A Methodology and Framework for Automatic Layout Independent GUI Testing of Applications Developed in Magic xpa

Daniel Fritsi, Csaba Nagy, Rudolf Ferenc, Tibor Gyimothy

Department of Software Engineering
University of Szeged, Hungary
fritsi@frontendart.com, {ncsaba|ferenc|gyimi}@inf.u-szeged.hu

Abstract. Testing an application via its Graphical User Interface (GUI) requires lots of manual work, even if some steps of GUI testing can be automated. Test automation tools are great help for testers, particularly for regression testing. However these tools still lack some important features and still require manual work to maintain the test cases. For instance, if the layout of a window is changed without affecting the main functionality of the application, all test cases testing the window must be re-recorded again. This hard maintenance work is one of the greatest problems with the regression tests of GUI applications.
In our paper we propose an approach to use the GUI information stored in the source code during automatic testing processes to create layout independent test scripts. The idea was motivated by testing an application developed in a fourth generation language, Magic. In this language the layout of the GUI elements (e.g. position and size of controls) are stored in the code and can be gathered via static code analysis. We implemented the presented approach for Magic xpa in a tool called Magic Test Automation, which is used by our industrial partner who has developed applications in Magic for more than a decade.

1 Introduction

Thoroughly testing an application via its user interface is not an easy task for large, complex applications with many different functionalities. Testers have to follow certain steps of thousands of test cases and need to evaluate the results manually. This hard work can be supported by automatic GUI testing tools, as these tools are able to follow and record user events (mouse, keyboard, etc.) generated by testers then play back these events to the application under test. This is a great help for regression tests, for example, where the aim is to re-test the application after a change. However, there remains still a lot of manual work to be done. Testers need to record the test case for the first time when they create it, and they need to maintain the recorded scripts as the application evolves.

Current tools support the most popular 3rd generation languages (e.g. C/C++, Java, C#), however higher level languages such as 4th generation languages

(Magic 4GL, ABAP, Informix) became also popular in software development. Developers programming in these languages do not write source code in the traditional way, but they develop at a higher level of abstraction, for instance, using an application development environment. In such languages the application code usually stores the description of the user interface too (e.g. structure of a window or a form and position, color or size of a control). In our paper we use this information to make the automatic testing process GUI layout independent. That is, a recorded test script does not depend on exact coordinates or the layout of the GUI, so the same test case can be reused later even when the developers make minor changes to the user interface of the application (e.g. they rearrange the buttons in a window).

One of the greatest problems with regression tests for GUI applications is that even a minor change in the GUI may result in rewriting all the test cases [4]. As a possible solution, our technique may significantly reduce the costs of maintaining regression tests to keep the quality of a GUI application assured.

The main contributions of this paper are:

- we propose a method to record and play back automatic GUI test scripts that are unaffected by minor changes of the GUI, hence they are layout independent;
- we present our approach in an "in vivo" industrial context, as our tool is used by our industrial partner for testing Magic xpa applications. The presented approach was implemented during a research project in co-operation with our industrial partner, SZEGED Software Inc. During the project the tool was experimentally used for automated GUI testing, and it was extended with additional features. For further details on the project please refer to its webpage[1].

2 Automated Software Testing

Sommerville introduces the main goal of software testing in [18] as follows: *„testing is intended to show that a program does what it is intended to do and to discover program defects before it is put into use"* . In the field of software testing, automated software testing is a relevant software engineering topic nowadays, mostly motivated by the industry. As a result many papers and books have been published in this area [5], [6], [7], [12], [17]. Here we elaborate on the literature and on related tools focusing on those that are closely related to our work.

Testing automation frameworks are usually divided into 5 generations [10], [11]. 1st generation frameworks are so-called record/playback tools that are based on simple test scripts where one script relates to one test case. The 2nd generational tools have scripts that are better designed to use/reuse functions, for example. 3rd generation frameworks take data out of the test scripts so a test script may be re-executed several times on different data. This concept is called

[1] http://www.infopolus2009.hu/en/magic

data-driven testing [7], [19]. Another concept, usually referred to as 4th generation testing is called keyword-driven, where the test creation process is separated into a higher level planning stage and an implementation stage, thus keywords defined at higher level drive the executions [2], [3], [7]. New techniques sometimes bring test automation to an even higher, so-called scriptless level (5th generation), where automated test cases are designed by engineers instead of testers/developers [9].

