Creating GUI Testing Tools Using Accessibility Technologies

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, and existing tools do not fully address different aspects of the testing process, test engineers create custom testing tools to automate the testing process. These tools interact with GAPs by performing actions on their GUI objects. An extra effort that test engineers put in writing test tools is paid off when these tools are run repeatedly on different GAPs. Unfortunately, creating custom GUI testing tools is a laborious and intellectually intensive process, during which test engineers use platform-specific libraries and techniques. As a result, these tools are expensive, difficult to maintain and evolve, and they often run only on specific platforms.

We offer a universal approach for creating custom testing GUI tools. This approach is lightweight, portable, non-intrusive, universal, and cheap, and it combines a non-standard use of accessibility technologies for accessing and controlling GAPs in a uniform way with a visualization mechanism that enables test personnel to interact with GUI objects by performing point-and-click, drag-and-drop operations on GAPs. We describe how we used this approach to create various GUI testing tools, delve into technical features of accessibility technologies, and review our experience with this approach.

1 Introduction

Graphical User Interface (GUI)-based APplications (GAPs) are ubiquitous and provide a wealth of sophisticated services. Manual *black-box testing* of 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 [10][11][6]. In order to automate this process, testing tools are created for test engineers, and these tools mimic users by performing actions on GUI objects of GAPs using some underlying testing frameworks. An extra effort put in creating custom testing tools is paid off when these tools are run repeatedly to determine if GAPs behave as desired.

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. Testing tools are valuable in creating *test scripts* that are programs that interact directly with GUI objects when realizing some testing logic. Reusing testing logic repeatedly is the ultimate goal of test automation.

Crafting testing tools from scratch is a significant investment. There are many reasons why new testing tools should be created. Existing commercially available testing tools are expensive and closed, they do not export interfaces that test engineers can use to automate custom testing tasks. In addition, open-source as well as commercial tools have limited functionality, and it is difficult to extract their standalone components to use them in open testing environments.

An extra effort that test engineers put in writing test tools is paid off when these tools are run repeatedly on different GAPs, reusing testing logic that is created with the help of these tools. In addition, releasing new versions of GAPs with modified GUIs breaks their corresponding test scripts thereby obliterating the benefits of test automation [14][5]. To reuse these scripts, test engineers should fix them, and this process is laborious and intellectually intensive, and GUI testing tools provide significant help in alleviating this problem [12].

Unfortunately, creating custom GUI testing tools is a laborious and intellectually intensive process, during which test engineers use platform-specific libraries and techniques. As a result, these tools are expensive, difficult to maintain and evolve, and they often run only on specific platforms.

GUI testing tools are important in creating models of GUIs that can guide testing of GAPs by checking operations in scripts against elements of GUI models. Specifi-

cally, model-based GUI regression testing involves building and comparing high-level models of GAPs before applying algorithms that construct test cases for evolved GAPs [23]. 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.

We offer a novel approach for creating custom testing GUI tools that interact with GUI objects. 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 interact with GUI objects by performing point-and-click, drag-and-drop operations on GAPs. We described how we used this approach to create various GUI testing tools, delve into technical features of accessibility technologies, and review our experience with this approach.

2 Background

In this section, we give background on the current state of the art of test automation for GAPs, and we present a test automation model.

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 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.

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

[1] http://en.wikipedia.org/wiki/QuickTest_Professional
[2] http://abbot.sourceforge.net
[3] http://selenium.openqa.org
[4] http://www-306.ibm.com/software/awdtools/tester/functional

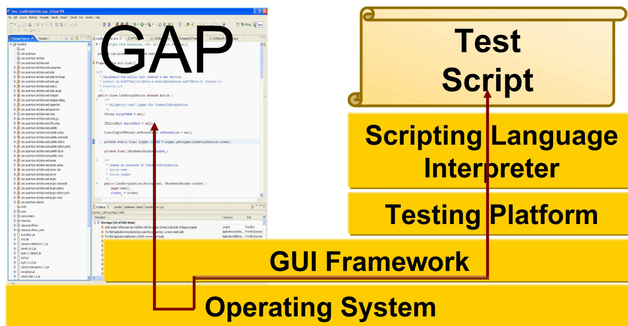

Figure 1. A model of interactions between test scripts and GAPs.

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.

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.

3 Our Solution

In this section, we present core ideas behind our approach, provide a background on accessibility technologies, and give a high-level overview of how our solution is used to create web services from GAPs.

