaneously employed graduate students, who have Industry experience in software engineering, to divide each software subject into its high level constituent functional units, each of which implements a high level feature of the software. We obtained 5 units for CS1, 7 for PDF, and 5 for OT. These units were used as a guideline to the undergraduate students creating the scripts: within each functional unit, certain features of the application needed to be covered by a number of test cases. The students had the freedom to determine the exact sequences of events per test case. All participants were asked to start the applications in a known start state.

Even with a carefully engineered process of test script creation, we examined all scripts manually and discarded a few that contained no checkpoints; the numbers finally retained were 101, 140, and 129 scripts, for CS1, PDF1, and OT1, respectively. The maximum, minimum, and average LOC of test scripts are seen in Table 3. As is evident, the scripts ranged in size from small to fairly large.

TABLE 3
Size (LOC) of Test Scripts

	Min	Max	Avg
CS1	5	305	38
PDF1	5	74	26
OT1	7	344	29

As an additional sanity check, all the scripts were executed on their respective subject applications (original versions for which they were created) to ensure that they were in fact executable. Table 4 shows the percentage of code and events covered. Even though this study is not designed to evaluate the quality of QTP test scripts, we note that the coverage is quite high. GUI applications typically have significant

5. http://crosswordsage.sourceforge.net/
6. http://www.pdfsam.org/
7. http://www.omegat.org/
8. http://www.cs.umd.edu/∼atif/SITAR-TSE-Data/

fractions of their code for non-GUI operations (e.g., exceptions, communication), which we do not expect to exercise.

TABLE 4
Coverage of Original Suites

Coverage (%)	CS1	PDF1	OT1
Code	93.1	62.1	66.6
Event	76.8	77.6	88.6

We believe that our process of dividing each software application into its functional units, and using these to guide test script creation yielded us a fairly diverse set of tests with a broad coverage of each application. However, because diversity in our test suite may influence our results (e.g., one repair may influence many "identical" test scripts) we demonstrate that our test scripts are diverse in two ways: (1) in terms of uniqueness of event sequences covered, and (2) code coverage. Because of our prior experience with GUI testing [30], we consider the former to be more important than the latter; we provide the latter because of the popular use of code coverage as a measure of test comprehensiveness.

We first determined that we had no duplicate test cases. Further, we determined that none of our test scripts were "contained in" (and hence redundant) any of our other test scripts. More specifically, we say that a test script s_i is contained-in s_j iff it's event sequence is a part of s_j.

Next, because our scripts are sequences of events, we show diversity by computing the *edit distances* between each pair of event sequences, and then obtain the average edit distance and its ratio to average test length. The results are shown in Table 5. The numbers show that for all 3 applications, the average edit distance of event sequences is greater than the average length of test cases - this is possible only when there is diversity of test cases in the test suite, i.e., the results demonstrate a reasonably significant diversity of the test suites for the 3 applications.

TABLE 5
Similarity of Event Sequences of Test Cases

	Avg Dist	Avg Len	Ratio
CS1	44.3	37.8	1.17
PDF1	28.7	26.2	1.10
OT1	36.2	29.1	1.24

Finally, we measure the number of test cases needed to cover all events and event pairs covered by the test suites by a greedy algorithm. To obtain the same event coverage as the test suites, 99, 112 and 118, scripts are required for CS, PDF and OT, respectively. For event pair coverage, 100, 130 and 125, scripts are required for CS, PDF and OT, respectively. These results show there is a reasonable degree of diversity between test cases in the test suites.

We now show diversity in terms of code coverage. Before we start, we first note that in all 3 GUI applications, there is a large portion of code written for data and GUI initialization. Each script is expected to execute this large initialization code, thereby reducing diversity of code covered. Even then, our test scripts varied in terms of the code they covered. Table 6 shows the minimum, maximum and average coverage percentage of each test script per AUT. In addition, we use the *Jaccard similarity coefficient* to measure the similarity between the sets of lines covered in each pair of test cases. The results are shown in Table 7. The average similarity for the 3 AUTs is 0.56, 0.81 and 0.64, respectively, for the 3 applications. A Jaccard index of 0.56-0.81 demonstrates a reasonable degree of diversity in our test scripts. Also, if we take into consideration the frequency of each line hit, the test scripts will be even more diverse.

TABLE 6
Coverage of Test Cases

	Min	Max	Avg
CS1	11.6	76.1	43.7
PDF1	38.1	56.9	48.1
OT1	10.0	37.6	23.9

TABLE 7
Similarity of Code Coverage of Test Cases

	Min	Max	Avg
CS1	0.13	1.00	0.56
PDF1	0.63	1.00	0.81
OT1	0.23	1.00	0.64

Baseline: In order to address **RQ1** and to establish a baseline for our repairing technique, we executed all test scripts without modifications on the new versions of the applications. We counted the number of scripts (Scripts columns in Table 8) that did not complete successfully due to GUI modifications. We also show the number of checkpoints in the unusable scripts (Checkpoints columns in Table 8).