Our approach can be considered as a 3rd generation approach, because with carefully designed test scripts, the data can be separated from the execution process. The idea of keyword driven testing is also similar, but our test script is still at lower level, close to the implementation.

The idea of supporting the recording and playback of test cases by using test scripts based on GUI information from source code is novel to our best knowledge in 4GL context. However, static analysis is a common tool to support GUI testing in other approaches, e.g. for generating test scripts [8], [13], [14], [16].

There are a number of automatic GUI testing tools available for software engineers. Just to mention some examples, GUITest[1] is a Java library for automated robustness testing, Selenium[2] is a GUI testing tool for Web applications. As an application testing a web page it also provides solutions to simplify test scripts by using the identifier of a control from the HTML code of the web page. This is a similar approach to ours for Web applications. TestComplete[3], HP Quality Center and Quick Test Professional (QTP)[4] tools are also a widely used for applications written in 4GLs. Microsoft also provides automated GUI testing for instance via GUI Automation of the .NET Framework[5].

3 Specialties of a Magic Application

In the early 80's Magic Software Enterprises (MSE) introduced a new fourth generation language, called Magic 4GL. The main concept was to program an application at a higher level meta language, and let an application generator engine create the final application. A Magic application could run on popular operating systems such as DOS and Unix, so applications were easily portable. Magic evolved and a new version of Magic has been released, uniPaaS and lately Magic xpa. The new version supports modern technologies such as RIA, SOA and mobile development too.

The unique meta model language of Magic contains instructions at a higher level of abstraction, closer to business logic. When one develops an application in Magic, she/he actually programs the Magic Runtime Application Environment

[2] http://seleniumhq.org/

[3] http://smartbear.com/products/qa-tools/automated-testing/

[4] HP Test Management (accessed 2013): http://www8.hp.com/us/en/software-solutions/software.html?compURI=1170256

[5] Microsoft UI Automation Overview (accessed 2013): http://msdn.microsoft.com/en-us/library/ee684076%28v=vs.85%29.aspx

(MRE) using its meta model. This meta model is what really makes Magic a RADD (Rapid Application Development and Deployment) tool.

Magic comes with many GUI screens and report editors as it was invented to develop business applications for data manipulation and reporting. The most important elements of Magic are the various entity types of business logic, namely the data tables. A table has its columns which are manipulated by a number of programs (consisting of subtasks) linked to forms, menus and help screens. These items may also implement functional logic using logic statements, e.g. for selecting variables (virtual variables or table columns), updating variables, conditional statements.

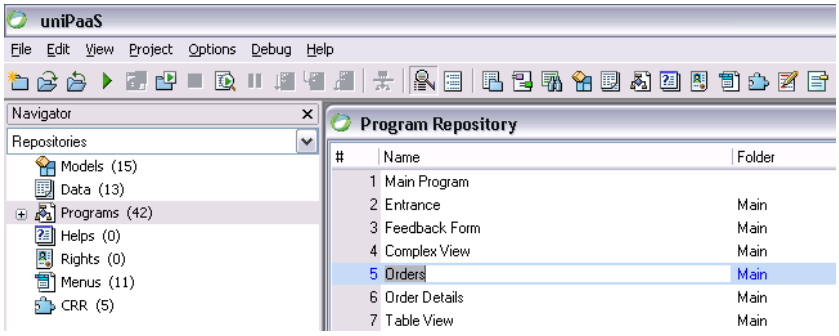

Fig. 1. A screen shot of the Magic xpa application development framework.

Figure 1 is a screen shot of the Magic xpa development environment. Some major components of Magic xpa, as a 4th generation programming language are:

Data Objects. These are essentially the descriptions of the database tables. Just as the tables and their columns and primary or foreign keys are defined in a database, we can define these objects in Magic xpa too.

Programs. The logic of an application is implemented here. Programs are top-level tasks with several subtasks below them. A task always works on some Data Objects and performs some operations on them. We can define which database tables should the task use, and which operations should the task perform on them.

Menus. In the application, we can use different high-level menus and pop-up menus, which can be defined here.

