Automatic Test Generation From GUI-Based Applications For Testing Web Services

Kevin M. Conroy, Mark Grechanik, Matthew Hellige, Edy S. Liongosari, and Qing Xie

Accenture Technology Labs

Chicago, IL 60601

Email: {kevin.m.conroy, mark.grechanik, matthew.hellige, edy.s.liongosari, qing.xie}@accenture.com

Abstract—Graphical User Interface (GUI) APplications (GAPs) are ubiquitous and provide various services. Since many GAPs are not designed to exchange information (i.e., interoperate), companies replace legacy GAPs with web services, that are designed to interoperate over the Internet. However, it is laborious and inefficient to create unit test cases to test the web services.

We propose a novel approach for generating tests for web services from legacy GAPs. This approach combines accessibility technologies for accessing and controlling GAPs in a uniform way with a visualization mechanism that enables nonprogrammers to generate unit test cases for web services by performing drag-and-drop operations on GUI elements of legacy GAPs. We built a tool based on our approach, and we used this tool to generate unit test cases from different GAPs. We believe that our approach is unique, and our evaluation suggests that our approach is effective and it can be used to generate test cases from nontrivial GAPs.

I. INTRODUCTION

Organizations custom-build *Graphical User Interface (GUI) APplications (GAPs)* and acquire them from third-party vendors to assist in business operations. GAPs are ubiquitous and provide various services, and they are often required to exchange information (i.e., to interoperate [4]) in order to improve the quality and to extend the coverage of their services. However, it is difficult to interoperate GAPs because many of them are not designed for easy data exchanges.

Web services are software components that interoperate over the Internet, and they gain widespread acceptance partly because of the business demand for applications to exchange information [7]. Unlike GAPs, using web services enables organizations to automate business processes by increasing the speed and effectiveness of information exchange. Naturally, different organizations are replacing legacy GAPs with web services, so that these organizations can speed up decision-making in business processes.

When legacy applications are rewritten, they often serve as references to test new applications, which we call *target applications*. Testing new target applications is expensive, taking up to 80% of the project cost [5]. *Application reference testing* is a form of regression testing where the target application is tested against the previous legacy application which is called the *reference application* [20] [21]. In this paper we consider a form of reference testing where target applications are tested using data from their corresponding reference applications. Since a subset of functionality is migrated from reference to target applications in most cases, it is beneficial to reuse test cases from these reference applications.

Unfortunately, many legacy applications do not have test cases with which these applications were tested: these test cases are lost or in undocumented formats. However, since legacy applications are used for many years, they accumulate a lot of data in their data storages. This data may be extracted from reference applications to test corresponding target applications. However, it is laborious and inefficient to understand the source code of the reference applications and schemas for their data storages in order to write programs to extract test cases for target applications. In general, applications rarely have automated oracles [6], [16]–[18]. Some companies spend on average close to 25% of project time to extract test data from reference applications [1].

Our goal is to extract data from legacy reference GAPs and use this data to generate test cases for target web services. Many legacy applications are GUI-based, and they expose data through their GUIs. GUI testing approaches use different techniques to control and manipulate GAPs in order to drive input data through GUI elements and switch between different GUI screens while GAPs perform background computations enroute. The intuition behind our approach is that we reverse the GUI testing process by replaying GAPs in order to make them expose stored data in their GUI elements. Our idea is to utilize GUI elements of the reference GAPs in order to extract test data and generate unit test cases for target applications.

We propose a *SysteM for Application Reference Testing (Smart)* for generating tests from reference GAPs and applying these tests to the corresponding target web services. Smart allows users to specify how they use reference GAPs, and then replay these GAPs for different input data using a prerecorded operational path, retrieving data from different GUI elements en route. Smart uses this retrieved data to generate unit test cases to test target web services.

Our contribution includes:
- a novel approach for generating unit test cases and test harnesses for web services from their reference GAPs;
- a generic and noninvasive mechanism that enables test personnel to extract test cases from GAPs using intuitive visual interfaces, without accessing the source code of reference GAPs and the schemas of their proprietary data storages;
- a tool that implements our approach.

We designed a tool based on our approach, and we used this tool to extract test cases from different reference GAPs.

We describe our experience with this tool, and measure and analyze its performance characteristics. The results suggest that our approach is efficient and effective. We believe that our approach is unique; we know of no other approaches whose contributions address the same aspects of our contribution.

II. A MOTIVATING EXAMPLE

A commonly used application in Accenture is `People Directory`, a Windows-based enterprise-level application that allows users to access and modify information on over $140,000$ employees. `People Directory`, whose GUI is shown in Figure 1 is a closed and monolithic GAP that does not have any programming interfaces and stores its data in a proprietary format. The application is designed only for user-level interactions.

Suppose that this reference GAP is replaced with a web service, that is a new target application. Using this web service will enable various applications to obtain access to information on Accenture employees. Our goal is to retrieve as much information about employees as possible so that we can generate unit test cases for the target web service.

One approach to accomplish this goal is to understand the format of the data storage for the GAP `People Directory`, relations between data elements in this storage, and write a program that extract this data and creates test cases. The other approach is to instrument the source code of this GAP, so that when it is run, the instrumented code saves the data with which this program operates in a test file. Even though both approaches offer certain benefits, they have one significant drawback - programmers should write or modify source code to generate unit test cases, and it is laborious and inefficient.

An efficient way to extract employee information is to enumerate all employees using the GUI of this application. Each employee is assigned a level, and there are twenty five levels. The user can search employees by selecting their levels in the combo box GUI element to which an arrow point with circled label `1`. By selecting a level in the combo box and pressing on the button labeled `Search`, a list of employees is loaded in the grid view to which an arrow point with circled label `2`. When a row in the grid view is selected, additional information about an employee is shown in the text fields to which an arrow point with circled label `3`. This information can be used to create unit test cases.