The outcomes for PDF2 and OT2 are quite startling – none of the scripts ran to completion. For PDFsam, this was because the main window's title was changed; and the root of the menu-tree was renamed. All test scripts must first execute an event in the main window; a change to the title broke them all. Additionally, those that accessed the menu must start at the root of the menu; the renaming further deteriorated the already-broken scripts. Similarly, the title of OmegaT's window also changed; this broke all test scripts. Additionally, capitalization of the first letter of 21 menu-items effected 63 scripts. Even CS2 lost 93.1% of its scripts. Of these, 43 scripts failed to open the data files created by CS1 because of format changes; 48 scripts broke because CS2 expects crossword size to be specified but the scripts did not know how to

interact with the new text fields; and 58 scripts were effected by change of the *Save* window's title from *Open* to *Save*. Certain scripts are counted more than once in these reported numbers because they were impacted by more than one change. In their current form, these scripts achieve very low coverage on the new versions of the applications (17.2% code and 9.8% event for CS2; and 0% for others).

Delving deeper into the causes for our test script failures provided us with valuable insights into their possible repair, especially the difficulty of fully automatic repair. For example, even though some GUI modifications such as change of window/widget titles are straightforward, automatically repairing impacted scripts is challenging. A tool will need to automatically determine the mapping between old and new titles, an algorithmically daunting task, especially when dealing with incomplete models. Other changes require domain knowledge for repair. For example, CS2 always requires crossword size to be specified, something not needed in CS1. Such a repair cannot be automated without domain knowledge. Finally, there are state-based relationships between events that require complex modeling.

TABLE 8
Unusable Test Scripts

	Original Scripts		Unusable Scripts		
	Scripts	Checkpoints	#	%	Checkpoints
CS2	101	343	94	93.1	334
PDF2	140	429	140	100	429
OT2	129	452	129	100	452

We have shown that a large percentage—as much as 100%—of the original test scripts become unusable after GUI modifications. We discussed the nature of the GUI modifications that made the test scripts unusable. **These results address RQ1**.

Repair Process: We trained 11 participants for 3 hours each on the usage of *SITAR*. They were then divided into 3 teams, each of which worked with a single application. The CS2, PDF2, and OT2 teams had 4, 3, and 4 members, respectively.

The process of using *SITAR* starts with ripping and obtaining the EFGs. The sizes of the resulting EFGs are shown in Table 9. The fraction of events recognized by the ripper, and hence added to the EFGs, are quite low for CS2 and OT2. **This addresses RQ2**. The missing events will need to be supplied manually by a tester during the repair process.

TABLE 9
Original Sizes of EFGs

	Vertices/Events	Edges
CS2	58 (of 102)	490
PDF2	158 (of 176)	15376
OT2	244 (of 389)	2914

Next, *SITAR* mapped all QTP events to logical events in the EFGs. As expected, not all events could be mapped. The actual numbers are shown in Table 10. Because of the low fraction of events recognized by the ripper for CS2 and OT2, a very small percentage of events from the scripts could be mapped to their logical counterparts.

TABLE 10
Events from Scripts Successfully Mapped

	Total Events #	Recognized Events #	Percent
CS2	3931	475	12%
PDF2	4456	2908	65%
OT2	4656	1619	35%

The iterative repair process started with *SITAR* attempting to repair each script, providing feedback to a human tester, and posing questions when needed, and recording the responses in the mapping or annotated EFG. A summary of the results is shown in Table 11. The results are quite encouraging for PDF2 and OT2, with 89% and 82% test scripts repaired, respectively. We also executed all repaired scripts on their respective applications. The code and event coverage of these scripts is also reasonably high (Table 12) relative to the original scripts. It is interesting that the coverage of repaired suites is quite close to the coverage of original test suites on some applications (like PDF2 and OT2). The percentage of repaired scripts is not as high for CS2; we revisit the reasons for this later. However, we find the code coverage of the 41% repaired is quite high taking into consideration that there are functional changes and new functionalities are introduced which will be impossible to cover by scripts from the original version.

TABLE 11
Repairing Results

	Originally Unusable	Repaired	Percent Repaired
CS2	94	39	41%
PDF2	140	125	89%
OT2	129	106	82%

TABLE 12
Coverage of Regression Test Scripts

Coverage (%)	CS2	PDF2	OT2
Code	68.3	51.2	62.3
Event	59.8	68.7	77.9

Finally, for **RQ3** we now address the issue of *Checkpoints*. We manually examined all the checkpoint outcomes (numbers shown under "Total" in Table 13) in our repaired scripts. Because we had already executed all test scripts on the new application versions, we knew which checkpoints had returned *TRUE* values (numbers shown under "OK" in Table 13); we manually verified that all these were correct. We were

left with a few checkpoints that failed. We manually examined the reasons for these failures.

