Maintaining and Evolving GUI-Directed Test Scripts

Mark Grechanik, Qing Xie, and Chen Fu
Accenture Technology Labs
Chicago, IL 60601
{mark.grechanik, qing.xie, chen.fu}@accenture.com

Abstract

Since manual black-box testing of GUI-based APplications (GAPs) is tedious and laborious, test engineers create test scripts to automate the testing process. These test scripts interact with GAPs by performing actions on their GUI objects. An extra effort that test engineers put in writing test scripts is paid off when these scripts are run repeatedly. Unfortunately, releasing new versions of GAPs with modified GUIs breaks their corresponding test scripts thereby obliterating benefits of test automation.

We offer a novel approach for maintaining and evolving test scripts so that they can test new versions of their respective GAPs. We built a tool to implement our approach, and we conducted a case study with forty five professional programmers and test engineers to evaluate this tool. The results show with strong statistical significance that users find more failures and report fewer false positives ($p < 0.02$) in test scripts with our tool than with a flagship industry product and a baseline manual approach. Our tool is lightweight and it takes less than eight seconds to analyze approximately 1KLOC of test scripts.

1 Introduction

Manual *black-box testing* of *Graphical User Interface (GUI)-based APplications (GAPs)* is tedious and laborious, since nontrivial GAPs contain hundreds of GUI screens and thousands of GUI objects. Test automation plays a key role in reducing high cost of testing GAPs [9][10][6]. In order to automate this process, test engineers write programs using scripting languages (e.g., JavaScript and VBScript), and these programs (*test scripts*) mimic users by performing actions on GUI objects of these GAPs using some underlying testing frameworks. An extra effort put in writing test scripts is paid off when these scripts are run repeatedly to determine if GAPs behave as desired.

Crafting test scripts from scratch is a significant investment. Test engineers implement sophisticated *testing logic*, specifically they write code that processes input data, uses this data to set values of GUI objects, acts on them to cause GAPs to perform computations en route, retrieves the results of these computations from GUI objects, and compares these results with oracles to determine if GAPs behave as desired. Reusing testing logic repeatedly is the ultimate goal of test automation.

Unfortunately, releasing new versions of GAPs with modified GUIs breaks their corresponding test scripts thereby obliterating the benefits of test automation [12][5]. Consider a situation when a list box is replaced with a text box in the successive release of some GAP. Test script statements that select different values in this list box will result in exception when executed on the text box. This simple modification may invalidate many statements in test scripts that reference this GUI object. This and many other similar modifications are typical between successive releases of different GAPs, including such well-known GAPs as Adobe Acrobat Reader and Microsoft Word. As many as 74% of the test cases become unusable during GUI regression testing [16], and internal evaluations of automated testing in Accenture show that even simple modifications to GUIs result in 30% to 70% changes to test scripts. To reuse these scripts, test engineers should fix them, and this process is laborious and intellectually intensive.

Given the complexity of these scripts, it takes from hours to days to fix them, so that they can test successive releases of the corresponding GAPs. Existing tools detects exceptions in test scripts at runtime, i.e., test engineers must run these scripts (often for many hours because these scripts contain loops) in order to execute statements that reference modified GUI objects. Exceptions interrupt continuous testing and they require human intervention to fix them, and it defeats the purpose of test automation.

After interviewing professional testers at different companies we determined that they often discard old test scripts and write new ones from scratch when new versions of GAPs are released, due to the reasons mentioned above. When rewriting scripts under time pressure, test engineers often implement key components of testing logic poorly.

Since test scripts are not shipped with their corresponding GAPs to customers, these scripts are rarely checked for adherence to requirements. In general, newly written scripts are less sophisticated and effective than the old scripts, which have been used and refined for some time.

Test engineers often lack time and necessary skills to understand and fix old scripts, especially if these scripts were created by other engineers. Existing approaches provide little help to address this pervasive and big problem. Currently, over six thousand of test personnel at Accenture deal with test scripts. The annual cost of manual maintenance and evolution of test scripts is estimated to be between $50 to $120 millions just in Accenture alone.

Models of GUIs can guide script-based testing by checking operations in scripts against elements of GUI models. Specifically, model-based GUI regression testing involves building and comparing high-level models of GAPs before applying algorithms that construct test cases for evolved GAPs [21]. GUI models can also be extracted from the source code of GAPs. However, there are two fundamental limitations to extracting models from GAPs' source code.

First, in black-box testing, source code of GAPs is not available [4], and GUI testing is inherently black-box since operations are performed on GUI objects rather than objects in the source code. Currently, testing is often outsourced to external organizations, and the source code is not shared with these organizations for many reasons. Thus, testing organizations must proceed with black-box testing, and deriving precise GUI models from source code is not an option.

Second, even if the source code is available, there are limitations that render approaches of deriving GUI models from source code ineffective. Consider a situation when GUI objects are created using the *Application Programming Interface (API)* call `CreateWindow`, which is used in a large number of Windows GAPs. This API call takes a number of parameter variables including a string variable that holds the value of the type of the GUI object. The value of this variable is often known only at runtime, making it impossible to derive GUI models from the source code.

In addition, deriving models from the source code depends on knowing the precise semantics of API calls that create and manipulate GUI objects (e.g., `CreateWindow`), building appropriate parsers and analyzers for languages which are used to create GUI applications, and developing *Integration Development Environment (IDE)*-specific tools that extract GUI models from IDE GUI resource repositories. The number of tuples is measured in tens of thousands in the Cartesian product of API calls $\times$ programming languages $\times$ IDEs. This large number of combinations makes it difficult to come up with an approach that would work with source codebases of different GUI applications.