In order to get information on all employees, we need to repeat this procedure by selecting different levels of employees and iterating through the retrieved list of employees for each selected level. Since no programming-level knowledge is required to accomplish these operations, unit test cases can be generated by extracting data on employees from the GUI of this GAP with some automatic tool that mimics user-level actions. Creating this tool is one of the contributions of this paper.

III. THE PROBLEM STATEMENT

Our goal is to design an approach for generating unit test cases with a high-degree of automation. These test cases
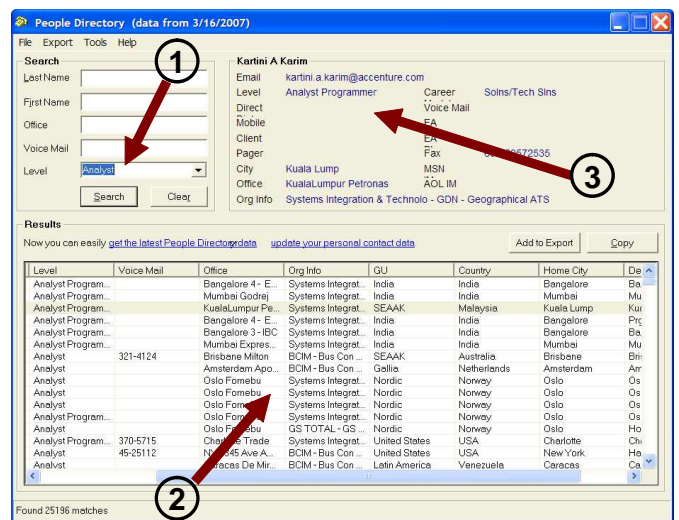

Fig. 1. The front-end of the application `People Directory`.

should be used to test web services which are the target applications of their respective reference GAPs.

Our approach should be non-invasive, specifically it should not require users to modify GAPs' source code or understand the structures of their data stores. Our approach should be easy to use, so that test personnel can generate unit test cases using only a basic knowledge of how to interact with GAPs to accomplish business tasks. Using visual drag-and-drop functionality is an ideal solution.

GAPs are written in many different languages and they are running on different platforms. There are many platform- or language-specific techniques for extracting data from GAPs. However, implementing these techniques in tools for different platforms and languages results in multiple versions of the source code for these tools, subsequently their increased cost, and eventually difficulties in maintaining and evolving different codebases. The wide applicability of our solution should be achieved by using an underlying technology for controlling and manipulating GAPs that is language-neutral and it is common to major computing platforms.

In general, the amount of information presented in GUI elements varies among different GAPs. If `D` is the total number of data items stored in the backend data store of some GAP, and `G` is the number of data items shown in GUI elements of this GAP, we can define the GUI coverage of the data for this GAP as `C=G/D` provided that these data items are useful for generating unit test cases. For example, the GAP `People Directory` has the `C`-ratio close to 100%. We assume that reference GAPs, for which our approach is applicable, have a significant `C`-ratio. In addition, we assume that all data that can be retrieved from GUI elements of the reference GAPs are useful for testing target applications or parts thereof.

IV. AN OVERVIEW OF OUR SOLUTION

The intuition behind our solution is that GAPs are replayed using special uniform techniques in order to make these GAPs

expose stored data in their GUI elements. This data is used to generate unit test cases and test harnesses to test target web services. We describe our solution using the architecture for Smart, which is shown in Figure 2 with block arrows marked with numbers indicating sequences of operations.

A central component of Smart is the Test Designer (or Designer). Its purpose is to allow Smart users to specify how to use GUI elements of reference GAPs to generate unit test cases and how to map these GUI elements to target web services, specifically to exposed methods and their parameters. These mappings are used to generate test harnesses that use generated unit test cases.

Using the Smart tool, the user sends a request to the Proxy to load a *Reference APplication (RAP)* `(1)`. A Proxy is a generic program that receives requests, extracts data from GAPs in response to these requests, and sends the extracted data back to requesters. Proxies use the accessibility layer to control and manipulate RAPs uniformly by providing programmatic access to their GUI elements `(2)`. Thus this accessibility layer can be viewed as a virtual machine that can be used to control and manipulate GAPs. We provide background on accessibility technologies in Section V-B.

From a tester's point of view, GUI elements have up to four functions: action producers, input data acceptors, output data retrievers, and state checkpoints. Action producers enable RAPs to switch to different states. The GUI element `Button` is an example of an action producer; clicking on a button switches a RAP to a different state. Some GUI elements may have all four functions, for example, a combo box may be a state checkpoint, may contain output data, may accept input data, and may produce some action when the user makes a selection.

Input data acceptors are GUI elements that take data from users (e.g., text boxes). Output data retrievers are GUI elements that contain data (e.g, list views or text boxes). These elements serve as data suppliers for generating unit test cases. Finally, state checkpoint elements can be any GUI elements that are required to exist on screens in order for RAPs to function correctly. Since initializing GUI elements

is asynchronous, it is important to make sure that key GUI elements are initialized and ready for use before accessing them. Clearly, any output GUI element is also a checkpoint element since Smart cannot retrieve any data from it unless it is initialized. In case of the RAP `People Directory`, Smart cannot proceed until the grid view element is initialized and ready to use.

The front end of the Designer has two views: *Reference Application View (RAV)* and *Target Application View (TAV)*. RAV displays information on RAP, its GUI elements and their programming representations. TAV displays the structures of target applications, specifically web services that come from *Web Service Description Language (WSDL)* files.

After dragging-and-dropping required GUI elements from the RAP onto the RAV and defining the names and functions of these elements in RAV, the user generates the Replay Script `(3)`. The Replay Script is a Java program that contains code for controlling and manipulating RAP to switch it from state to state. When executed, the Replay Script sends a sequence of commands to the Proxy in order to control and manipulate the RAP to extract data from it `(4)`. This is a primary way to obtain data for generating unit test cases. The Replay Script stores retrieved data unit as test cases `(5)`.