TABLE 13
Checkpoints

	Total	OK	Failure Classes		
			I	II	III
CS2	162	151	3	3	5
PDF2	384	374	0	0	10
OT2	404	369	20	7	8

We identified 3 major reasons. (I) *Object Reference:* The checked GUI object is changed in the new version; the reference to this object should have been repaired so that QTP can access it; this was not done correctly by the tester. (II) *Expected Output:* The values of some properties (e.g., *Title*, *Label*) changed in the new version; the checkpoint still refers to the old values. These can only be fixed manually. (III) *Artifact of Execution:* In a small number of cases, while the checkpoints seem to be specified correctly, they fail unexpectedly. We attribute this to problems with test replay, primarily due to timing between QTP and the applications. We revisit this issue later. **This addresses RQ3.**

We measure cost in terms of time spent on each repair operation; and hence we count the number of operations. We distinguish between two types of operations: *Input*, which requires modification of a GUI object or inputting a QTP script line, and *Confirm*, which includes confirming the correctness of original script line, selecting a suggested path to repair test scripts, and deleting a script line.

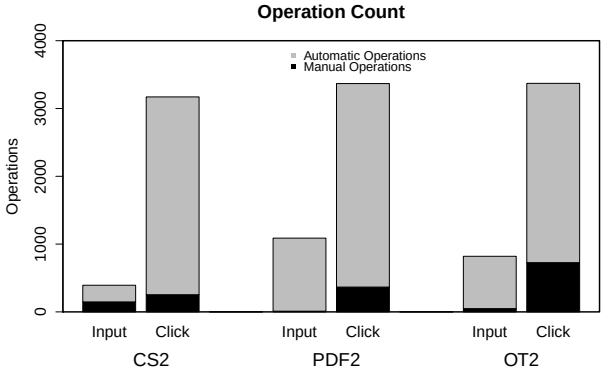


Fig. 10. Operation Cost of *SITAR*

Figure 10 shows that for most of the applications, the number of manual input operations is quite small compared to the automatic input operations. We also note that the proportion of manual confirm operations increases with the size of the application and the incompleteness of the initial EFG. OmegaT is the largest application and its initial EFG model misses many elements compared to the second largest application, PDFsam, thus it needs the greatest proportion of manual confirm operations. And for all the 3 subjects, the confirm operations performed by *SITAR* are much larger than those performed manually. **This addresses RQ4.** We informally note that the sizes of the subjects, the number of test scripts, completeness of the ripped models, changes between application versions effect the time of repairs. A more detailed study of these factors is subject for future work.

To evaluate the effectiveness of repairing knowledge accumulation of *SITAR*, we measure the manual cost of repairing in Figure 11. The x-axes show progress of the repairing process in terms of percentage. The y-axis of the left plot shows the *operation ratio* (the ratio between number of manual operations and lines of code of repaired test scripts); the right plot shows *time cost per line of code* (measured in seconds). The results show that (1) the cost in the early stage is significantly higher because of the incompleteness of initial model and lack of knowledge to perform fixes, (2) the manual operation cost declines quickly and achieves a low level at 20%-40% of repairing process, (3) the repairing time cost per line of *SITAR* is less than half a second most of the time. The time cost for CS2 and PDF2 is below 1 second even for the first 10% scripts. The time cost for OT2 is relatively high in the early stage but declines very quickly, primarily because the earlier manual repair decisions are reused automatically *SITAR* when the same situation arises for later test scripts. We now provide a finer-grained analysis of the nature of repairs and how they impact downstream repairs.

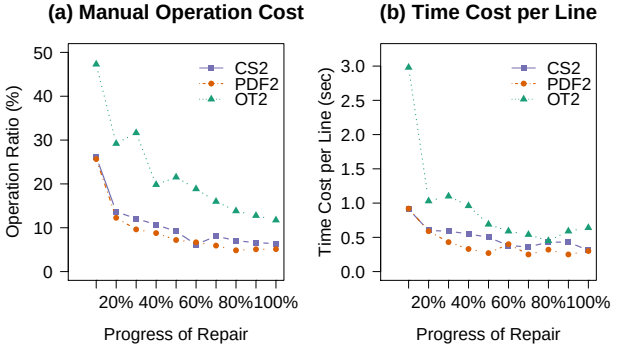


Fig. 11. Cost Effectiveness

We start with Table 14, which lists the number of different types of operations performed to repair the test scripts for each AUT. The rows marked *Manual* show the number of decisions (modify/add/confirm discussed earlier) performed by the tester. The modify operations can be repaired by simply using "search and replace", and they make up a big portion of all fix types in our case study. These decisions are "cached" into the model and annotations and reused by *SITAR* when the need for the same repair arises; these are shown in rows marked *Auto*. For example, the manual confirmation of event/edges happens when an

event/edge is missing in the current EFG model; once the tester confirms the repair, the event/edges in question are added to the EFG and all future confirmations of correctness of these events/edges is performed automatically.

