Smart: A Tool for Application Reference Testing

Qing Xie, Mark Grechanik, and Matthew Hellige
Accenture Technology Labs
Chicago, IL 60601, USA
{qing.xie, mark.grechanik, matthew.hellige}@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 to demonstrate a SysteM for Application Reference Testing (SMART) novel approach for generating tests for web services from legacy GAPs. During demonstration of Smart we will show how this tool enables nonprogrammers to generate unit test cases for web services by performing drag-and-drop operations on GUI elements of legacy GAPs. We published a research paper that describes our approach and the results of the evaluation of Smart [2].

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Testing tools

General Terms

Verification

Keywords

Reference Testing, GUI, Web Service

1. INTRODUCTION

When legacy *Graphical User Interface (GUI) APplications (GAPs)* 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 [3]. *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*[10, 11, 6]. In this demo 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 [7, 9, 8, 4]. Some companies spend on average close to 25% of project time to extract test data from reference applications [1].

A core of our proposed demonstration is to extract data from legacy reference GAPs and use this data to generate test cases for target web services, that are software components that interoperate over the Internet, and they gain widespread acceptance partly because of the business demand for applications to exchange information [5]. 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 to demonstrate 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. A detailed description of our approach and the results of the evaluation of SMART can be found in our research paper [2].

2. AN OVERVIEW OF SMART

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 1 with

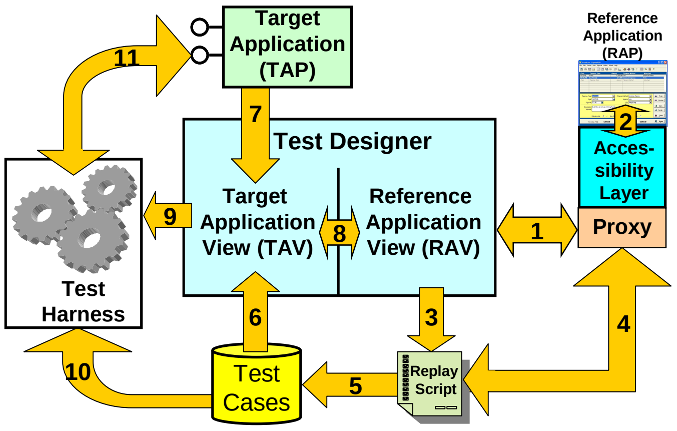

Figure 1: The architecture of SMART.

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.

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.

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.

3. REFERENCES

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

[2] K. M. Conroy, M. Grechanik, E. S. Liongosari, M. Hellige, and Q. Xie. Automatic test generation from gui-based applications for testing web services. In *ICSM*, page to appear, 2007.

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

[4] L. K. Dillon and Q. Yu. Oracles for checking temporal properties of concurrent systems. In *FSE '94*, pages 140–153, Dec. 1994.

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

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

[7] D. Peters and D. L. Parnas. Generating a test oracle from program documentation: work in progress. In *ISSTA '94*, pages 58–65, New York, NY, USA, 1994. ACM Press.

[8] D. J. Richardson. Taos: Testing with analysis and oracle support. In *ISSTA '94*, pages 138–153, New York, NY, USA, 1994. ACM Press.

[9] D. J. Richardson, S. Leif-Aha, and T. O. O'Malley. Specification-based Test Oracles for Reactive Systems. In *ICSE '92*, pages 105–118, May 1992.

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

[11] P. A. Vogel. An integrated general purpose automated test environment. In *ISSTA '93*, pages 61–69, 1993.