TAV enables testers to map test cases to methods of the *Target APplication (TAP)*. Smart reads in a description of the target application in a WSDL file `(7)`, and this description contains exposed interfaces, their methods, and parameters of these methods. With Smart, test personnel can specify mappings between input parameters of the methods of the target applications and GUI elements `(8)` thereby assigning data items from test cases as input data to methods of the target web service `(6)`. Analogously, users map return values of these methods to test oracles.

Once testers define all mappings, the Designer outputs a test harness `(9)` that contains the driver for running tests on the TAP. When this harness is run, it uses test cases `(10)` to test the target web service `(11)`. Testing is done by running Test Harness that invokes methods of the target web service, passes test data as the parameters to these methods, and uses return values to compare them with the generated test oracles. We show an example of the code for test harnesses and discuss it in Section VI-D.

V. Techniques

In this section, we present core ideas behind our approach, provide a background on accessibility technologies, and describe the structure of GAPs. Using accessibility technologies to work on the structure of GAPs constitutes the set of techniques that we use to implement our solution.

A. Core Ideas

Our main idea is to extract test cases from reference GAPs by mimicking a human-driven procedure of interacting with GAPs. This procedure can be described as follows. After a GAP is started and initial screens appear, users may read data from and enter data into some GUI elements. Then users
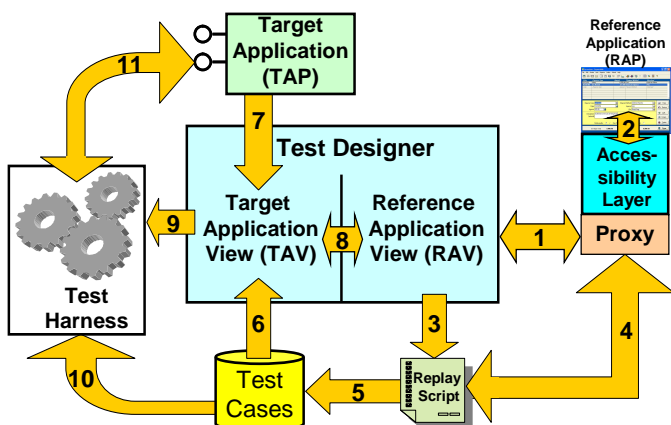

Fig. 2.   The architecture of Smart.

initiate transitions by causing some actions (e.g., select a menu item or click on a button). As a result of these actions, GAPs perform computations and show GUI screens that may be different from the previous ones. Again, users read and enter some data and perform actions. This cycle continues until users quit these applications.

In order for automatic data extraction tools to mimic this procedure, they should be able to access GUI elements of GAPs programmatically. A core idea of our solution is that GAPs and their GUI elements are programming objects whose values can be set and retrieved and whose methods are associated with actions that users perform on these elements. For example, a combo box element displays a number of items, and selecting an item in the combo box invokes a method that performs some computation. To control this combo box programmatically, a service should invoke methods and set values of the fields of a programming object that represents this combo box which is hosted in a GUI of some GAP.

A key technique is to use GAPs as programming objects and GUI elements of these GAPs as fields of these objects, and to perform actions on these GUI elements by invoking methods on the objects that represent these GAPs. Unfortunately, external tools cannot access and manipulate GUI elements of GAPs as pure programming objects because GUI elements only support user-level interactions. Accessibility technologies overcome this limitation by exposing a special interface whose methods can be invoked and the values of whose fields can be set and retrieved thereby controlling GUI elements that have this interface. We give an overview of the accessibility technologies in the next Section V-B.

B. Accessibility Technologies

Accessibility technologies provide different aids to disabled computer users [3]. Specific aids include screen readers for the visually impaired, visual indicators or captions for users with hearing loss, and software to compensate for motion disabilities. Most computing platforms include accessibility technologies since electronic and information technology products and services are required to meet the *Electronic and Information Accessibility Standards* [3]. For example, *Microsoft Active Accessibility (MSAA)* technology is designed to improve the way accessibility aids work with applications running on Windows, and *Sun Microsystems Accessibility* technology assists disabled users who run software on top of *Java Virtual Machine (JVM)*. Accessibility technologies are incorporated into these and other computing platforms as well as libraries and applications in order to expose information about user interface elements.

Accessibility technologies provide a wealth of sophisticated services required to retrieve attributes of GUI elements, set and retrieve their values, and generate and intercept different events. In this paper, we use MSAA for Windows, however, using a different accessibility technology will yield similar results. Even though there is no standard for accessibility *Application Programming Interface (API)* calls, different technologies offer similar API calls, suggesting slow convergence

towards a common programming standard for accessibility technologies.

The main idea of most implementations of accessibility technologies is that GUI elements expose a well-known interface that exports methods for accessing and manipulating the properties and the behavior of these elements. For example, a Windows GUI element should implement the `IAccessible` interface in order to be accessed and controlled using the MSAA API calls. Programmers may write code to access and control GUI elements of GAPs as if these elements were standard programming objects. Accessing GUI elements is done by navigating the structure of GAPs that is described in the next section.

Using accessibility technologies, programmers can also register callback functions (or hooks) for different events produced by GUI elements thereby obtaining timely information about states of the GUI elements of the GAPs. For example, if a GUI element receives an incorrect input and the GAP shows an error message dialog informing the user about the mistake, then a previously registered callback can intercept this event signaling that the message dialog is being created, dismiss it, and send an "illegal input" message to the tool that controls the GAP.

Since accessibility layers are supported by their respective vendors and hooks are technical instruments which are parts of accessibility layers, using hooks is legitimate and accepted to control and manipulate GAPs. In addition, writing and using hooks is easy since programmers use high-level accessibility API calls, and they do not have to deal with the complexity of different low-level binary rewriting techniques.