TABLE 14
Number of Operations

		Modify	Add	Confirm
CS2	Manual	138	9	253
	Auto	50	98	2918
PDF2	Manual	10	0	366
	Auto	1078	0	3002
OT2	Manual	47	1	726
	Auto	770	2	2645

As explained in our algorithm, *SITAR* always tries to search for a path in the EFG when an edge needs to be repaired. So as an example, 9 manual additions in CS2 means that new events were suggested by *SITAR* to the testers through path search 9 times, and added to the EFG. The tester may simply confirm the *SITAR*-suggested path or make modifications to the suggested path. Subsequently, these 9 decisions were used 98 times to automatically repair future broken scripts.

In most cases, the number of manual modification/addition operations are much fewer than automatic operations. The only exception is the modification operation in CS2. This is caused by a bug (related to the "Save" dialog, and will be explained in detail when discussing reasons of failed repair).

We now show, in Table 15, the impact of repair decisions/operations, modify/add/confirm, on numbers of test scripts. We show the top 5 most influencial decisions per application. These 5 decisions impacted the largest number of test scripts. Column "F#" shows the unique identifier that we assigned to each fix operation. "#Infl" shows the number of scripts influenced by this individual operation. Column "#Tot Infl" shows the total number of test scripts influenced by all operations thus far - it is a cumulative sum of the "#Infl" entries. Column "#Fixed" shows the number unusable scripts that are fixed by the operations so far. Some obvious problems, such as the main-window-title problem for PDF2 and OT2, are ignored because they impact all scripts and thus make the "#Tot Infl" column always have the maximum value (i.e., total number of test cases in the test suite). We observe that for all 3 applications, there are a few fixes that are related to multiple test cases. But it is interesting to observe that fixing just the most influential one does not mean fixing the test case. Even if we ignore such universal problems as the main-window-title problem, most test cases cannot still be repaired by a single fix. For example, the first fix operation influences 79 test cases in PDF2 but none of them are fixed by this individual operation because these test cases require multiple fixes.

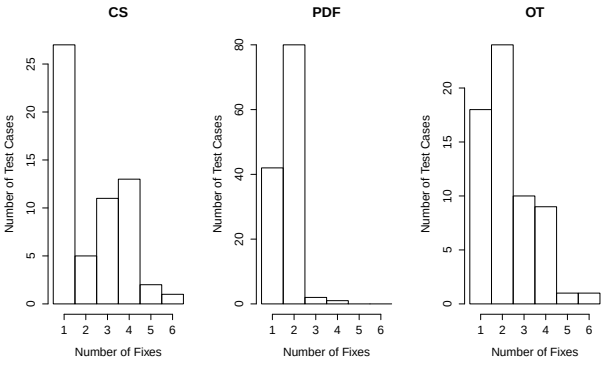


Fig. 12. Number of Fixes Required for each Test Cases

In addition, three histograms are provided in Figure 12 to show the distribution of number of fixes required for all test cases. And unsurprisingly, most test cases require very few (1, or 2) fixes, and there are very few test cases that require more than 5 fixes. **This addresses RQ5.**

TABLE 15
Individual Fixes and Test Cases Repaired

	F#	#Infl	#Tot Infl	#Fixed
CS2	1	47	47	22
	2	4	50	26
	3	1	51	26
	4	1	52	26
	5	1	53	26
PDF2	1	79	79	0
	2	29	84	28
	3	29	112	53
	4	24	119	75
	5	23	122	97
OT2	1	31	31	6
	2	29	36	22
	3	18	41	29
	4	11	52	30
	5	9	52	38

We analyzed the test scripts that *SITAR* failed to repair, and classified the reasons for failure as (1) Functionality Change of application, (2) Model Limitation, (3) Testers' Mistake, and (4) Others. The numbers of test scripts per class are shown in Table 16. As mentioned earlier in this section, the files created in CS1 could not be opened in CS2 because the data file format had changed. This caused 43 test scripts to remain unusable. We feel that such changes cannot be automatically handled by any test repair technique. An additional 6 test scripts could not be repaired because of model limitation. In both versions 0.3.3 and 0.3.4 of Crossword Sage, if a user tries to close the Main window, a modal dialog "Confirm Action" is shown. If the user tries to close the Main window *after* doing some edits without saving, a modal dialog "Save?" is shown. In version 0.3.3 after editing without saving, when the user clicks the "Save" menu a "Save" dialog opens; but if the user dismisses this dialog (without actually saving),

then the software incorrectly assumes that the file was saved. Subsequently, closing the Main window (incorrectly) opens the "Confirm Action" dialog. This bug was removed in Version 0.3.4; the same actions now open the "Save?" dialog. Even though this is due to a bug, we cannot model such context-specific behavior without state in the EFG. A richer state-based model is needed. We encountered similar model limitation problems in OT2.