As it turns out, multiple disparate type systems make automated GUI testing very difficult. Existing regression testing approaches work in settings where test harnesses are written in the same language and use the same type system as the programs that these harnesses test (e.g., JUnit test harnesses are applied to Java programs). In contrast, when testing GAPs two type systems are involved: the type system of the language in which the source code of the GAP is written and the type system of the language in which test scripts are written. When the type of the GUI object is modified, the type system of the test script "does not know" that this modification occurred, thereby aggravating the process of maintaining and evolving test scripts.

Our contribution is a novel approach that automatically identifies changes between GUI objects and locates test script statements that reference these modified GUI objects. The input is GUIs of the successive releases of the same GAP and the test script for the prior release of the GAP. These GUIs are compared and modified GUI objects are located. Then, the test script is analyzed statically to invalidate statements that reference these modified GUI objects. This analysis results in warnings that enable test engineers to fix errors in test scripts in a way similar to how compilers issue warnings that enable programmers to fix programs.

We built a tool based on our approach as an Eclipse plugin, it is lightweight, and it takes less than eight seconds to analyze approximately 1KLOC of test scripts. We conducted a case study with 45 professional programmers and test engineers to evaluate our approach. The results showed with strong statistical significance that users find more test script statements and report fewer false positives ($p < 0.02$) in test scripts with our tool than with a flagship industry testing tool and a manual approach.

2 The Problem

In this section, we give background on test automation with scripts and formulate the problem statement.

2.1 Background

The objectives of test automation are, among other things, to reduce the human resources needed in the testing process and to increase the frequency at which software can be tested. Traditional *capture/replay* tools provide a basic test automation solution by recording mouse coordinates and user actions as test scripts, which are replayed to test GAPs. Since these tools use mouse coordinates, test scripts break even with the slightest changes to the GUI layout.

Modern capture/replay tools (e.g., *Quick Test Professional (QTP)*[1], *Abbot*[2], *Selenium*[3], and *Rational Functional*

[1] http://en.wikipedia.org/wiki/QuickTest_Professional
[2] http://abbot.sourceforge.net
[3] http://selenium.openqa.org

Tester (RFT)[4] avoid this problem by capturing values of different properties of GUI objects rather than mouse coordinates. This method is called *testing with object maps*, and its idea is to reference GUI objects by using unique names in test scripts. Test engineers assign unique names to collections of the values of the properties of GUI objects, and they use these names in test script statements to reference these objects.

In testing with object maps, the pairs (`uname`, {<p, v>}), where {<p, v>} is the set of the pairs of values `v` of the properties `p` of a GUI object, are collected during capture and stored in *object repositories (ORs)* under the unique name `uname`. During playback, the references to "`uname`" in scripts are translated into operations that retrieve {<p, v>} from ORs, and the referenced GUI object is identified on the screen by matching the retrieved values against its properties. This extra level of indirection adds some flexibility since cosmetic modifications to GUI objects may not require changes to test scripts. Changing the GUI object property values in the OR ensures that the corresponding GUI objects will be identified during playback.

However, many changes still break scripts, for example, changing the type of a GUI object from the list box to the text box. We define test script statements that access and manipulate GUI objects as *failures* if these statements are broken because of modifications made to the referenced GUI objects in the successive releases of GAPs. Test engineers put a lot of efforts in detecting and understanding failures, so that they can fix test scripts to make them work on modified versions of GAPs.

2.2 Test Automation Model

A test automation model that illustrates interactions between test scripts and GAPs is shown in Figure 1. Statements of test scripts are processed by the scripting language interpreter that is supplied with a testing platform. When the interpreter encounters statements that access and manipulate GUI objects, it passes the control to the testing platform that translates these statements into a series of instructions that are executed by the underlying GUI framework and the operating system.

At an abstract level we can view the layers between test scripts and GAPs as a reflective connector. A connector is a channel that transmits and executes operations between test scripts and GAPs. Reflection exposes the type of a given GUI object, and it enables test scripts to invoke methods of objects whose classes were not statically known before the GAP is run. This model combines a connector between scripts and GAPs with reflection so that test scripts can access and manipulate GUI objects at run-time.

[4]http://www-306.ibm.com/software/awdtools/tester/functional

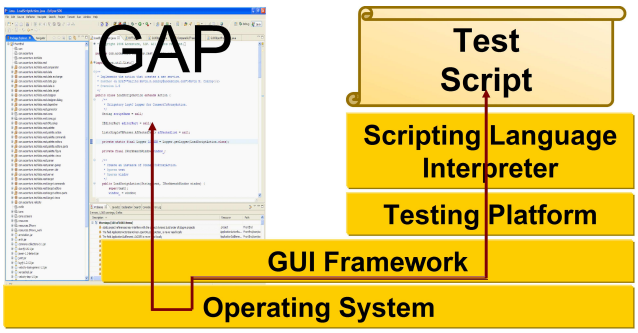


Figure 1. A model of interactions between test scripts and GAPs.

Each statement in test scripts, which accesses and manipulates GUI objects consists of the following operations: `(1)` navigate to some destination GUI object and `(2)` invoke methods to perform actions on this object, including getting and setting values. Using implementations of the concepts of reflection and connector, statements in test scripts can navigate GUI objects in GAPs and perform operations on these objects. This is the essence of the current implementations of test automation tools.

2.3 Fundamental Problems

Several fundamental problems make it difficult to maintain and evolve test scripts. First, specifications for GUI objects are often not available, and these objects are created dynamically in the GAPs' processes and the contexts of the underlying GUI frameworks (e.g., Windows or Java SWT). In this paper we deal with black-box testing, so obtaining information about GUI objects from the source code of GAPs is not an option. Therefore, test engineers have to use capture/replay tools to extract values of properties of GUI objects, so that these objects can be later identified on GUI screens by matching these prerecorded values with the properties of GUI objects that are created at runtime. Because complete specifications of GUI objects are not available, it is difficult to analyze statically how GUI objects are accessed and manipulated by test script statements.