C. The Structure of GAPs

In event-based windowing systems (e.g., Windows), each GAP has a main window (which may be invisible), which is associated with the event processing loop. Closing this window causes the application to exit by sending the `DestroyWindow` event to the loop. The main window contains other GUI elements of the GAP. A GAP can be represented as a tree, where nodes are GAP GUI elements and edges specify that children elements are contained inside their parents. The root of the tree is the main window, the nodes are container elements, and the leaves of the tree are basic elements.

Each GUI element is assigned a category (class) that describes its functionality. In Windows, a basic class of all GUI elements is the class `window`. Some GUI elements serve as containers for other elements, for example, dialog windows, while basic elements (e.g., buttons and edit boxes) cannot contain other elements and are designed to perform some basic functions. This hierarchical containment is reflected in the core idea of the object-oriented design of our solution where classes representing container windows have fields that are instances of the classes that represent contained GUI elements.

VI. IMPLEMENTATION

In this section, we describe the low-level details of the system implementation of Smart, which is based on our

approach. We describe the components of the architecture for Smart and delve into our implementation of the Designer. Finally, we give a high-level overview of how Designer is used to generate unit test cases from reference GAPs and use these test cases to test target web services.

A. Representing GUIs Programmatically In Smart

Recall our idea that GUI elements are programming objects whose values can be set and retrieved and whose methods can be invoked. As such, GUI elements can be represented as classes residing inside web services. Since GUI screens consist of different GUI elements, these screens are also GUI elements (i.e., windows) and can also be represented as classes whose fields are instances of the classes representing these GUI elements. Finally, GAPs consist of different GUI screens, and GAPs can be represented as classes whose fields are instances of the classes representing constituent GUI screens. We view GAPs as state machines, and web services control GAPs by transitioning them to different states using programming objects that represent these GAPs.

To summarize, GAPs are state machines whose states are defined as collections of GUI elements, their properties (e.g., style, read-only status, etc.), and their values. When users perform actions they change the state of the GAP. In a new state, GUI elements may remain the same, but their values and some of their properties change.

A schematic state machine describing transitions between screens of a GAP is shown in Figure 3. The states of the GAP are depicted using the screens that are represented as trees of GUI elements, and the transitions are shown with arrows labelled with actions that users perform on GUI elements. The Kleene star over the label `action` stands for zero or more actions.

We distinguish between intermediate and final states. Consider clicking on a link in a web-based application. After loading a page that displays a progress GUI element, the browser is redirected to the destination page. Clearly, the page with the progress GUI is an intermediate state of the GAP, and the users are interested in the destination page, which is a final state. Users distinguish intermediate states from final states by visually inspecting GUI screens to check to see if required GUI elements are shown. In order to perform this function automatically, a method of a web service should inspect GAPs to analyze their structures and their GUI elements to detect intermediate and final states. This analysis is done by traversing GUI trees and comparing them to the trees that are recorded by the Designer when users perform operations on GAPs as described in Section VI-C.
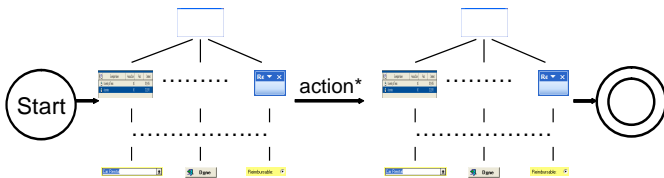

Fig. 3. A schematic state machine describing transitions between screens of a GAP, where these screens are represented as trees of GUI elements.

B. The Designer Implementation

The Designer is the central component of the SMART since it enables users to generate tests from GAPs. Its implementation is based on representing GAPs as state machines.

The Designer takes inputs describing the states of the GAPs and generates classes whose methods control GAPs by setting and getting values of their GUI elements and causing actions that enable GAPs to switch to different states. When the user switches the GAP to some state, the Designer records this state by traversing the GUI tree of the GAP post-order using the accessibility technology. For each node of the tree (i.e., a GUI element), the Designer emits code for classes are linked to these GUI elements, and these classes contain methods for setting and getting values and performing actions on these elements.

C. A Birds-Eye View

We present a birds-eye view of how Designer is used by giving an example of generating tests for a target web service whose reference GAP is `People Directory`, which we described in Section II. We have built a tool called Smart Designer (or simply Designer) that enables users to extract data from GAPs and generate tests using our approach.

The front end of Designer is shown in Figure 4. The tab `SMART Class Explorer` displays information on the GAP, its screens and GUI elements. The tab `Properties` shows properties of GUI elements displayed in the tab `SMART Class Explorer`. The tab labeled `Screen1` shows classes that programmatically represent GUI elements of the reference GAP. There can be multiple tabs labeled `Screen` for different reference GAPs. Finally, the tab labeled `Service` displays a visual representation of the WSDL for a target web service. The data displayed in the tabs describe the reference GAP `People Directory` and its target web service.

Using the front end of Smart, the user specifies GUI elements of the reference GAP that hold values that can be used to generate tests cases. These elements include GUI elements that should receive input data and sequences of actions that the user should perform in order to get access to GUI elements containing data that should be extracted for future tests.

The user interacts with the GAP `People Directory` by selecting a level from the combo box GUI element to which an arrow with circled label `1` points in Figure 1, to get a list of employees. For example, the user selects the level `Analyst` on the GUI screen, and then clicks on the button labeled `Search`. These actions cause the GAP to switch to the screen on which the grid view is loaded with the information on employees with the corresponding work level.