TABLE 16
Test Scripts that *SITAR* could not Repair

Class	CS2	PDF2	OT2
Functionality Change	43	0	0
Model Limitation	6	0	10
Tester Mistake	2	7	3
Tool Problems	4	8	10

Because *SITAR*'s decisions are largely driven/confirmed by a human tester, any mistakes made by the tester will percolate to the models maintained by *SITAR*, which will finally impact the quality of repairs. The most common mistake made by testers was forgetting to modify or delete a QTP line. Sometimes testers also made incorrect decisions in repairs that are context-sensitive. We tried our best to develop a user-friendly tool that minimizes human error and confusion. In our implementation of *SITAR*, we provide a user interface with a display area in which scripts are shown line by line in run time, synchronized with the process of script checking. The scanning process stops and highlights, in red, the offending script line that needs repair by human intervention. *SITAR* provides multiple repair options to the user in the same interface so as to make it convenient to use. *SITAR* also automatically disables contradicting options to avoid mistakes; we include revoking mechanisms which allows the tester to revoke previous actions as long as the tester has not gone to the next script. We believe these approaches help reduce the risk of a tester to do an incorrect repair. However, as is the case with most user-driven software, users make mistakes. Indeed, in our experiment, we encountered a small number of problems that were caused by mistakes made by testers.

Finally, the nature of GUI testing test harnesses caused a small number of problems for test repair. These problems are an artifact of the orchestration between all the tools that we used. QTP, although it is a robust tool, must synchronize with the application that is being tested. Due to the nature of GUI replay tools, this synchronization is necessarily artificial because the application under test was never designed to work with an automated test harness; rather it was designed for human users. QTP uses mechanisms such as "wait for window" or "wait for widget" to continuously examine the screen for synchronization cues. Sometimes these mechanisms do not work as expected, causing test scripts to fail or hang. **This addresses RQ6.**

Summing up the Results: In this study, we showed the strengths of *SITAR*, in terms of the number of repaired scripts, their checkpoints, and code and event coverage. We also studied the weaknesses of *SITAR*, in terms of classes of scripts and checkpoints that we did not repair. As is the case with all studies, its results too are subject to threats to validity. To minimize threats to internal validity, we relied on robust tools, such as QTP and ripper for this work. We also ensured that our data is correct by continuously inspecting our data collection codes and results carefully with 3-4 participants for each application. To minimize threats to external validity, we used open-source applications as our subjects; we had no influence over their codes or evolution. However, we recognize that these applications do not represent the wide range of possible GUIs; results are expected to be different for other GUI application types. We also employed multiple testers to create the test scripts and carry out the repairs. However, they were all students. We realize that test scripts made in a University setting may not cover the wide spectrum of possible Industry features.

By sharing the data we used/generated in this study, we hope to encourage other researchers to enhance the study, thereby helping to further reduce threats to validity, and contribute to this much understudied problem of test repair.

6 Conclusion & Future Research Directions

We described *SITAR*, a new technique to repair low-level test scripts that have become unusable due to GUI modifications. Our work is unique in that we developed (1) new mechanisms to handle repairs without perfect knowledge of the GUI and its changes, (2) new annotations in an initially incomplete GUI model to facilitate repairs, (3) mapping between the code- and model-level to realize translation from code to model and vice versa, and (4) mechanisms to incorporate and cache human input into the overall process. Our results on 3 open-source software subjects are promising. We were able to repair a non-trivial fraction of an otherwise completely unusable test suite.

The work has laid the foundation for much future research. Our results showed that the stateless EFG model that we used caused a number of test scripts to remain unusable. For example, 6 scripts in CrosswordSage and 10 scripts in OmegaT could not be repaired because certain events in these scripts required the software to be in specific states to execute; however, this state-based information was not encoded in the EFG, which is why these test cases could not be repaired. Our *dominates* edge partially helps with the

issue of state/context by *requiring* the execution of specific events to setup the state for certain subsequent events. For example, in Crossword, a size of the crossword is required in the new version whereas all crosswords have a fixed default size in the old version. By annotating EFG edges along the path to "size" as dominates, testers ensured that the scripts setup the state with correct size before performing other events, resulting in usable repaired test scripts. We will study the use of better stateful models on the quality of repairs; at the same time, we will need to study issues of scalability and usability as state increases the complexity and size of models.