The other problem is that test scripts are run on testing platforms externally to GAPs, and therefore cannot access GUI objects as programming objects that exist within the same programs. Using API calls exported by testing platforms is a primary mode of accessing and manipulating GUI objects, and these API calls lead to various run-time errors in test scripts especially when their corresponding GAPs are modified.

Consider a test script statement written using QTP `VbWindow("Login").VbButton("DoIt").Click`.

The API calls `VbWindow` and `VbButton` are exported by the QTP testing framework. Executing these API calls identifies a window whose property values match those stored in some OR under the name "`Login`," and this window contains a button whose property values match those stored in some OR under the name "`DoIt`". By calling the method `Click`, this button is pressed. Since API calls take names of the property values of GUI objects as string variables, and GUI objects are identified only at runtime, it is impossible to apply effective sound checking algorithms. These problems exacerbate the process of detecting and understanding failures in test scripts, making maintenance and evolution of these scripts expensive and prohibitive.

Our investigation revealed that these fundamental problems are inherent for most existing open-source and commercial automated testing tools. In this paper, we concentrate on QTP, a flagship automated testing tool manufactured by Hewlett-Packard Corp[5].

2.4 Current Approach

Currently, test engineers run test scripts that are written for the previous releases of a GAP on the successive releases of this GAPs to determine if these scripts can be reused. They use some existing tool that includes a script debugger (e.g., QTP). Once a statement that accesses a modified GUI object is reached, the testing platform generates an exception and terminates the execution of the script. The engineer analyzes the exception, fixes the statement, and reruns the script again. This process is repeated until the script runs without throwing any exceptions.

Often it takes a long time until statements that reference changed GUI objects are executed. Test scripts contain loops, branches, and fragments of code that implement complicated testing logic in addition to statements that access GUI objects. Consider a test script that contains a loop with code that reads in and analyzes data from files, computes some result from this data, and inserts it in some GUI object. Computing this result may take hours depending on the sizes of the files. Test scripts often contain multiple computationally intensive loops that are interspersed with statements that access GUI objects. Each time an exception is thrown because of a failure, the results of the execution are discarded, and the script should be rerun after engineers fix this failure. Commenting out loops (when possible) speeds up execution, but it changes the logic of test scripts, and subsequently the quality of repairs.

In addition, existing testing tools provide little information about how to fix failures in test scripts. When a test script is executed against a new version of the GAP, existing tools have no information about changes between GUI objects that lead to exceptions. As a result, test engineers must analyze GUIs manually to obtain this information and relate it to the exceptions, and this is a laborious and intellectually intensive process.

[5]While the worldwide market for automated test tools is over $1.1Bil, QTP is used by over `90%` of Fortune `500` companies [3].

2.5 The Problem Statement

Our approach helps test personnel only when modifications of GUI objects affect corresponding test scripts. We do not consider cases when test scripts should be evolved independently of GUI changes, for example, when test engineers determine that they should add new testing logic to test scripts.

We do not attempt to make our approach sound and complete. A sound approach ensures the absence of failures in test scripts if it reports that no failures exist, or if all reported failures do in fact exist, and a complete approach reports all failures, or no failures for correct scripts. Our approach should statically detect failures that result from modifications of GUI objects with a high degree of automation and good precision.

3 Our Solution

In this section, we present core ideas behind our approach that we call *Reducing Effort in Script-based Testing (REST)* and we describe the REST architecture.

3.1 Core Ideas

In order to enable checking of references to GUI objects in test scripts statically, we base our solution on three core ideas. First, we should compare GUIs of the successive releases of GAPs to determine what GUI objects are modified. Second, using the results of this comparison we detect what references to GUI objects in test scripts are affected by modified GUI objects. Finally, once it is known what statements in test scripts are affected by the modifications to GUIs, we should analyze these scripts to determine what other statements are affected as a result of using values computed by the statements that reference modified GUI objects.

3.2 Architecture

The architecture of REST is shown in Figure 2. Solid arrows show command and data flows between components, and numbers in circles indicate the sequence of operations in the workflow. The inputs to REST are two GUIs of the versions `N` and `N+1` of the running GAP, the OR, and the test script for the GAP of the version `N`.

The first step involves modeling the GAPs. GUIs of GAPs are modeled as trees whose nodes are composite GUI objects (e.g., frame) that contain other GUI objects,

leaves are primitive or simple GUI objects (e.g., buttons), and parent-child relationships between nodes (or nodes and leaves) defines a containment hierarchy. The root of the tree is the top container window.

The GUI Modeler (1) obtains information about the structure of the GUI and all properties of individual objects and it outputs (2) GUI trees GT_n and GT_{n+1} for the versions N and N+1 respectively. These trees are compared (3) by the GUI tree Comparator in order to determine what GUI objects are modified between the versions of the GAPs. The Comparator outputs (4) the GUI difftree that is a combination of the GUI trees GT_n and GT_{n+1} with matched GUI objects mapped to each other.

In general, it is an undecidable problem to compute a correct mapping function between two GUI trees fully automatically. We briefly describe our semiautomatic mapping algorithm in the Section 4.2, and we provide a utility (5) for the user to modify mappings between GUI objects manually. Normally, given that GUI screens contain less than a hundred GUI objects, fixing incorrectly identified mappings manually does not require any serious effort.

The Script Analyzer is a component that analyzes test scripts to determine the impact of GUI changes on these scripts. To do that, (7) the script for GAP of the version N is parsed using the Script Parser and (8) the parse tree is generated. This tree contains an intermediate tree representation of the test script where references to GUI objects are represented as nodes.