The purpose of interacting with the GAP is to allow the Designer to record the structures of the screens and user actions on the GAP and then transcode these actions into programming instructions that the Replay Script will execute against the GAP in order to generate unit test cases. These instructions are encoded in the implementation of the class shown in the tab `Screen1` of the Designer as the class `Screen1`. By instantiating this class and invoking its methods
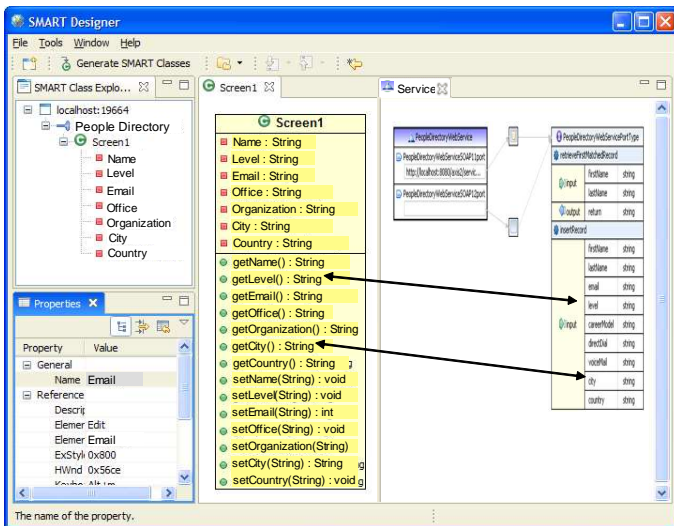
Fig. 4. The front-end of the Designer.

in the generated test script, these instructions control and manipulate the GAP leading it to different states in which GUI elements contain data that can be extracted for generating unit test cases.

When recording the sequence of screens, the Designer obtains information about the structure of the GUI and all properties of individual elements using the accessibility-enabled interfaces. Designer uses this information to generate class `Screen1` whose diagramic representation is shown in the tab `Screen1`.

At the design time, the user should specify functions of GUI elements, specifically what GUI elements receive values or serve as inputs and what elements produce output results. To do that, the user moves the cursor over the GUI element, and the Designer uses the accessibility API calls to obtain information about this element. To confirm the selection, a frame is drawn around the element with the tooltip window displaying the information about the selected element. Then, the user clicks the mouse button and drags this element (or rather its image) onto the tab `Screen1` of the Designer. After releasing the mouse button, the dragged element is dropped onto the tab area, resulting in new members added to the class `Screen1`.

The Designer displays a property dialog box prompting the user to specify the function of the dropped element (i.e., input, output, action, or checkpoint), its name, and some other properties. Once the user has dragged-and-dropped all GUI elements, set properties of their corresponding programming objects, and loaded the description of the target web service from its WSDL file, it is time to connect these elements and properties of the loaded service with arrows that specify the how data that are extracted from GUI elements are mapped to input parameters of the methods of the web service. For example, by drawing an arrow between the method `getLevel()` of the reference GAP and the input parameter `Level` of

the web service, the user specifies that the data from the corresponding GUI element will be used in the corresponding input parameter of the method of the web service. While it is possible to specify how to transform the data, we do not consider these modifications in this paper for simplicity.

Once all mappings are created, the Replay Script and the Test Harness can be generated simply by clicking on the button `Generate SMART Classes`. The Designer uses the information captured for each screen and input elements to generate the Java code and unit test cases. We give a fragment of the generated code for test harness in the next section.

D. Generating Test Harness

An example of a fragment of the generated code for test harness is shown in Figure 5. This fragment contains the code for the function `test()` that is extracted from a JUnit-based class whose purpose is to test a target web service for the reference GAP `People Directory`.

```
 1: @Test public void test () {
 2:   PDServiceStub stub =
 3:     new PDServiceStub(ENDPOINT);
 4:   PDServiceStub.InsertEmployee req1 =
 5:     new PDServiceStub.InsertEmployee();
 6:   req1.setName(_Employee_Name);
 7:   req1.setLevel(_Employee_Level);
 8:   req1.setEmail(_Employee_Email);
 9:   // ...
10:   PDServiceStub.InsertEmployeeResponse
11:     res1=stub.insertEmployee(req1);
12:   PDServiceStub.LookupEmployee
13:     req2=new PDServiceStub.LookupEmpl();
14:   req2.setEmail(_Employee_Email);
15:   PDServiceStub.LookupEmplResponse
16:     res2 = stub.LookupEmpl(req2);
17:   assertEquals("Employee Name",
18:     _Employee_Name, res2.getName());
19:   assertEquals("Employee Level",
20:     _Employee_Level, res2.getLevel());
21:   // ...
22: }
```

Fig. 5. Example of a fragment of the generated code for test harness.

The target web service exports methods `InsertEmployee` and `LookupEmpl`, to insert and locate employees correspondingly in some persistent storage of the service. Lines `2-3` show how the service is instantiated and the reference `stub` to this service is created. In lines `4-5` a reference to the method `InsertEmployee` is created, and in lines `6-9` the values for parameters to this methods are set. Lines `10-11` show that this method is called, inserting employee information into the persistent storage.

Once the method `InsertEmployee` is completed, the correct behavior of the service should be verified by calling the method `LookupEmpl`, which should return the inserted

employee data from the persistent storage. Lines `12-13` show that the reference `req2` to this method is created, and in line `14` the value of an employee email address for the parameter to this method is set. `LookupEmpl` is invoked in line `15`, and in lines `16-21` returned values for the fields of employee information are compared with the expected values, i.e., test oracles are asserted.

VII. Experimental Evaluation

In this section we describe the methodology and provide the results of experimental evaluation of our approach. We describe our feasibility study in which we extract data from reference GAPs and use this data to test corresponding target web services. In our evaluation, we target performance characteristics of the Smart approach since it is important to show that GUI overhead is negligible and will not affect the scalability of Smart.

A. Feasibility Study

In this feasibility study we describe data extraction experiments first, and then we show how we used the extracted data to test web services.