We will also examine the benefits and potential problems of additional automation. In our current work, we take a conservative approach to repair, i.e., all repair decisions are made by a human tester. *SITAR* reuses the manual decisions for subsequent repairs. We hypothesize that this conservative approach yields repaired scripts that are "closer" to testing the business logic originally intended by the script creator. Indeed, this is somewhat validated by the observation that all our checkpoints in the scripts remained intact and useful. In particular, we will examine three approaches towards additional automation. First, we will attempt to execute all scripts before repairing them, even if they are only partially executable, the intention being this will increase the completeness of our initial EFG model by adding more may-follow edges covered by the partial executions. However, such executions may also lead to an incorrect initial model as the modified software may be buggy and have incorrect flows – parts of test scripts may execute successfully when they should in fact have not. Second, we will push our algorithms to make certain decisions fully automatically without human input. The risk, of course, is that a fully automatic approach may lead to a repair that breaks the business logic of the original scripts. We recognize that there has to be a balance between automation and preservation of intent of test script to test a certain business logic. Such an approach requires empirical evaluation.

Third, we will explore approaches such as the one proposed by Grechanik et al. [5] to identify changes in GUI objects and report their locations in GUI test scripts to assist manual test repair. Additionally, analyzing text and finding similarities before/after modifications in the GUI may also help automate some repair of certain types of broken scripts. The challenges, of course, will be to come up with effective dictionaries that work across a range of software GUIs, text processing algorithms that are applicable to GUI lexicons, and image matching for widgets that do not have text labels, e.g., icons and toolbar buttons.

Our empirical evaluation demonstrated a range of modifications that we may term as simple (e.g., change of title) to complex (e.g., new context-based flow of execution). Indeed, all the changes shown under the *Modify* column, which make up the majority of our repairs, in Table 14 may be made by simply using text "search and replace". Hence, we may be able to map a range of repair transformations, from simple (finding and replacing title text) to complex (detecting state-based relationships), which we can use to develop a multi-step repair process. We envision the tester starting with the simplest transformation first, repairing scripts quickly repairable, and then focusing on scripts that are difficult to repair. We intend to study the impact, cost, quality of this process in future work. We expect however that some simple transformations, e.g., find/replace, if applied naively could in fact make scripts unusable.

As is the case with most research involving empirical evaluation and human subjects, our evaluation has several weaknesses, which we will address in future work. Our weaknesses stem mostly from the human cost of conducting the experiments. First, we trained a large number of volunteers to use our tools and applications. Second, we used a relatively large pool (hundreds consisting of thousands of lines) of manually developed test scripts. These scripts also included hundreds of checkpoints for each AUT; inserting checkpoints to check subtle business logic components is an expensive process. Third, we repeated the process of repair on each application multiple times to minimize accidental errors. Fourth, all repair reports and reasons why some scripts were not successfully repaired needed to be manually studied and categorized.

We have three concrete directions to improve our experiments. First, we intend to study the test repair process across multiple versions of the subject applications. Given that we already have test scripts for one version per application (say v_1), and that all our applications have at least one preceding version (v_0) and a later version (v_2), we can bootstrap this extension by first considering repairs from v_1 to v_0 (we have already shown v_1 to v_2), thereby doubling our empirical results without investing in manual collection of new test scripts (an extremely time intensive task). We can trace the evolution of test scripts as they undergo multiple repairs across versions, v_0 to v_1 to v_2, examining their ability to re-test functionality they were originally created to test. We expect that this ability will degrade as scripts undergo multiple repairs over long periods of time. Another related research question is to study the effects of size of change on quality and cost of repair. We hypothesis that if test repair is done after each commit, we will be able to quickly fix test scripts and maintain quality. We would consider our technique to work very well in such scenarios because smaller modifications will help to gradually improve the EFG model and mapping tables. The information collected in previous minor versions will probably still be valid after very minor changes and thus benefit the script repair in later

minor versions. On the other hand, if we wait for a large number of changes before the tests are repaired, the repair would be expensive and may yield low quality scripts. Much of this will depend on the nature of the software and nature of changes over time. We intend to study how this works in practice.

Second, we will explore the use of GUI mutations to give us a more controlled environment for experimentation. We can mutate the GUI using mutation operators, creating multiple versions with exactly a single change (the mutation). This controlled modification will allow us to study how changes affect test scripts. A major challenge for such a mutation-based approach has been the lack of commonly accepted GUI mutation techniques and tools. Our recent work in this direction [31] [32] has been promising and we believe that may make it possible to study GUI script repair using GUI mutations in our future work.

Third, we are working with an Industry partner to apply our test repair approach to their test scripts. While such an application will help us understand the strengths and weaknesses of our research, one intellectual outcome that we desire is to get a better understanding of metrics that test designers in Industry will use/develop to evaluate the "quality" of the repaired test scripts. Because they have never been exposed to the concept of test repair, they currently do not have such metrics. We envision metrics such as "how many of the originally covered statements are covered after the repair." However, we do not yet know whether they will quantify repairs on a per-script or per-suite basis.

7 Acknowledgments

This research is sponsored in part by National Basic Research Program of China (973 Program 2014CB340702), NSFC Program (61170067 and 61373013). This work was also supported by the US National Science Foundation under Grant No. CNS-1205501. In addition, we thank all the volunteers who contributed tirelessly to our empirical study.