Form Entries. Magic xpa has a form editor, where we can define the properties of a window (e.g. title, size and position) and we can place controls and menus on a form and customize them. A graphic window, a form is FormEntry in Magic xpa. In the Magic xpa development environment we can use many built-in controls or we can define our custom controls too. A form is always defined within a task. The form editor of Magic xpa is shown in Figure 2.

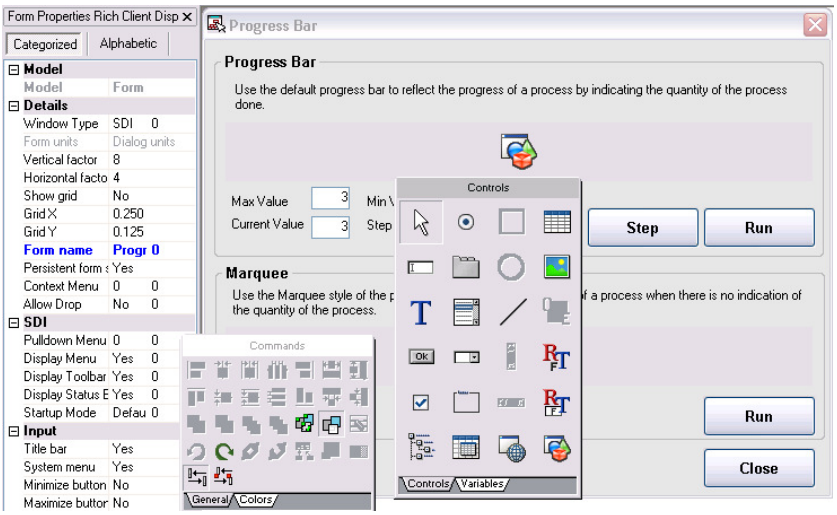

Fig. 2. A screen shot of the form editor of Magic xpa.

4 Automatic GUI Testing of a Magic Application

We implemented a tool called *Magic Test Automation*, which enables the automatic GUI testing of applications implemented in Magic xpa. The automatic testing of a Magic application has three main steps (see Figure 3):

1. Analyzing the Magic application. Here we perform a static analysis of the application to gather all the required data of its GUI.
2. Recording GUI events. This is the step where we monitor the mouse and keyboard events and use them to create layout independent test scripts.
3. Playback recorded GUI events. We use the layout independent test scripts to simulate mouse and keyboard events on the application being tested.

In case of layout-independent testing, once the application gets changed in the future, it is enough to repeat the analyzing and the playback steps, and re-recording test cases is not necessary.

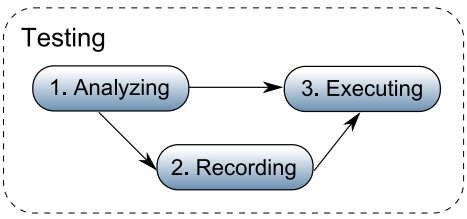

Fig. 3. Main steps of automatic GUI testing of a Magic application.

4.1 Static Analysis of Magic Applications

A Magic application does not have source code in the "traditional way", it is described by a save file of its current model. In the older Magic versions this save was a structured text file, but in the newer versions such as Magic xpa, this is an XML file. During the analysis of a Magic application we extract information from this source file. As the result of the analysis, a graph describing the structure of the program is created, which is called an Abstract Semantic Graph (ASG).

A node of the ASG represents an item in the source code. All these nodes are instances of the corresponding source elements. Two nodes can be connected with two types of relations: aggregation and association. Aggregation can be used to describe complex grammar elements (edges of the syntax tree) and with association we can describe semantic details (e.g. identifier references). The graph is created by a static analyzer tool, which parses the save file of the application being analyzed, creates the nodes and puts them together in the ASG.

For further details about reverse engineering Magic applications please refer to our previous work [15].

4.2 Recording GUI Events

Recording GUI events is the process where we record the way the user interacts with the application under test into a certain script format. We catch the events generated by the user and we try to identify the related source element, then transform it to a command of a test script. Of course, user could write such a script manually, but for complex test cases it would be almost impossible.

Traditional, coordinate based automatic testing techniques record the event type and its position. In our layout-independent technique we record the event type and the identifier (in the source code) of the control on which the event occurred.