1) Extracting Data From Reference GAPs: To demonstate our approach, we extracted test data from four subject applications: the Accenture application `People Directory` (PD), an open source application `University Data` (UD), and two commercial GAPs such as `Quicken Expensable 98` (QE) and `ProVenture Invoices and Estimates` (PIE). QE and PIE are closed and monolithic GAPs that run on Windows. PIE allows users to create and print invoices, estimates, and statements, and to track customer payments and unpaid invoices. QE allows users to enter and retrieve expenses. UD allows users to select any of the fifty states and obtain a list of public universities for the selected state. By selecting a university a user can obtain information on it. We discussed PD in Section II.

Our experience confirms the benefits of our approach. We extracted data for test cases from the subject GAPs without writing any additional code, modifying the applications, or accessing their proprietary data stores. Since subject applications are closed and monolithic GAPs that neither expose any programming interfaces nor publish their data in known formats, we generated unit test cases from these GAPs using basic information about how users interact with them to accomplish tasks. We carried out experiments using Windows XP Pro that ran on a computer with Intel Pentium IV 3.2GHz CPU and 2GB of RAM.

The results of this experiment are shown in Table I. The first column shows the name of the GAP and its comma-separated state identifier that is the sequence number of the screen. The next three columns show the number of visible and invisible GUI elements and the number of GUI elements including their parents in the hierarchy that are used as parameters or action targets in web services. The fifth column shows the size of the XML generated by the Designer to describe the given state of the GAP. For example, the GAP PD has the biggest amount of XML data in state 2 since PD retrieves a list of employees in this state. Finally, the last column shows the time taken to generate and parse the state XML.

Even though XML is not inherent in Smart, there is a possible problem with the size of the XML data especially considering that Replay Scripts and GAPs that they control may be located on different computers. Sending large amounts of XML data may saturate the network traffic. A way to address this problem is to use XML data compression, or optimize the system to reduce the amount of data sent over the network. This is a subject of our future work.

In our feasibility study, we compared an effort required to extract data from PD using Smart with the programming effort to analyze and write a parser for data files of PD. It took us approximately two hours to write a program that parses data files and extracts employee information. Compared to that, it took us less than ten minutes to generate unit test cases from the GAP using our approach.

2) Testing Target Web Services With The Extracted Data: We used the extracted data to test web services for PD and UD. These services expose two methods. One method insert test data into a persistent data storage, and the other method searches for data using a key specifying a part of the test data. For example, in case of PD the search key was the email address of an employee.

Our experience showed that using data extracted from reference GAPs allows users to start testing web services quickly, without any delay normally required to create test cases. In addition, since test harnesses are generated automatically using Smart, it saves time for test personnel that they would otherwise spend writing source code for these harnesses.

B. Performance Evaluation

Naturally, extracting data programmatically is more efficient than doing it through GUI elements of GAPs. The additional overhead cost, `OC`, consists of the GAP startup time, the initialization time for the internal structures representing GUI elements, screen switching time, and communicating time between Proxies and GAPs.

The goal of the performance experiment is to evaluate how much performance penalty Smart as a GAP-based data

GAP Name,	Number of GUI Elements			State XML,	Time,
State No	Visible	Invisible	Used	Bytes	Sec
QE, State 1	152	23	16	46,345	0.3
QE, State 2	152	23	16	47,822	0.3
QE, State 3	193	36	28	73,339	0.4
QE, State 4	193	36	28	75,204	0.4
QE, State 5	152	23	12	46,838	0.3
PIE, State 1	226	51	35	96,018	0.5
PIE, State 2	207	46	41	106,225	0.5
PD, State 1	28	16	23	22,382	0.3
PD, State 2	28	16	23	2,386,299	2.7
UD, State 1	47	16	39	28,398	0.3
UD, State 2	47	16	39	53,275	0.5

TABLE I

Experimental results for using Smart on subject GAPs.

extraction approach incurs versus a pure programmatic one. In addition, we show that generating test cases in batches results in a significant performance gain.

1) Performance Stress Test: We designed the performance test to measure the reliability and sustainability of data processing throughput of our implementation of the Smart data extraction system. The test script simulated users who extracted employee data from PD. For each employee in PD, the process included extracting information about this employee and creating a corresponding test date entry, with the last step in this process being either the return to the initial screen of the GAP, or the termination of the GAPs. The test was run at a user load for duration of `24` hours in order to minimize various effects of other applications and services running on the same computer over the extended period of time.

We repeated this test three times for PD. The first test was run with a data extracting component implemented as purely programmatic component with no GUI interfaces used to to extract data. The component parsed files containing employee information and created test cases from it. The second test was run against PD as a GAP whose screens were reused by returning to original screens, not restarting the GAP. The last test was run against the GAP PD which was restarted after each transaction. We report an average time per transaction for each test.

Experimental results from evaluating how much performance penalty these GAP-based data transfers incur versus pure programmatic ones are shown in Figure 6. The vertical axis shows the average time in seconds per data exchange transaction, and the bars correspond to the tests. The fastest transaction takes on average `0.5` seconds when no GAPs are used, that is the data is extracted purely programmatically from the data files of PD. The performance drops when GUI elements of PD are used to encapsulate the functionality required to extract the data. The average time per transaction increases to `1.4` from `0.5`, which is 180% increase. The difference between these average transaction times is `0.9` second, which we attribute to the overhead of the GUI computations.

The situation worsens for the third test when the GAP is required to restart every time the data exchange is performed. The overhead associated with restarting of the GAP increases the average time per transaction to `2.8` seconds. While the overhead in percentage looks bad, its absolute values are small, especially when considering that generating unit test cases from reference GAPs is done once for a given reference GAP. This overhead is negligible when comparing with the development effort required to write programs to extract data or to do it manually.

