e-ISSN: 2348-6848
p-ISSN: 2348-795X
Volume 05 Issue 07
March 2018

Sitar: Gui Test Script Repair

Gangadhar

P. Veera muthu

ABSTRACT — — The system tests of a GUI-based application require that the test cases, consisting of sequences of user actions / events, be executed and the output of the software verified. To allow a new automated test, such test cases are increasingly encoded as low-level test scripts, to be automatically reproduced using test harnesses. Every time the GUI changes, the widgets move, the windows merge, some scripts become unusable because they no longer encode the valid input sequences. In addition, because the software output may have changed, it is possible that your test-assertion oracles and checkpoints-encoded in the scripts no longer correctly check the desired GUI objects. Introducing 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 reparative transformations and human input to repair the test, and synthesizes a new test script "repaired." During this process, SITAR also repairs the reference to the GUI objects used in the control points, producing a final test script that can be executed automatically to validate the revised software. SITAR amortizes the cost of human intervention in multiple scripts by accumulating human knowledge as annotations in the EFG. An experiment using QTP test scripts suggests that SITAR is effective because 41-89 percent of the unusable test scripts were repaired. The annotations significantly reduced the human cost after the 20 percent test scripts were repaired.

Index terms: GUI test, GUI test script, test script repair, human knowledge accumulation

An important problem in the Autorun test for graphical software applications is that a large number of unacceptable tests may become unusable each time the software is modified [1], [2], [3]. However, the GUI test can not be avoided: when the only way to interact with a program is the graphical user interface (GUI), the system test, ie the whole program test, [4] requires testing using the graphical user interface. The graphical user interface (GUI) test case consists of strings of user actions / events that are performed on the program by graphical user interface (GUI) elements and checkpoints that determine whether the program was executed correctly or not. These system tests are closely related to the GUI structure, meaning they refer to certain GUI tools and encryption sequences ("First click the list of files, click Open menu item, select a file, then click Open button") allowed It is through the GUI. During maintenance, if the GUI changes, some tests become unusable either because (1) the sequence of events that the graphical user interface is modified is no longer allowed, or (2) their test points (assertions) no longer scan the objects correctly GUI-oriented objects are unusable and are not a problem when you manually test the graphical user interface. The human laboratory performs the test cases according to the test plan and manually verifies the validity of the outputs [5]. If what they see in the GUI is different from what the test plan describes, they are often able to use common sense about minor changes, whether they have encountered an error or a deliberate change, and if necessary, review the test plan. In contrast, unmanageable test cases are a major problem of automation. If you experience an automated test tool (unexpected screen, UI) that is different from what you expect, it simply fails or stops. Although autorun is desirable because text tests can run multiple times, their benefits are rapidly diminished by high maintenance costs when large numbers of tests become unusable and require re-encoding or re-registration [1] each time the GUI is modified . In the previous work, we dealt in part with the problem of unusable test cases. However, we focused only on high-level test cases based on models, which were represented as abstract events, not as scripts. Originally created with auto-built methods [6], some of these test cases in our previous work have become unusable due to software modifications. We have corrected the test cases based on this model [1] by developing a new "fix" technique that matches the unusable test cases at the model level and converts them into new test cases that are usable at the model level.

In SITAR design, we provide the following research contributions. EFG format mechanisms are automatic / imprecise automatically with human input to repair complex test texts while retaining the ability of scripts to test AUT logic. Mechanisms to deal with repairs without full knowledge of the graphical user interface and their changes. Based on the current approximation of the EFG group, the reform proposals are automatically sent to human laboratories, which then choose to fix the most applicable fix. Mechanisms to integrate and synthesize human inputs into the overall process that speed up the process and produce a more accurate EFG model. Mechanisms for the repair of checkpoints at checkpoints. Most checkpoints remain valid in the repaired scripts, indicating that the logic of the encrypted business in the original test scripts still exists after the fixes. New explanations in EFG to facilitate reforms. Specifically, we introduce a new entry that controls the edges. Match the code and model level to achieve code-to-form translation and vice versa. Output the EFG model annotated from AUT and is more accurate than the model that was automatically obtained by reverse engineering. These annotations take the form of definite events / edges added by a human laboratory. EFG speeds up the rigorous repair process as well as the ability to improve test testing and repair for later versions of AUT.

Existing System:

Most of the GUI tests used in the industry are encrypted as text (eg, VBScript) or manually recorded for exchange by the testing device / tools (eg HPQuickTest Professional (QTP), Selenium. These issues have maintenance problems and need to be hindered. Cases are very important and relevant because they are usually based on carefully selected cases and functional requirements. Human testers invest their time in transferring their knowledge and experience in test texts and checkpoints to business logic in a comprehensive manner, Complex and valuable.

 Pinto et al. Provides an excellent treatment of the reality surrounding the development and maintenance of the test kit. They discuss different real-life cases in which test cases are added, removed and reformulated in practice. They also point out that, different from previous cases, repair testing is more complex and difficult to automate, and current testing methods that focus on assertions may not be actually applicable.

Disadvantages of existing system:

 In the previous work, partially addressed the problem of unusable test cases. However, only focused on test cases based on a high-level model, represented in abstract events, not as scripts.

 Originally created automatically using modelbasedapproaches, some of these test cases in earlier versions became unusable due to software modifications.

 Previous technology does not directly rely on handwritten test cases for a number of reasons. First, an event flow diagram (EFG) - formed, the basis of the reform was insufficient because it lacked vital information (eg, status, annotations) required to reform the text.

 Secondly, the way we obtained from the EFGs of the GUIs (using reverse engineering via the Rippingbased GUI on the rear - firsttraversal) was very limited.

 Ripping GUI is based on dynamic analysis and therefore suffers from incomplete underlying implicit analyzes, resulting in partial EFGs. We can not use partEFGs to fix the script because often, these scripts contain events absent from partial EFGs.

Proposed System:

 In this paper, we provide ScrIpT repAireR (SITAR), a technology for repairing low-level test texts that are unusable. By increasing the level of abstraction of unusable test scripts - from the level of script language to the form level - SITAR is working on applying model-based transformations for usable test cases at the model level, and then synthesizing new low-level usable scripts.

 To make the repair, it relies on incomplete and inaccurate EFG models of the GUIs, as well as human inputs.

 SITAR is used to repair transformations and human inputs (as explanatory notes) to fix test cases in ts0 that can execute and confirm assertions on A1.

Advantages of the proposed system:

 Our work is unique in the sense that we are also reforming the expected output (ie, the Oracle test) as part of the test cases.

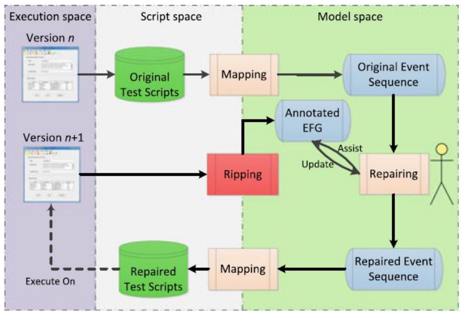
 Our business is unique in its development

 New mechanisms for dealing with reforms without full knowledge of the GUI and its changes,

Ann Annotations New Graphical User Interface Incomplete To facilitate fixes,

 Mapping between the level of code and level to access the translation from the code to the model and vice versa, and

 The mechanism involves the insertion and removal of human inputs into the overall process.SYSTEM ARCHITECTURE:



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 three 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, six 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.

Author Details

Gangadhar

P. Veera muthu