Hence, the most important task of recording is to identify the source element on which the actual user event happened. To be able to do this, we use dynamic traces of the executed application to identify the currently running tasks and form elements that are displayed on the screen. Once we catch a user event based on its position on the screen and the dynamic traces, we can identify the certain control of the source code, which is actually stored in the ASG. Figure 4 illustrates the process of the recording.

Recording is performed on Windows platform using Windows API. Catching a user event is based on Windows' hook mechanism (*SetWindowsHookEx*, *HookProc* functions).

In Figure 5 we illustrate the possible steps that a tester would perform testing a sample window of a Magic application. We recorded the illustrated steps with the Magic Test Automation tool and saved the script in Python format. Figure 6 shows the resulting Python script.

It can be seen that the Magic Test Automation tool connects the Magic code with the ASG and generates a script using the obtained identifiers. A traditional

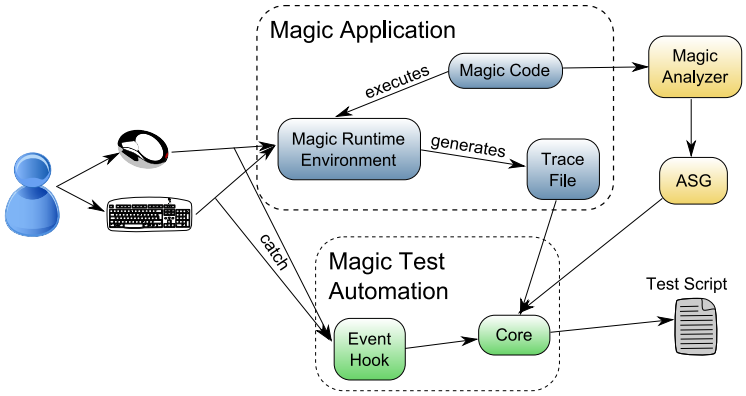

Fig. 4. Recording GUI Events.

coordinate-based method would result in a script containing only coordinates, e.g. as it can be seen in Figure 7.

One can see that both scripts contain the same amount of instructions. When we execute the two scripts they will produce the same result, but what happens when we rearrange the window? (For illustration, see a rearranged window in Figure 8.) The application would work as before, but the controls would be in different positions. If we executed the layout independent Python script the result would be the same as before, because the Magic Test Automation tool recalculates the coordinates by the unique identifiers. In contrast, if we play the position based Python script then the result will be negative, because the controls are not in the positions as before.

4.3 Playback Recorded GUI Events

Once we have the test script, we need to be able to playback the recorded user events to the application, this is based on executing events of the script. However this is not enough, as the execution needs to be evaluated and we must make sure that the program under test behaves the same way as it did when we recorded the test script. This is done during the validation phase.

Executing Events In traditional, coordinate based techniques, executing a user event is simple, as the recorded event must be sent to the application with the recorded position. In our layout-independent technique we have no coordinates stored in the test script, but we store the identifier of the control.

Hence, during execution we calculate the coordinates of the control from the ASG, and transform these coordinates to positions on the screen.

If the application is modified, we can re-run the same test script, but with the ASG of the new version of the application (see Figure 9 for illustration).

To play back a recorded test script, first of all we need the script file, and the ASG to connect the unique identifiers of it with the corresponding Windows