Recall that GUI objects are described using unique names with which property values of these objects are indexed in ORs. These names are resolved (10) into the values of properties of GUI objects using the component OR Lookup (11) with which the Script Analyzer interacts when it performs analyses.

Thus the Script Analyzer takes (9) the parse tree, (6) the GUI difftree, and (11) values of properties of GUI objects as its inputs and (12) produces a change guide that contains messages about possible failures in test scripts.

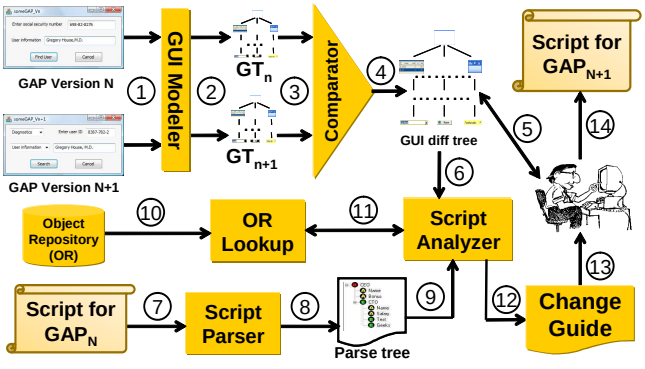


Figure 2. The architecture of REST.

For example, a message says that at line 23 the script S sets a value of GUI object ⟨Login, Name⟩, however, the type of this object is changed from EditBox to ListBox. Test engineers (13) review these messages and modify the original test script for the GAP of the version N so that it can test the successive version N+1 of this GAP. In addition, they make corresponding modifications to the property values of the GUI objects in the OR.

We built REST as a plugin for Eclipse based on this architecture, and we briefly review core features of our implementation in the next section. A short movie demonstrating how REST works is available at our website[6].

4 Implementation

There are two challenging aspects of our implementation of REST: a reflective connector and an impact analysis component that uses the results of a GUI model comparison algorithm. In this section, we briefly describe our implementation of these components of REST.

4.1 Accessibility As a Connector

Since we cannot access and manipulate GUI objects as pure programming objects (they only support user-level interactions), we use *accessibility technologies* as a universal mechanism that provides programming access to GUI objects. Accessibility technologies provide different aids to disabled computer users (e.g., screen readers for the visually impaired). Most computing platforms include accessibility technologies since it is mandated by the law [1].

The main idea of most implementations of accessibility technologies is that GUI objects expose a well-known interface that exports methods for accessing and manipulating these objects. For example, a Windows GUI object should implement the IAccessible interface in order to be accessed and controlled using the *Microsoft Active Accessibility (MSAA)* API calls. In REST, accessibility serves as a uniform reflective connector that enables test scripts to access and control GUI objects of GAPs (as described in Section 2.2), so that REST does not depend on specific (and often proprietary) testing platforms.

4.2 Impact Analysis

The impact analysis component computes the impact of modified GUI objects on statements that reference these objects, and the impact of affected statements on other statements in test scripts. The latter is accomplished by a combination of routine control and data-flow analyses, while the former is a function of GUI object modifications.

[6] http://www.markgrechanik.com/Rest/Rest.html

Using MSAA, REST traverses the GUI tree starting from the root, i.e., the top container window. Once at some GUI object, REST extracts values of its properties and encodes them in an internal representation format. These properties are the name of the object, its coordinates, type, position in the tree hierarchy, and the style (e.g., color, type of the border, modality). These properties are used to compare trees in order to find modified GUI objects.

We designed and built a comparison algorithm for computing mappings between GUI objects of GUI trees. This algorithm takes as its input the GUI trees Γ_n and Γ_{n+1} for two successive releases of the same GAP and computes the mapping set μ that contains relations between GUI objects θ and ρ of these trees along with their match score σ, $((\theta,\rho) \in \mu, \sigma), \theta \in \Gamma_n, \rho \in \Gamma_{n+1}, 0 \leq \sigma \leq 1$. The main idea of the algorithm is to compute the match score σ for each pair of GUI objects between successive releases of the GAP, and the final mappings between GUI objects are determined on a basis of the highest scores.

The score is computed as the normalized weighted sum $\sigma = \Sigma w_p |p_\theta - p_\rho|$, where p_θ and p_ρ are the values of the property p of the GUI objects θ and ρ, and $0 \leq w_p \leq 1$ is the weight that is assigned to the property p. When $\sigma = 1$, the objects are identical, when $\sigma = 0$, the objects do not match at all, and when $0 < \sigma < 1$, the match is partial. Given m objects in the GUI trees Γ_n and n objects in Γ_{n+1}, the algorithm produces mn scores for each pair of objects. The mapping is confirmed if σ exceeds a certain threshold value, 0.1, which we chose experimentally.

The weights for the properties of GUI objects were also chosen experimentally. Experimental results showed that using this algorithm between 60% to 80% of mappings are identified correctly. Incorrectly computed mappings affect the precision with which REST identifies failures in test scripts, leading to more false positives. However, as the case study showed, even with this precision REST performs better than competitive approaches.

5 Case Study Design

To determine how effective REST is, we conduct a case study with 45 participants. Our goal is to evaluate how well these participants can find failures in test scripts (when running against the new version of the GAP) using three different approaches: manual, using *Quick Test Pro (QTP)* (a flagship industrial testing tool from HP), and using REST as a guiding tool. Specifically, we want to determine using what approach users can report more *correctly identified failures (CIF)* in test scripts that result from changed GUI objects between successive releases of the subject GAPs, and with what approach users report fewer false positives (FPs), i.e., correct statements in test scripts that participants report as failures by mistake.