3.1 Core Ideas

Our main goal is to enable test engineers to create GUI testing tools that mimic 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 objects. Then users 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 GUI testing tools, which we collectively call *the Designer*, to mimic this procedure, they should be able to access GUI objects of GAPs programmatically. A core idea of our solution is that GAPs and their GUI objects are programming objects whose values can be set and retrieved and whose methods are associated with actions that users perform on these objects. For example, a combo box object 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, the Designer 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 solution is to use GAPs as programming objects and GUI objects of these GAPs as fields of these objects, and to perform actions on these GUI objects by invoking methods on the objects that represent these GAPs. Unfortunately, the Designer cannot access and manipulate GUI objects of GAPs as pure programming objects because GUI objects 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 objects that have this interface. We give an overview of the accessibility technologies in the next Section 3.2.

3.2 Accessibility Technologies

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. In fact, we use accessibility technlogies as the universal reflective connector as it is defined in our model in the Section 2.2.

Accessibility technologies provide different aids to disabled computer users [2]. 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* [2]. 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 objects.

Accessibility technologies provide a wealth of sophisticated services required to retrieve attributes of GUI objects, set and retrieve their values, and generate and intercept different events. In this paper, we use MSAA for Win-

dows, 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 objects expose a well-known interface that exports methods for accessing and manipulating the properties and the behavior of these objects. For example, a Windows GUI object 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 objects of GAPs as if these objects were standard programming objects.

Using accessibility technologies, programmers can also register callback functions for different events produced by GUI objects thereby obtaining timely information about states of the GUI objects of the GAPs. For example, if a GUI object 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 Designer that controls the GAP.

3.3 Hooks

Hooks are user-defined libraries that contain *callback functions* (or simply *callbacks*), which are written in accordance with certain rules dictated by accessibility technologies. Hooks are important for the Designer because they enable users to extend the functionality of GAPs, specifically to inject faults or mutate interfaces without changing GAPs' source code. Writing hooks does not require any knowledge about the source code of GAPs.

In our approach, a hook library is generic for all GAPs, and its goal is to listen to events generated by the GAP into which this hook is injected as well as to execute instructions received from integrated systems. An example of an instruction is to disable a button until certain event occurs. The power of hook libraries is in changing the functionalities of existing GAPs without modifying their source code.

Main functions of the generic hook are to receive commands to perform actions on GUI objects, to report events that occur within GAPs, and to invoke predefined functions in response to certain commands and events. 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 low-level binary rewriting techniques.

When a target GAP is started, the accessibility layer loads predefined hook libraries in the process space of this applications and registers addresses of callbacks that should be invoked in response to specified events. Since hooks "live" in the process spaces of GAPs, their callbacks can affect every aspect of execution of these GAPs.

3.4 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 objects of the GAP. A GAP can be represented as a tree, where nodes are GAP GUI objects and edges specify that children objects are contained inside their parents. The root of the tree is the main window, the nodes are container objects, and the leaves of the tree are basic objects.

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

3.5 Representing GUIs Programmatically

Recall our idea that GUI objects are programming objects whose values can be set and retrieved and whose methods can be invoked. As such, GUI objects can be represented as classes residing inside web services. Since GUI screens consist of different GUI objects, these screens are also GUI objects (i.e., windows) and can also be represented as classes whose fields are instances of the classes representing these GUI objects. 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 objects, 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 objects may remain the same, but their values and some of their properties change.

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 object, 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 objects 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 objects 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.

4 Implementation Techniques

In this section we describe implementation techniques that we used with accessibility technologies to generate programming objects that control and manipulate GAPs.

4.1 Representing GUI Objects As Classes

The Designer takes inputs describing states of 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. The Designer also emits code that handles exceptions that may be thrown when web services control GAPs (e.g., when a GAP shows a dialog error box that informs users about incorrectly formatted input, the Designer or scripts intercept the dialog, suppress it, and throw a programming exception in the calling client).

When a GAP switches 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 a class whose declaration is shown in Figure 2. Generated classes are linked to GUI elements, and these classes contain methods for setting and getting values and performing actions on these elements.

4.2 Representing GUI Screens As States

A declaration of the base class `State` is shown in Figure 3. The Designer creates classes using this class `State` as a template for GAP states. All GUI elements in a given state are fields of the generated state class and these fields are the instances of their corresponding classes (e.g., the field `GUIObject`). A GAP transition is caused by calling

```
public class GUIElementClass {
   public void Set( type var ) {...}
   public type Get() {...}
   public void DoIt(String act}{...}}
```

Figure 2. Declaration of a class representing a GUI element.

the method `Switch`, which in turn calls the `DoIt` method of an object representing some GUI element in the given state. The member variable `stateID` is a state enumerator, which is incremented every time the method `Switch` is called.

```
public class State {
    private int stateID = 0;
    public void Switch(){
        ++stateID;
        switch(stateID) {
           case 1: GUIObject.DoIt(null);
                break;
           ..........  }}}
```

Figure 3. A declaration of the class `State`.