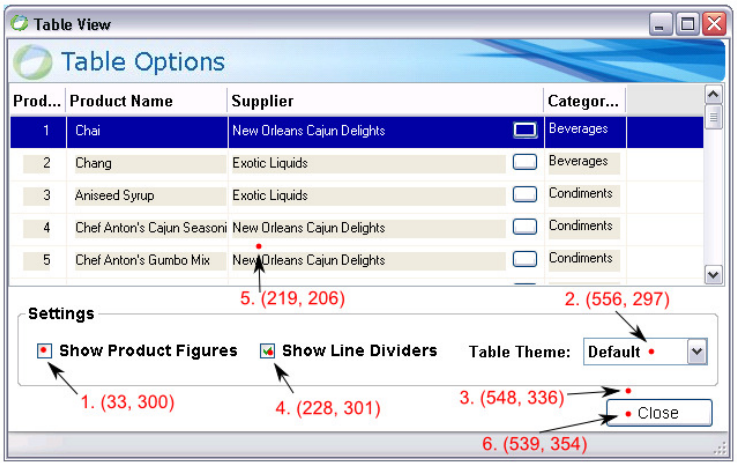

Fig. 5. An example window of a Magic xpa application with example steps for testing its GUI.

```
from testrunner import *
from testrunner_ext import *

def runScript():
    assert waitFor(3014, findWindow(741, 379, "Table View")) == True
    mouseClickAtControl("Fe~Table View::Ct~Divider Box", MouseButtons.LEFT, 11, 23)
    mouseClickAtControl("Fe~Table View::Ct~Theme Combo", MouseButtons.LEFT, 44, 14)
    mouseClickAtControl("Fe~Table View::Ct~Theme Combo", MouseButtons.LEFT, 47, 33)
    mouseClickAtControl("Fe~Table View::Ct~Show Figures Box", MouseButtons.LEFT, 13, 17)
    mouseClickAtControl("Fe~Table View::Ct~Table", MouseButtons.LEFT, 218, 129)
    mouseClickAtControl("Fe~Table View::Ct~Close Button", MouseButtons.LEFT, 36, 22)
```

Fig. 6. A layout independent Python script for the steps in Figure 5.

GUI elements. During the playback we must execute the Magic application in the Magic Runtime Environment and we must load the script file in the Magic Test Automation tool. The script file contains the recorded keyboard and mouse events which the Test Automation tool first interprets and then executes. (An illustration can be seen in Figure 10.)

During the interpretation we locate GUI elements in the ASG via their unique identifiers. After that, we identify the same Windows controls of the running application. This identification is sometimes quite complex as the lower level implementation of a control may be totally different than the simple Magic control. Suppose a complex tree control or a group box built from many smaller controls. In order to solve this identification problem we collect all information from the ASG that we need to identify a GUI element (position, size), but this is still not enough as the application can simultaneously display multiple windows and parent windows too. Therefore, we need all information from its parent elements too. This way we know that on which window the current element is located. Using the Windows API we can find windows and GUI elements by

```
from testrunner import *
from testrunner_ext import *

def runScript():
    assert waitFor(3014, findWindow(741, 379, "Table View")) == True
    mouseClickAXY(MouseButtons.LEFT, 33, 300)
    mouseClickAXY(MouseButtons.LEFT, 556, 297)
    mouseClickAXY(MouseButtons.LEFT, 548, 336)
    mouseClickAXY(MouseButtons.LEFT, 228, 301)
    mouseClickAXY(MouseButtons.LEFT, 219, 206)
    mouseClickAXY(MouseButtons.LEFT, 539, 354)
```

Fig. 7. A coordinate based Python script for the steps in Figure 5.

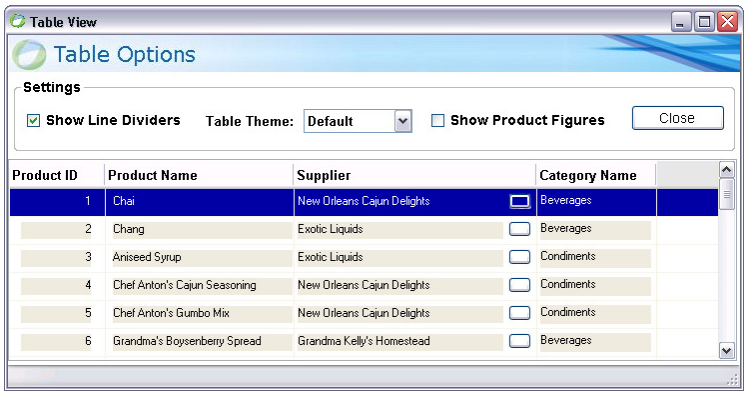

Fig. 8. A rearranged window of the example uniPaaS application (see Figure 5).