References

[1] A. Memon, "Automatically Repairing Event Sequence-based GUI Test Suites for Regression Testing," *ACM Transactions on Software Engineering and Methodology*, vol. 18, no. 2, pp. 4:1–4:36, 2008.

[2] B. Daniel, T. Gvero, and D. Marinov, "On test repair using symbolic execution," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'10)*, 2010, pp. 207–218.

[3] B. Daniel, Q. Luo, M. Mirzaaghaei, D. Dig, D. Marinov, and M. Pezz, "Automated GUI Refactoring and Test Script Repair," in *Proceedings of the International Workshop on End-to-End Test Script Engineering (ETSE'11)*, 2011, pp. 38–41.

[4] A. Kervinen, M. Maunumaa, T. Pääkkönen, and M. Katara, "Model-based testing through a GUI," in *Proceedings of the 5th international conference on Formal Approaches to Software Testing*, 2006, pp. 16–31.

[5] M. Grechanik, Q. Xie, and C. Fu, "Maintaining and Evolving GUI-directed Test Scripts," in *Proceedings of the International Conference on Software Engineering (ICSE'09)*, 2009, pp. 408–418.

[6] A. Memon, "An Event-flow Model of GUI-based Applications for Testing," *Software Testing, Verification and Reliability*, vol. 17, no. 3, pp. 137–157, 2007.

[7] T. Lalwani, *QuickTest Professional Unplugged: 2nd Edition*. KnowledgeInbox, 2011.

[8] D. Burns, *Selenium 1.0 Testing Tools: Beginners Guide*. Packt Publishing, 2010.

[9] A. Memon, I. Banerjee, and A. Nagarajan, "GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing," in *Proceedings of the Working Conference on Reverse Engineering (WCRE'03)*, 2003, pp. 260–269.

[10] A. Memon and M. L. Soffa, "Regression Testing of GUIs," in *Proceedings of the European Software Engineering Conference Held Jointly With ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE'03)*, 2003, pp. 118–127.

[11] S. Huang, M. B. Cohen, and A. M. Memon, "Repairing gui test suites using a genetic algorithm," in *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ser. ICST '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 245–254. [Online]. Available: http://dx.doi.org/10.1109/ICST.2010.39

[12] C. Fu, M. Grechanik, and Q. Xie, "Inferring types of references to gui objects in test scripts," in *Software Testing Verification and Validation, 2009. ICST'09. International Conference on*. IEEE, 2009, pp. 1–10.

[13] S. Zhang, H. Lü, and M. D. Ernst, "Automatically repairing broken workflows for evolving gui applications," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ser. ISSTA 2013. New York, NY, USA: ACM, 2013, pp. 45–55. [Online]. Available: http://doi.acm.org/10.1145/2483760.2483775

[14] S. R. Choudhary, D. Zhao, H. Versee, and A. Orso, "WATER: Web Application TEst Repair," in *Proceedings of the International Workshop on End-to-End Test Script Engineering (ETSE'11)*, 2011, pp. 24–29.

[15] M. Leotta, D. Clerissi, F. Ricca, and C. Spadaro, "Comparing the maintainability of selenium webdriver test suites employing different locators: A case study," in *Proceedings of the 2013 International Workshop on Joining AcadeMiA and Industry Contributions to Testing Automation*, ser. JAMAICA 2013. New York, NY, USA: ACM, 2013, pp. 53–58. [Online]. Available: http://doi.acm.org/10.1145/2489280.2489284

[16] N. Alshahwan and M. Harman, "Automated session data repair for web application regression testing," in *Proceedings of the International Conference on Software Testing, Verification, and Validation (ICST'08)*, 2008, pp. 298–307.

[17] M. Mirzaaghaei, F. Pastore, and M. Pezze, "Automatically repairing test cases for evolving method declarations," in *Proceedings of International Conference on Software Maintenance (ICSM'10)*, 2010, pp. 1–5.

[18] ——, "Supporting test suite evolution through test case adaptation," in *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ser. ICST '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 231–240. [Online]. Available: http://dx.doi.org/10.1109/ICST.2012.103

[19] M. Mirzaaghaei, "Automatic test suite evolution," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 396–399. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025172

[20] B. Daniel, V. Jagannath, D. Dig, and D. Marinov, "Reassert: Suggesting repairs for broken unit tests," in *Proceedings of the International Conference on Automated Software Engineering (ASE '09)*, 2009, pp. 433–444.