5 Our Experience

In this section, we describe GUI testing tools that we created using the accessibility-based approach.

5.1 SMART

A *SysteM for Application Reference Testing (Smart)* for generating tests from reference GAPs and applying these tests to the corresponding target web services [9]. 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.

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

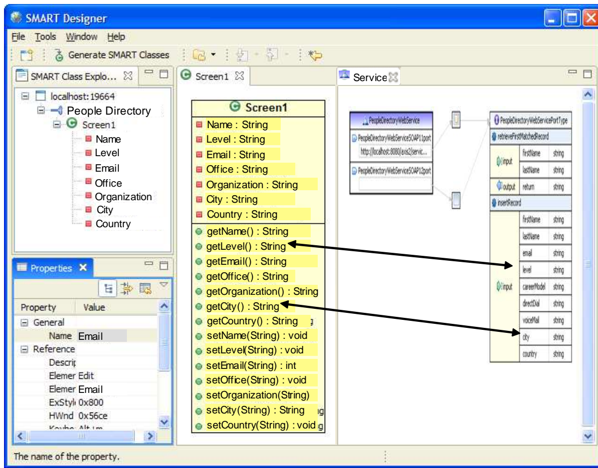

Figure 4. The front-end of the Designer for SMART.

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 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 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.

A central component of Smart is the 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.

Once testers define all mappings, the Designer outputs a test harness that contains the driver for running tests on the TAP. When this harness is run, it uses test cases to test the target web service. 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.

5.2 REST

We built a tool for *Reducing Effort in Script-based Testing (REST)* for guiding test personnel through changes in test scripts so that they can test their modified GAPs [12]. The core of our approach is to enable test personnel to main-

tain and evolve test scripts with their respective GAPs by providing assistance in determining how to change these scripts with a high degree of automation and precision.

The input to REST are GUIs of the successive releases of the same GAP and a test script for the prior release of the GAP. After extracting models of these releases using accessibility technologies, these models are compared and modified GUI objects are located. Then, the test script is analyzed to determine references to these modified GUI objects and their impact on other statements in this script. The result of this analysis is issued warnings that help test engineers to fix errors in test scripts so that they can test successive releases of their GAPs

6 Related Work

GAPs present special challenges to regression testing because the input-output mapping does not remain constant across successive versions of the software [20][17]. 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 [3], analyzing changes in functions, types, variables, and macro definitions [7][15], using def-use chains [13], constructing procedure dependence graphs [8][22], and analyzing code and class hierarchy for object-oriented programs [16][21]. 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 [18]. To our knowledge GUITAR uses platform and language-specific techniques that cannot be easily extended to other platforms. For example, it is unclear how GUITAR can be extended to web-based GAPs.

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 [23]. We consider our approach complementary since it could be used to extend tools easily to extract GUI models from GAPs that run on different platforms and improve its precision and usability by utilizing richer models.

When it comes to extracting information from GAPs and their GUI elements, the term screen-scraping summarily describes various techniques for automating user interfaces [1][]. 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 [19]. 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.

There are few commercial tools available for controlling and manipulating GUI objects. AttachmateWRQ and Seagull Software Corp. are the largest and oldest companies providing tools for controlling and manipulating GAPs. For each platform they have separate lines of products that exploit ad-hoc platform-specific techniques for screenscraping. These and other companies have built different tools for controlling and manipulating GAPs for different platforms. Doing that results in multiple versions of the source code for these tools, subsequenlty their increased cost, and eventually difficulties in maintaining and evolving different codebases. In constrast, our approach uses a platform-independent mechanism for controlling and manipulating GAPs.

7 Conclusion

We offer a novel approach for creating custom testing GUI tools. 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 interact with GUI objects by performing point-and-click, drag-and-drop operations on GAPs. We described how we used this approach to create various GUI testing tools, delved into technical features of accessibility technologies, and reviewed our experience with this approach.

References

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

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

[3] 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.

[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] K. M. Conroy, M. Grechanik, M. Hellige, E. S. Liongosari, and Q. Xie. Automatic test generation from gui applications for testing web services. In *ICSM*, pages 345–354, 2007.

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

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

[12] M. Grechanik, Q. Xie, and C. Fu. Maintaining and evolving gui-directed test scripts. In *ICSE '09*, pages 571–579, 2009.

[13] 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.

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

[15] 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.

[16] 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.

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

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

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

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

[21] 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.

[22] 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.

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