5.1 Hypotheses

We introduce the following null and alternative hypotheses to evaluate how close the means are for the CIFs and FPs for control and treatment groups. Unless we specify otherwise, participants of the treatment group use REST, and participants of the control group use either manual approach or QTP. We seek to evaluate the following hypotheses at a 0.05 level of significance.

H_0 The primary null hypothesis is that there is no difference in the numbers of CIFs and FPs between participants who attempt to locate failures in test scripts manually, using QTP, or REST.

H_1 An alternative hypothesis to H_0 is that there is statistically significant difference in the numbers of CIFs and FPs between participants who attempt to locate failures in test scripts manually, using QTP, or REST.

Once we test the null hypothesis H_0, we are interested in the directionality of means, μ, of the results of control and treatment groups. We are interested to compare the effectiveness of REST versus the QTP and a baseline manual approach with respect to CIFs and FPs.

H1 (CIFs of REST versus QTP) The effective null hypothesis is that $\mu_{cif}^{QTP} = \mu_{cif}^{REST}$, while the true null hypothesis is that $\mu_{cif}^{QTP} \geq \mu_{cif}^{REST}$. Conversely, the alternative hypothesis is that $\mu_{cif}^{QTP} < \mu_{cif}^{REST}$.

H2(FPs of REST versus QTP) The effective null hypothesis is that $\mu_{fp}^{QTP} = \mu_{fp}^{REST}$, while the true null hypothesis is that $\mu_{fp}^{QTP} \leq \mu_{fp}^{REST}$. Conversely, the alternative hypothesis is that $\mu_{fp}^{QTP} > \mu_{fp}^{REST}$.

H3(CIFs of REST versus Manual) The effective null hypothesis is that $\mu_{cif}^{M} = \mu_{cif}^{REST}$, while the true null hypothesis is that $\mu_{cif}^{M} \geq \mu_{cif}^{REST}$. Conversely, the alternative hypothesis is that $\mu_{cif}^{M} < \mu_{cif}^{REST}$.

H4(FPs of REST versus Manual) The effective null hypothesis is that $\mu_{fp}^{M} = \mu_{fp}^{REST}$, while the true null hypothesis is that $\mu_{fp}^{M} \leq \mu_{fp}^{REST}$. Conversely, the alternative hypothesis is that $\mu_{fp}^{M} > \mu_{fp}^{REST}$.

In addition, we want to know if the performance of the participants who have testing experience differs from those who do not have any testing experience. The categorical variables are testing experience and reported CIFs and FPs.

H5 (Independence of testing experience from CIFs) the testing categorical variable is independent from the variable CIF; the alternative is that they are associated.

H6 (Independence of testing experience from FPs) the testing categorical variable is independent from the variable FP; the alternative is that they are associated.

5.2 Subject GAPs and Test Scripts

We selected four open source subject GAPs based on the following criteria: easy-to-understand domain, limited size of GUI (less than 200 GUI objects), and two successive releases of GAPs with modified GUI objects. `Twister` (versions 2.0 and 3.0.5) is a real-time stock quote downloading programming environment that allows users to write programs that download stock quotes[7]. `mRemote` (versions 1.0 and 1.35) enables users to manage remote connections in a single place by supporting various protocols (e.g., SSH, Telnet, and HTTP/S)[8]. `University Directory` (versions 1.0 and 1.1) allows users to obtain data on different universities[9]. Finally, `Budget Tracker` (versions 1.06 and 2.1) is a program for tracking budget categories, budget planning for each month and keeping track of expenses[10]. Most of these applications are nontrivial, they are highly ranked in Sourceforge with the activity over 95%.

Next step was to obtain test scripts for subject GAPs. We obtained existing test scripts from sample script libraries that come with QTP. These scripts contained both GUI and non-GUI related code (e.g., setting values of environment variables and reading and manipulating directories contents). To make these scripts thorough, we generated statements that referenced GUI objects in the subject GAPs using QTP. Then we interspersed and replicated the generated statements throughout the test scripts. Information on subject GAPs and test scripts can be found in Table 1.

5.3 Methodology

We used a cross validation study design in a cohort of 45 participants who were randomly divided into three blocks labeled using different color labels. The study was sectioned in three experiments in which each block was given a different approach (manual, QTP, or REST) to apply to the subject GAPs. Thus each participants used each approach on different GAPs in the process of the case study. We randomly distributed participants so that each block has approximately the same number of participants with and without testing experience. Before the study we gave three one-hour tutorials on using each of these approaches on a GAP (mRemote) that was not used during the experiments thereby eliminating the knowledge of the GAP as a possible confounding factor.

[7] http://sourceforge.net/projects/itwister/

[8] http://sourceforge.net/projects/mremote/

[9] http://sourceforge.net/projects/universitydir/

[10] http://sourceforge.net/projects/budgettracker/

All participants are Accenture employees who work on consulting engagements as programmers and managers for different client companies. These participants have different backgrounds, experience, and belong to different groups of the total Accenture workforce of approximately 180,000 employees. Out of 45 participants (14 of whom are women), 23 had prior testing experience ranging from three weeks to ten years, and 18 participants reported prior experience with writing programs in scripting languages, including test scripts. Seven participants reported prior experience with QTP (which is used in this case study), six participants reported prior experience with other GUI testing tools. Twenty nine participants have bachelor degrees and ten have master degrees in different technical disciplines.

5.4 Normalizing Sources of Variations

Sources of variation are all things that could cause an observation to have a different value from another observation. We identify sources of variation as the prior experience of the participants with tools, GAPs, and test scripts used in this study, the amount of time they spend on learning how to use tools, and different computing environments which they use during the case study. The latter is extremely sensitive since some participants who use slow laptops with limited form-factor are likely to be less effective than other participants who use much better computing systems.