[21] R. B. Evans and A. Savoia, "Differential testing: A new approach to change detection," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE '07. New York, NY, USA: ACM, 2007, pp. 549–552. [Online]. Available: http://doi.acm.org/10.1145/1287624.1287707

[22] K. Taneja, D. Dig, and T. Xie, "Automated detection of api refactorings in libraries," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated*

Software Engineering, ser. ASE '07. New York, NY, USA: ACM, 2007, pp. 377–380. [Online]. Available: http://doi.acm.org/10.1145/1321631.1321688

[23] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 54–72, Jan. 2012. [Online]. Available: http://dx.doi.org/10.1109/TSE.2011.104

[24] A. Memon and Q. Xie, "Using transient/persistent errors to develop automated test oracles for event-driven software," in *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, ser. ASE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 186–195. [Online]. Available: http://dx.doi.org/10.1109/ASE.2004.73

[25] Q. Xie and A. M. Memon, "Designing and comparing automated test oracles for gui-based software applications," *ACM Trans. Softw. Eng. Methodol.*, vol. 16, no. 1, Feb. 2007. [Online]. Available: http://doi.acm.org/10.1145/1189748.1189752

[26] T. Xie, "Augmenting automatically generated unit-test suites with regression oracle checking," in *Proceedings of the 20th European Conference on Object-Oriented Programming*, ser. ECOOP'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 380–403.

[27] W. Yang, Z. Chen, Z. Gao, Y. Zou, and X. Xu, "Gui testing assisted by human knowledge: Random vs. functional," *J. Syst. Softw.*, vol. 89, pp. 76–86, Mar. 2014. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2013.09.043

[28] L. S. Pinto, S. Sinha, and A. Orso, "Understanding myths and realities of test-suite evolution," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 33:1–33:11. [Online]. Available: http://doi.acm.org/10.1145/2393596.2393634

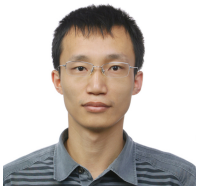
[29] B. N. Nguyen and A. M. Memon, "An observe-model-exercise* paradigm to test event-driven systems with undetermined input spaces," *Software Engineering, IEEE Transactions on*, vol. 40, no. 3, pp. 216–234, 2014.

[30] A. M. Memon, M. L. Soffa, and M. E. Pollack, "Coverage criteria for gui testing," in *ACM SIGSOFT Software Engineering Notes*, vol. 26, no. 5. ACM, 2001, pp. 256–267.

[31] R. A. P. Oliveira, A. Emil, Z. Gao, and A. Memon, "Definition and evaluation of mutation operators for gui-level mutation analysis," in *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*. IEEE, 2015, p. to appear.

[32] E. Algroth, Z. Gao, R. A. Oliveira, and A. Memon, "Conceptualization and evaluation of component-based testing unified with visual gui testing: an empirical study," in *Proceedings of the 2015 Eighth International Conference on Software Testing, Verification and Validation*, ser. ICST '15. Washington, DC, USA: IEEE Computer Society, 2015, p. to appear.

Zebao Gao is a PhD student at the Department of Computer Science, University of Maryland, College Park. Zebao received his BS and MS degrees from Nanjing University, China. He used to work as a software-development intern at eBay CDC and a research intern at Baidu Inc. His previous research experiences include program fault localization, test script repairing and testing coverage criteria. Since 2013, Zebao becomes a research assistant at the EDSL lab where he works on the GUITAR and Comet projects. His current research interests include GUI testing, reverse engineering and program analysis. He is applying empirical study and building automatic testing techniques and frameworks to bridge the gaps between research and testing practices in industry.



Zhenyu Chen is currently an Associate Professor at the Software Institute, Nanjing University. He received his bachelor, and Ph.D. in Mathematics from Nanjing University. He worked as a Postdoctoral Researcher at the School of Computer Science and Engineering, Southeast University, China. His research interests focus on software analysis and testing. He has about 80 publications in journals and proceedings including TOSEM, TSE, JSS, SQJ, IJSEKE, ICSE, FSE, ISSTA, ICST, QSIC etc. He has served as PC co-chair of QSIC 2013, AST2013, IWPD2012, and the program committee member of many international conferences. He has won research funding from several competitive sources such as NSFC. He is a member of the IEEE.

Yunxiao Zou is a PhD Student at the Department of Computer Science of Purdue University. He received BS and MS from Nanjing University, China. His research interest includes software testing, program analysis and profiling. In addition to his interests in Computer Science, he likes mathematics as well.

Atif Memon is a Professor at the Department of Computer Science, University of Maryland, where he founded and heads the Event Driven Software Lab (EDSL). His research interests include software security, program testing, software engineering, experimentation, and computational biology. He is currently working in all these areas with funding from the US National Institutes for Health, the US Defense Advanced Research Projects Agency, the US National Security Agency, and the US National Science Foundation. He is currently serving on a National Academy of Sciences panel as an expert in the area of Computer Science and Information Technology, for the Pakistan-U.S. Science and Technology Cooperative Program, sponsored by United States Agency for International Development (USAID). In addition to his research and academic interests, he handcrafts fine wood furniture.