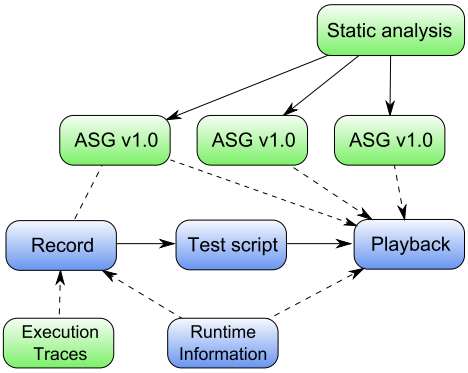

Fig. 9. After a new version, the same test script can be executed with the new ASG.

header texts, positions and parent window identifiers. So, we get the handle of the window with the *FindWindow* and *FindWindowEx* functions by the header text and other attributes of it, which we read from the ASG. We can also calculate

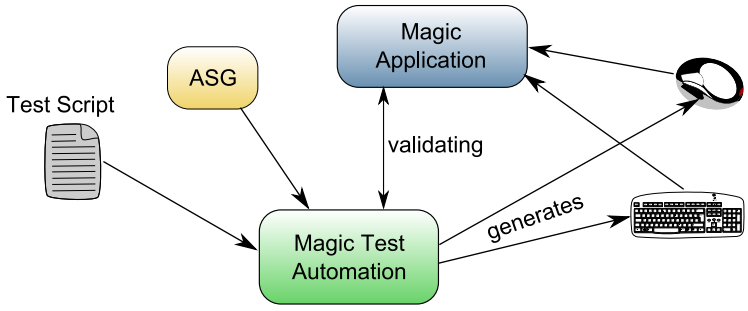

Fig. 10. Running Recorded GUI Events.

the relative coordinates to the window of the currently searched GUI element. As the GUI element can be within other GUI elements such as a group box, we start looking for it from the bottom of the Windows control tree and walk upwards to the top. We recalculate the relative coordinates until we get to the searched GUI element.

It is not always enough to know which Windows control matches a control with a unique ASG identifier because we must know the exact position where to click within the GUI element. In case of a button this is irrelevant, but in case of a tree view it is not. The Magic Test For complex controls, the automation tool generates script files where we store a position as the relative position to the identified Magic control. Based on these coordinates we can calculate the absolute position where we can generate the keyboard or mouse event using the Windows API.

Evaluating an execution Some steps of the evaluation can be done automatically after the test script was executed, however it is always necessary to tell the automation tool the validation steps manually after recording a test script. The tester can do it by inserting validation (e.g. assert) functions into the script file after the corresponding event handler. The Magic Test Automation tool supports the following validation possibilities:

- To check anywhere in the application's control tree, or in a particular window whether it contains a text or there is a window with a given title.
- Comparison of a specific GUI element's text with a given text.
- Verify that a GUI element is in focus or not.
- Verify that a GUI element is enabled or not.
- Verify that a check box or radio button is checked or not.

The Magic Test Automation tool will check these asserts and report the result of a test script accordingly.

Another advantage of these validation functions is that in addition to evaluate the results of an execution, one can use them in the previously mentioned delay functions too. For example, one can easily say that she/he wants to wait until a

```
from testrunner import *
from testrunner_ext import *

def runScript():
    assert waitFor(3014, findWindow(741, 379, "Table View")) == True
    mouseClickAtControl("Fe~Table View::Ct~Divider Box", MouseButtons.LEFT, 11, 23)
    assert validate(checkState("Fe~Table View::Ct~Divider Box", CheckStates.Checked)) == True
    mouseClickAtControl("Fe~Table View::Ct~Theme Combo", MouseButtons.LEFT, 44, 14)
    mouseClickAtControl("Fe~Table View::Ct~Theme Combo", MouseButtons.LEFT, 47, 33)
    assert validate(compareText("Fe~Table View::Ct~Theme Combo", "Rose")) == True
    mouseClickAtControl("Fe~Table View::Ct~Show Figures Box", MouseButtons.LEFT, 13, 17)
    assert validate(checkState("Fe~Table View::Ct~Show Figures Box", CheckStates.Checked)) == True
    mouseClickAtControl("Fe~Table View::Ct~Table", MouseButtons.LEFT, 218, 129)
    assert validate(checkFocus("Fe~Table View::Ct~Show Figures Box", False)) == True
    mouseClickAtControl("Fe~Table View::Ct~Close Button", MouseButtons.LEFT, 36, 22)
    assert validate(findWindow(741, 379, "Table View")) == False
```

Fig. 11. Examples for validations in a Python script.

check box is checked or a specific text box contains a given text. Moreover, with Python scripts we can use them to control the execution of the test case. E.g. we can define complex test cases where we say that if a GUI element is activated then we want to do certain steps, otherwise we want to do a different chain of steps.