We design this experiment to drastically reduce the effects of covariates (i.e., nuisance factors) in order to normalize sources of variations. Using the cross-validation design we normalize variations to a certain degree since each participant uses all three approaches on different subject GAPs. We selected participants who had no prior knowledge of the subject GAPs. At the same time, subject GAPs belong to domains that are easy to understand, and these GAPs have similar complexity, so variations between them are negligent. However, different computing environments and prior experience of users with testing scripts and subject GAPs are major covariates.

We eliminated the effect of the computing environments by providing all participants with Dell Latitude D630 laptops with Intel Core 2 Duo Processor 2.4GHz with 4MB L2 Cache, 1Gb RAM, and 14.1" WXGA+ displays. Technical support at Accenture burnt the same standard Windows XP-based image on these laptops. We installed GAPs, scripts, and tools in a virtual machine that runs on top of the Microsoft Virtual PC thereby allowing participants to obtain a common environment of the entire experimental setup. This virtual machine can be downloaded from our website[11].

[11] http://www.markgrechanik.com/Rest/RestVM.zip

Subject Program	Size							Analysis, sec		Memory, Mb	Failures	
	Script LOC	Refd GUI, objs	Model, V_n Kb	Model V_{n+1} Kb	Add, GUI objs	Del GUI objs	APIs, No. of calls	Generate model	Fail Detect		CIFs	False Positives
Twister	492	54	30	38	81	12	42	1.7	5.2	105	16	19
mRemote	538	17	46	50	42	20	28	1.6	6.4	106	17	9
Univ Dir	920	36	33	35	35	9	29	1.4	6.2	105	13	2
Budget Tr	343	8	31	32	18	5	17	1.3	4.1	106	14	1

Table 1. Experimental results of applying REST to subject GAPs. Column `Size` contains seven subcolumns reporting the numbers of LOC in test scripts, the number of GUI objects that are referenced in the script, sizes of GUI models for versions V_n and V_{n+1}, numbers of added and deleted GUI objects, and the numbers of API calls that reference GUI objects. The column `Analysis` reports times to generate GUI models and to detect failures, followed by the column that shows the REST maximum memory consumption. The column `Failures` show the number of `CIFs` and `False Positives` reported by REST.

5.4.1 Tests and The Normality Assumption

We use one-way ANOVA, t-tests for paired two sample for means, and χ^2 to evaluate the hypotheses. These tests are based on an assumption that the population is normally distributed. The law of large numbers states that if the population sample is sufficiently large (between 30 to 50 participants), then the central limit theorem applies even if the population is not normally distributed [20, page 244-245]. Since we have 45 participants, the central limit theorem applies, and the above-mentioned tests have statistical significance.

5.5 Threats to Validity

A threat to the validity of this case study is that our subject GAPs have GUI screens that are of small to moderate size (a couple of hundreds of GUI objects). Increasing the size of GUIs of GAPs to thousands of GUI objects may lead to a nonlinear increase in the analysis time and space demand for REST. If it turns out to be the case, future work could focus on making REST scalable.

Since seven participants reported prior experience with QTP, this case study can be viewed as biased towards QTP versus the manual approach and REST. To reduce this bias, we provided a comprehensive tutorial on QTP for all participants of the study, and given the large number of participants we expect this bias to be negligent. However, the results of this study show that participants who had prior experience with QTP performed better with other competitive approaches. In addition, prior testing experience of the participants remains a source of variation.

We used nontrivial test scripts that contained code written by different test engineers, however, we cannot present metrics of how representative these scripts are of those used to test GAPs. In addition, we could not find any data that report percentage of coverage of GUI objects by test scripts. Finally, subject test scripts contain references to GUI objects that are located on one GUI screen per GAP. We relied on the fact that test personnel within Accenture is required to enforce modularity by writing one test script per GUI screen. Extending REST to support test scripts whose statements reference objects on different GUI screens is a routine exercise.

6 Results

6.1 Benchmark Evaluation

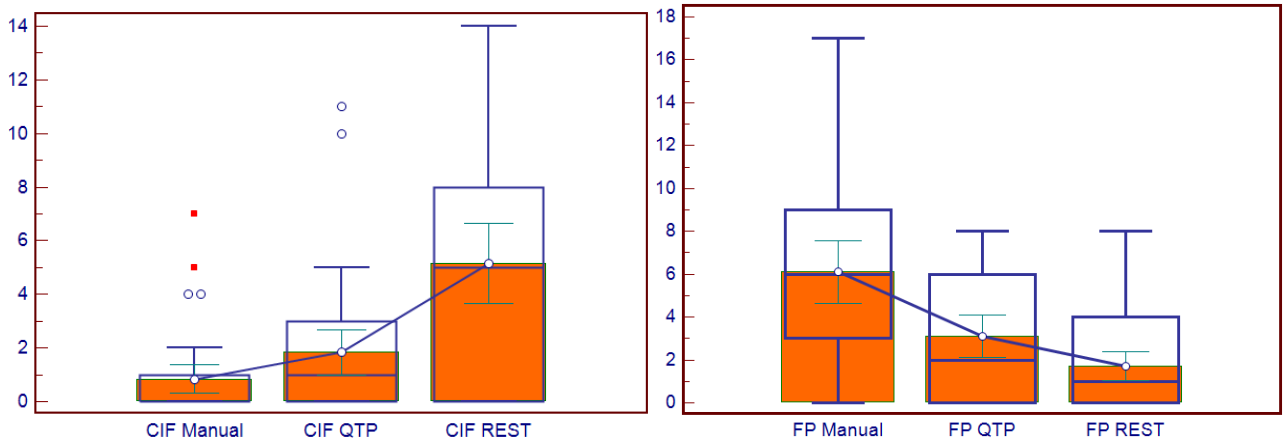
To measure characteristics of REST, we carried out experiments using Windows XP Pro that ran on a computer with Intel Pentium IV 3.2GHz CPU and 2GB of RAM. Experimental results of applying REST to the subject programs and scripts are shown in Table 1.