2) Batch Data Extractions: Consider a situation when unit test cases for a target web service are generated from two or more reference GAPs. For example, employee information extracted from the application PD can be used to locate expenses in the application QE for each employee. There are two ways to accomplish this task. One way is to get information about each employee from PD, one at a time,

and then use this information for QE to check to see if this employee has expenses. If expense information is present for this employee, then a test case is generated for the target application. Otherwise, the employee information is discarded, and the procedure is repeated for the next employee in PD.

The other way is to extract all available information for all employees from PD and use this information to locate expenses in QE. We call it batch data extraction. Our goal is to find out which way is more effective. Smart enables users to specify batch data exchanges between GAPs when extracting data for generating tests. Sending messages between GAPs is delayed in batch message transfers until a batch message containing many smaller messages is formed and transmitted at once.

The combined overhead associated with sending many small messages is higher than the overhead of sending one larger message that contains these small messages. Sending separate small messages involves adding control information in headers that specify additional information for message transfer, invoking functions that create and parse these messages, and often having to perform additional extraction operations on GUIs.

Extracting all data from GUI elements and sending them to the destination in one single message enables users to amortize a one-time overhead over many data items in the message. In addition, since the network latency time is one of the contributors to the overhead, sending fewer messages may result in the reduction of this overhead. Specifically, sending one large message instead of many small messages reduces the communication overhead between GAPs and Proxies. Also, when sending many messages, next message is usually send when the receipt of the previous message is acknowledged by a special response message. Sending and waiting for these acknowledgement messages adds unnecessary computation and communication overhead, which can be significantly reduced using batch transfers.

The goal our experiment is to show that exchanging messages in batches yields better performance of Smart. The graph showing the dependency of transfer time per item from the number of items transferred between GAPs is shown in Figure 7. The horizontal axis shows the number of data items
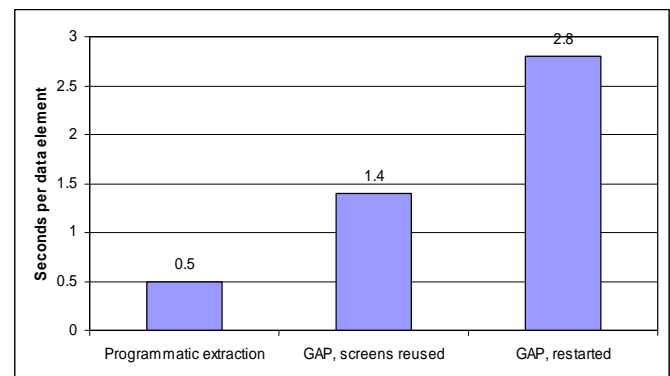

Fig. 6. Data extraction transaction throughput for Smart.

transferred per transaction. An average size of the data item is approximately `360` bytes. The vertical axis shows the time it takes to extract an item from PD, find expense information for this data item (i.e., employee) in QE, and extract the expense data from QE.

For a single item it takes approximately `3.8` seconds to transfer a data item considering the GAP computational overhead. As data items are transferred in batches, the amount of time it takes to transfer an item drops approximately `7.6` times to close to `0.5` second as the number of the items in a batch grows to `500`. Correspondingly, the size of the XML message containing these data items grows from `1.5Kb` to `130Kb`. However, transferring `1,000` items instead of `500` reduces the transfer time per item by only `1.7` times. Our explanation is that it takes longer to create and parse large messages as well as to transfer them, and this additional overhead dwarfs the reduction of the extraction and transfer time per message that contains test cases. Still, the our experiment shows that it is possible to gain significant performance by performing batch data extractions from GAPs.

C. Limitations

In general, Smart may not work well with GAPs whose GUIs are dynamically created. However, the number of such GAPs is small, and most GUIs are stable and may have small changes between releases, which happen infrequently. Since Smart is used once for a reference GAP to generate test cases, subsequent changes to the GUI of this GAPdo not affect the functionality of Smart.

When attempting to run two instances of Smart in parallel, we found that multiple instances of QE are prevented by design from running on the same computer. This problem can be solved by installing a copy of QE on a different computer and extending the Smart architecture to work in a distributed environment.
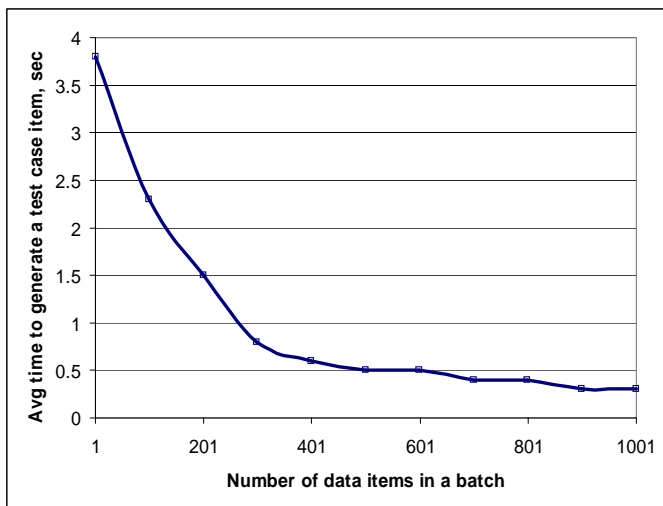

Fig. 7. Dependency of the average time that takes to generate an item of a test case from the number of data items extracted from GAPs using the batch extraction technique.

In general, managing many instances of the same GAP is difficult. For example, when running web-based applications, they open many popup windows. These windows and processes that control them are not linked explicitly to the application that opened them. Thus, when two or more web-based applications ran on the same computer simultaneously, data from these applications may be mixed. We are currently working on solving this problem.

VIII. RELATED WORK

Related work falls into the following three categories: *techniques to control and manipulate GAPs programmatically*, *reference testing*, and *techniques for testing web services*.