Figure 11 illustrates the Python script shown in Figure 6, extended with validation instructions. After clicking the check boxes there is a checkState function which checks that the check box is really checked or not. After selecting an item from the combo box there is a compareText function which checks that the combo box contains the correct text and after clicking in the table we check that the *"Fe˜TableView::Ct˜Show Figures Box"* has the focus or not. Finally, after clicking the *"Fe˜Table View::Ct˜Close Button"* button we check if the window closed successfully or not.

5 Comparison to Other Techniques

A comparison of some aspects of common techniques and our approach can be seen in Table 1. Here we elaborate on these techniques in details.

Keyword-driven testing A keyword in its simplest form is an atomic test step or an aggregation of more atomic steps. It describes an action to be performed, hence keyword-driven testing is usually referred as action-word testing too. Most of the cases the keyword-driven testing is divided into two stages:

- planning stage,
- implementation stage.

In the planning stage test engineers determine the test steps for each test case (e.g. entering a text into a text field, clicking on a button, etc.). Later, in the implementation stage the engineers can use a framework to write the previously planned test scripts in a format which can be executed by the framework. A

	Keyword-driven	Data-driven	Modularity-driven	Coordinate-based	White-box based	Presented approach
no need of programming skills to design test scripts	X	X	X			
no hard-coded data in test scripts		X				X
combinable test scripts			X	X	X	X
no source code required to design test scripts	X	X	X	X		
test script execution handles rearrangements in windows					X	X

Table 1. Key features of different testing techniques that our tool can handle.

special system under test may require unique actions and keywords which are important to be supported by the testing framework.

In some cases the planning stage and the implementation stage can be combined into one stage and engineers can write the scripts directly into the frameworks scripting format.

Our presented approach can be interpreted as keyword-driven testing because our implemented tool has its own scripting language which is able to understand specific keywords and translate them into mouse, keyboard or other input events. Similar to our tool, Selenium is also a record/replay tool. It is used for testing web applications. It has keywords like Goto WEBSITE or Enter "username", etc. TesComplete is also an automating testing tool which uses keywords to simulate input events. With TestComplete one can also record keyword-driven test scripts and edit them later manually. Another example for a keyword-driven test automation tool is TestArchitect[6] developed by LogiGear Inc.

Data-driven testing Data-driven testing is based on the separation of testing data and execution logic, the tester specifies inputs and verifiable outputs for a test script so that the test script is executed several times on different inputs. Data-driven testing is usually used for testing a form of an application with specific data. So the tester has to specify the input data which the testing framework enters manually into the form under test and then compares the result to the expected output. The main difference between keyword-driven testing and data-driven testing is that in keywod-driven test scripts the data is hard-coded into the test script (e.g. enter "test text" to a textbox) and if one wants to test e.g. the same textbox with different data she/he has to create another test script.

[6] http://www.testarchitect.com/

Our approach relies on Python scripts resulting that it can be used for data-driven testing. With Python, the input data can be stored in variables, which can be initialized even in a separate script file, hence the input and the execution logic can be totally separated. Moreover by using arrays for storing input and expected output data, loops can be used to execute the same keyword several times with the input array. This way hard-coded data sets can be eliminated from our test scripts. Compared to other tools, TestComplete is also capable of specifying input data for test recorded test scripts so TestComplete can also be used for data-driven testing. Using extensions Selenium is also capable of executing test scripts on various input data.

Modularity-driven testing Modularity-driven testing requires writing small, independent test scripts for each modules, packages and functions of the application under test. These small scripts are then used to create larger tests, realizing a particular test case. For example, if one wants to test one of the admin users' functions, she/he has to write a script for testing the login action and another separate script for testing the function itself. Then, in a larger test script, first the login script gets called and if it runs successfully the next script gets called which tests the admin's function.

One benefit of this technique is that one change in a module/function affects only its test cases and others might remain untouched during the maintenance of the test scripts.

With Python scripts, modularity-driven testing is also supported by our approach. One can write separate automated Python test scripts and combine them into a larger script by importing them.

Coordinate-based and white-box testing One common way for automated GUI testing is the coordinate-based testing, because the testing framework doesn't need to know anything about the tested application. Coordinate-based testing is a sort of keyword-driven testing. Usually a keyword contains a coordinate and a user action to be performed on the given coordinate. There are two kinds of coordinate based testing:

- Using absolute coordinates within the application window, where coordinates are relative to usually the upper left corner of the screen. This method does not appears to be very useful, but in many applications the position of the window is not important.
- Using coordinates that are relative to the upper left corner of the currently active window.