6.2 Case Study Results

In this section, we report the results of the case study and evaluate null hypotheses. We use one-way ANOVA, t-tests for paired two sample for means, and χ^2 to evaluate the hypotheses that we stated in Section 5.1.

6.2.1 Variables

A main independent variable is the approach (manual, QTP, REST) that participants use to find failures in test scripts. The other independent variable is participants' testing experience. Dependent variables are the numbers of correctly identified failures (CIFs) and false positives (FPs). We report these variables in this section. The effect of other variables (GAPs, test scripts, prior knowledge) is minimized by the design of this case study.



(a) Correctly Identified Failures (CIF).

(b) False Positive (FPs) by approach.

Figure 3. Statistical summary of the results of the case study for CIFs and FPs. The central box represents the values from the lower to upper quartile (25 to 75 percentile). The middle line represents the median. The thicker vertical line extends from the minimum to the maximum value, excluding outside and far out values, which are displayed as separate circles and small squares. The filled-out box represents the values from the minimum to the mean, and the thinner vertical line extends from the quarter below the mean to the quarter above the mean. An outside value is defined as a value that is smaller than the lower quartile minus 1.5 times the interquartile range, or larger than the upper quartile plus 1.5 times the interquartile range (inner fences). A far out value is defined as a value that is smaller than the lower quartile minus three times the interquartile range, or larger than the upper quartile plus three times the interquartile range (outer fences).

6.2.2 Testing the Null Hypothesis

We used ANOVA to evaluate the null hypothesis H_0 that the variation in an experiment is no greater than that due to normal variation of individuals' characteristics and error in their measurement. The results of ANOVA confirm that there are large differences between the groups for CIF with $F = 19.4 > F_{crit} = 3.08$ with $p \approx 5.9 \cdot 10^{-8}$ which is strongly statistically significant. The mean CIF for the manual approach is 0.84 with the variance 2.6, which is smaller than the mean CIF for QTP, 1.84 with the variance 6.6, which is smaller than the mean CIF for REST, 5.15 with the variance 20.7. Based on these results we reject the null hypothesis and we accept the alternative hypothesis H_1.

A statistical summary of the results of the case study for CIFs and FPs (median, quartiles, range and extreme values) are shown as box-and-whisker plots in Figure 3(a) and Figure 3(b) correspondingly with 95% confidence interval for the mean.

6.2.3 Comparing REST with QTP and Manual

To test the null hypothesis H1 and H2, we applied two t-tests for paired two sample for means, for CIFs and FPs for participants who used QTP and REST. The results of this test for CIFs and for FPs are shown in Table 2. The column `Samples` shows that 38 to 41 out of a total of 45 participants participated in all experiments (several participants missed one or two experiments). Based on these results we reject the null hypotheses H1 and H2, and we accept the alternative hypotheses that say that **participants who use REST report fewer false positives and correctly identify more failures in test scripts than those who use QTP**.

To test the null hypotheses H3 and H4, we applied two t-tests for paired two sample for means, for CIFs and FPs for participants who used the baseline manual approach and REST. The results of this test for CIFs and for FPs are shown in Table 2. Based on these results we reject the null hypotheses H3 and H4, and we accept the alternative hypotheses that say that **participants who use REST report fewer false positives and correctly identify more failures in test scripts than those who use a manual approach**.

6.2.4 Testing Relationships

We construct a contingency table to establish a relationship between CIFs and FPs for participants with and without testing experience as shown in Table 3. To test the null hypotheses H5 and H6 that the categorical variables CIFs and FPs are independent from the categorical variable testing experience, we apply two χ^2-tests, χ^2_{cif} and χ^2_{fp} for CIFs and FPs respectively. We obtain $\chi^2_{cif} = 21.3$ for $p < 0.0001$ and $\chi^2_{fp} = 11.5$ for $p < 0.0031$. The high values of χ^2 allow us to reject H5 and H6 in favor of the alternative hypotheses suggesting that **there is statistically strong relation-**

H	Var	Approach	Samples	Min	Max	Median	μ	σ^2	DF	C	p	T	T_{crit}
H1	CIF	QTP	41	0	11	1	1.76	6.24	40	0.23	$5.4 \cdot 10^{-5}$	4.52	2.02
		REST	41	0	14	4	5.0	20.15					
H2	CIF	Manual	38	0	7	0	0.84	2.62	37	0.08	$1.9 \cdot 10^{-6}$	5.65	1.87
		REST	38	0	14	5	5.16	20.73					
H3	FP	QTP	41	0	8	2	2.95	8.45	40	0.03	0.02	2.45	2.02
		REST	41	0	8	1	1.61	4.14					
H4	FP	Manual	40	0	17	6	6.2	18.8	39	0.14	$2.4 \cdot 10^{-7}$	6.24	2.02
		REST	40	0	8	1	1.73	4.31					

Table 2. Results of t-tests of hypotheses, H, for paired two sample for means for two-tail distribution, for dependent variable specified in the column Var (either CIF or FP) whose measurements are reported in the following columns. Extremal values, Median, Means, μ, variance, σ^2, degrees of freedom, DF, and the pearson correlation coefficient, C, are reported along with the results of the evaluation of the hypotheses, i.e., statistical significance, p, and the T statistics.

Test	CIFs			FPs		
Exp	Man	QTP	REST	Man	QTP	REST
Yes	29	30	115	128	85	33
No	3	40	81	104	33	32
Total	32	70	196	232	118	65

Table 3. Contingency table shows relationship between CIFs and FPs for participants with and without testing experience.

ship between testing experiences of participants and the numbers of reported CIFs and FPs. T-tests reveal that REST made a positive difference for inexperienced participants, while those with testing experience still performed better with REST than with QTP or the manual approach.