Techniques to control and manipulate GAPs programmatically: When it comes to extracting information from GAPs and their GUI elements, the term *screen-scraping* summarily describes various techniques for automating user interfaces [14] [2]. Macro recorders use this technique by recording the users mouse movements and keystrokes, then playing them back by inserting simulated mouse and keyboard events in the system queue [13]. Screen-scraping has multiple advantages over binary rewriting and code patching techniques as it does not require any modifications to the underlying computing platforms or applications' source and executable code. Our approach uses some aspects of screen-scraping, however, it differs from other screen-scraping techniques since it does not depend on parsing a scripting language that describes the GUI, and therefore it is more generic and uniform.

Reference testing: One of the most difficult challenges in software testing is to determine whether the application under test behaves correctly during its execution with regard to a test case. A reference, serves as the "specification" for functional correctness, is used to tackle this problem. In reference testing, the target application (application under test) is tested against the reference application (specification) for each test case [20], [21]. Actual outputs are recorded during the execution of the reference application. The recorded outputs are later used as expected output for the target application. Differential testing [10], [12] is a form of reference testing. It has been applied in large-scale software systems and shown effectiveness. However, in [10], [12], the test cases generated and feeded to both target and reference application are the same (or with a high degree of similarity). For example, both target and reference applications in [12] are C-compilers, but with different implementations; they expect the same C-program as the test case. While in SMART, the reference application is a GAP and a test case is a sequence of events; and the target application is web service with the test case as a service request sent to service provider.

Techniques for tseting web services: Most work on testing web services is primarily model-based verification. Narayanan et al [15] exploited DAML-S ontology to provide an semantic markup of web services and test web services by simulating their execution under different input conditions. Foster et al [8] presented a formal approach to modeling and verifying

the compositions of web services workflows using the finite satte processes notation.

WSDL has been used to generate test cases by many commerical tools, such as SOATest (http://www.parasoft.com), and LISA (http://www.itko.com). However, the expected output has to be manually specified for comparison. Serveral researchers use WSDL to generate test cases as well. Sneed and Huang [19] developed the WSDLTest tool for generating random requests for WSDL schemas with manually specified preconditions, and comparing the response against the manually specified postconditions. Martin et al [11] presented a framework that takes WSDL as input and performs robustness testing of web services automatically. However, the framework does not check the functional correctness of the service. Similarly, Fu et al [9] performed robustness testing of web services on the service provider side using fault injection techniques.

IX. CONCLUSION

We proposed a novel generic approach for generating tests for testing web services from their reference legacy GAPs. This approach combines a nonstandard use of accessibility technologies for accessing and controlling GAPs in a uniform way with a visualization mechanism that enables test personnel to generate unit test cases by performing point-and-click, drag-and-drop operations.

We designed a tool based on our approach, and we used this tool to extract test cases from reference GAPs. Our experiments suggest that our approach is efficient and effective. Choosing Smart to extract data for generating tests from reference GAPs saves a lot of time that would otherwise be spent to write programs that perform data extractions.

ACKNOWLEDGMENTS

REFERENCES

[1] Private conversations with Accenture project leaders working on application renewal projects.

[2] Screen-scraping entry in Wikipedia. *http://en.wikipedia.org/wiki/Screen_scraping*.

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

[4] *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*. Institute of Electrical and Electronics Engineers, January 1991.

[5] G. Dedene and J.-P. D. Vreese. Realities of off-shore reengineering. *IEEE Software*, 12(1):35–45, 1995.

[6] L. K. Dillon and Q. Yu. Oracles for checking temporal properties of concurrent systems. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 140–153, Dec. 1994.

[7] C. Ferris and J. A. Farrell. What are web services? *Commun. ACM*, 46(6):31, 2003.

[8] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of web service compositions. In *ASE '03: Proceedings of the Eighteenth IEEE International Conference on Automated Software Engineering (ASE)*, pages 152–163, 2003.

[9] C. Fu, B. G. Ryder, A. Milanova, and D. Wonnacott. Testing of java web services for robustness. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 23–34, New York, NY, USA, 2004. ACM Press.

[10] A. Groce, G. J. Holzmann, and R. Joshi. Randomized differential testing as a prelude to formal verification. In *ICSE*, pages 621–631, 2007.

[11] E. Martin, S. Basu, and T. Xie. Automated robustness testing of web services. In *Proceedings of the 4th International Workshop on SOA And Web Services Best Practices (SOAWS 2006)*, October 2006.

[12] W. M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[13] R. C. Miller. End-user programming for web users. In *End User Development Workshop, Conference on Human Factors in Computer Systems*, 2003.

[14] B. A. Myers. User interface software technology. *ACM Comput. Surv.*, 28(1):189–191, 1996.

[15] S. Narayanan and S. A. McIlraith. Simulation, verification and automated composition of web services. In *WWW '02: Proceedings of the 11th international conference on World Wide Web*, pages 77–88, New York, NY, USA, 2002. ACM Press.

[16] D. Peters and D. L. Parnas. Generating a test oracle from program documentation: work in progress. In *ISSTA '94: Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, pages 58–65, New York, NY, USA, 1994. ACM Press.

[17] D. J. Richardson. Taos: Testing with analysis and oracle support. In *ISSTA '94: Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, pages 138–153, New York, NY, USA, 1994. ACM Press.

[18] D. J. Richardson, S. Leif-Aha, and T. O. O'Malley. Specification-based Test Oracles for Reactive Systems. In *Proceedings of the 14th International Conference on Software Engineering*, pages 105–118, May 1992.

[19] H. M. Sneed and S. Huang. Wsdltest - a tool for testing web services. In *WSE '06: Proceedings of the Eighth IEEE International Workshop on Web Site Evolution*, pages 14–21, 2006.

[20] J. Su and P. R. Ritter. Experience in testing the motif interface. *IEEE Software*, 8(2):26–33, 1991.

[21] P. A. Vogel. An integrated general purpose automated test environment. In *ISSTA*, pages 61–69, 1993.