7 Related Work

GAPs present special challenges to regression testing because the input-output mapping does not remain constant across successive versions of the software [17][15]. Numerous techniques have been proposed to automate regression testing. These techniques usually rely on information obtained from the modifications made to the source code. Some of the popular regression testing techniques include analyzing the program's control-flow structure [2], analyzing changes in functions, types, variables, and macro definitions [7][13], using def-use chains [11], constructing procedure dependence graphs [8][19], and analyzing code and class hierarchy for object-oriented programs [14][18]. These techniques are not directly applicable to black-box GUI regression testing, since regression information is derived from changes made to the source code.

Closely related is a regression testing technique for GUIs (GUITAR), which repairs test cases that have become unusable for the modified GUIs [16]. A key difference is that REST works with arbitrary complex test scripts created by different test engineers while GUITAR maintain test cases that are generated by GUITAR itself. It is unclear if GUITAR can be extended to arbitary test scripts.

Proponents of model-based GUI-directed regression testing advocate building high-level models of GAPs before applying algorithms that construct test cases for evolved GAPs [21]. We consider REST complementary to this approach since it could enhance REST and improve its precision and usability by utilizing richer models.

8 Conclusion and Future Work

We offer a novel and effective approach called REST for maintaining and evolving test scripts so that they can test new versions of their respective GAPs. We built a tool to implement our approach, and the results of evaluation show that users find more failures and report fewer false positives in test scripts with our tool than with competitive approaches.

We consider REST as the first step towards developing different solutions for this big and pervasive problem. Specifically, we envision the following areas of research.

GUI Models. Constructing and utilizing richer GUI models has significant untapped potential for solving the problem of maintaining and evolving GUI-directed test scripts.

GUI Comparison Algorithms. It is important to develop algorithms that compare GUIs with a high degree of automation and precision.

Testing Platforms. Extending testing platforms with the capability to keep and process information about GUIs may enable researchers to create solutions in which these platform will "heal" broken test scripts.

Test Script Typechecking. A huge potential lies in developing type systems that enable typechecking test scripts with respect to GUI objects of GAPs.

Static and Dynamic Analyses. Few case study participants strongly suggested that REST should be integrated with QTP since the results of the REST static analysis can easily be verified by running test scripts under QTP. In addition, participants with testing experience reported that they used QTP during experiments with REST to verify some reported failures, and they thought it improved their productivity.

Acknowledgments

We warmly thank Sebastian Elbaum, Yannis Smaragdakis, Andrew Ko, and anonymous reviewers for their comments and suggestions.

References

[1] Section 508 of the Rehabilitation Act. *http://www.access-board.gov/508.htm*.

[2] T. Ball. On the limit of control flow analysis for regression test selection. In *Proceedings of ISSTA-98*, volume 23,2 of *ACM Software Engineering Notes*, pages 134–142, New York, Mar.2–5 1998.

[3] M.-C. Ballou. Worldwide distributed automated software quality tools: 2007-2011 forecast and 2006 vendor shares: Dominating quality. *IDC Report 210132*, 1, Dec. 2007.

[4] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, 2nd edition, 1990.

[5] S. Berner, R. Weber, and R. K. Keller. Observations and lessons learned from automated testing. In *ICSE '05*, pages 571–579, New York, NY, USA, 2005.

[6] A. Bertolino. Software testing research: Achievements, challenges, dreams. In *FOSE '07: 2007 Future of Software Engineering*, pages 85–103, Washington, DC, USA, 2007. IEEE Computer Society.

[7] J. Bible, G. Rothermel, and D. S. Rosenblum. A comparative study of coarse- and fine-grained safe regression test-selection techniques. *ACM Trans. Softw. Eng. Methodol.*, 10(2):149–183, 2001.

[8] D. Binkley. Reducing the cost of regression testing by semantics guided test case selection. In G. Caldiera and K. Bennett, editors, *ICSM*, pages 251–263, Washington, Oct. 1995.

[9] E. Dustin, J. Rashka, and J. Paul. *Automated Software Testing: Introduction, Management, and Performance*. Addison-Wesley, September 2004.

[10] M. Fewster and D. Graham. *Software Test Automation: Effective Use of Test Execution Tools*. Addison-Wesley, September 1999.

[11] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Transactions of Software Engineering and Methodology*, 2(3):270–285, July 1993.

[12] C. Kaner. Improving the maintainability of automated test suites. *Software QA*, 4(4), 1997.

[13] J.-M. Kim and A. A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *ICSE*, pages 119–129, 2002.

[14] D. C. Kung, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen. On regression testing of object-oriented programs. *The Journal of Systems and Software*, 32(1):21–31, Jan. 1996.

[15] A. M. Memon. *A Comprehensive Framework for Testing Graphical User Interfaces*. Ph.D. thesis, Department of Computer Science, University of Pittsburgh, July 2001.

[16] A. M. Memon and M. L. Soffa. Regression testing of GUIs. In *Proceedings of the ESEC and FSE-11*, pages 118–127, Sept. 2003.

[17] B. A. Myers. Why are human-computer interfaces difficult to design and implement? Technical report, Pittsburgh, PA, USA, 1993.

[18] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of java programs. In *OOPSLA*, pages 432–448, 2004.

[19] R. A. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. Test-suite augmentation for evolving software. In *ASE*, pages 218–227, 2008.

[20] R. M. Sirkin. *Statistics for the Social Sciences*. Sage Publications, third edition, August 2005.

[21] Q. Xie and A. M. Memon. Model-based testing of community-driven open-source GUI applications. In *ICSM*, pages 145–154, 2006.