Coordinate-based test scripts are the solution if there is no available information about the application under test. However, coordinate-based test scripts are hard to maintain as it might easily change what is exactly on the same coordinate next time when we execute the application.

If we have access to the source code or some documentations of the application under test during its testing phases process it is called white box testing.

Basically our method is a white box testing because we use the layout description of the application to create test scripts.

5.1 Drawbacks of the technique

Besides benefits, there are some important drawbacks which should be discussed here. First, we consider minor changes of the GUI those changes that simply rearrange the layout of the window and does not modify drastically the structure of it. Our method will recognize the control based on its unique identifier, which identifies the control based on its parents in the control tree. If the parent hierarchy changes, we will not be able to recognize the same control again.

Another important drawback is that the method works based on relative coordinates inside the identified controls. These coordinates may strongly depend on the internal layout of the control. For example, in tree controls if the order of the nodes varies between different executions, our tool may not follow the new structure. Similarly, our technique may fail in selecting an exact item from a listbox or a combobox if the list of elements changes.

Another way a developer can exploit our method is to change the size or position of a control at runtime. Since we read this information from the ASG, our method works as long as the size and the position of the control remains unchanged during execution.

6 Conclusions and Future Work

Our approach for layout independent automatic GUI testing is based on user interface descriptions stored in the source code. We use static code analysis to gather user interface descriptions and combine it with dynamic execution traces during the recording phase of a test case. The resulting test scripts contain only layout independent data which can be played back to the application later even if the user interface has been changed. This technique may dramatically lower the costs of regression tests where developers and testers have to maintain thousands of test cases.

We implemented our approach in a special 4GL environment called Magic xpa, and our implementation is currently used by our industrial partner where developers have been working with Magic for more than a decade. Our partner delivers wholesale products where high quality of the delivered product is top priority, which also requires thorough testing processes. We found our approach to be useful for our partner in their regression testing processes.

Using GUI information stored in the source code during automatic GUI testing is a novel approach for Magic 4GL. We note here that the idea can be easily generalized to other languages, where the GUI description can be extracted from the source code by static analysis (e.g. resource files of Delphi or C# applications). However this might not stand for languages where the GUI is usually constructed dynamically, for instance in Java, where the dynamic nature of GUI generation makes our approach hardly applicable.

As future work we plan to improve our validation techniques and to support testing Magic applications with automatic test script and test input generations based also on the results of static analysis.

Acknowledgements

This research was supported by the Hungarian national grants GOP-1.2.1-08-2009-0005 and GOP-1.1.1-11-2011-0039.

References

1. Sebastian Bauersfeld and Tanja E. J. Vos. Guitest: a java library for fully automated gui robustness testing. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 330–333, New York, NY, USA, 2012. ACM.
2. Hans Buwalda. Automated testing with action words, abandoning record and playback. In *Proceedings of the EuroStar Conference*, 1996.
3. Hans Buwalda and Maartje Kasdorp. Getting automated testing under control, software testing and quality engineering. *STQE magazine, division of Software Quality Engineering*, nov/dec 1999.
4. Dimitris Dranidis, Stephen P. Masticola, and Paul Strooper. Challenges in practice: 4th international workshop on the automation of software test report. *SIGSOFT Softw. Eng. Notes*, 34(4):32–34, July 2009.
5. Elfriede Dustin, Thom Garrett, and Bernie Gauf. *Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality*. Addison-Wesley Professional, 1st edition, 2009.
6. Elfriede Dustin, Jeff Rashka, and John Paul. *Automated software testing: introduction, management, and performance*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
7. Mark Fewster and Dorothy Graham. *Software test automation: effective use of test execution tools*. ACM Press/Addison-Wesley Publishing Co., 1999.
8. Svetoslav R. Ganov, Chip Killmar, Sarfraz Khurshid, and Dewayne E. Perry. Test generation for graphical user interfaces based on symbolic execution. In *Proceedings of the 3rd international workshop on Automation of software test*, AST '08, pages 33–40, New York, NY, USA, 2008. ACM.
9. Jeff Hinz and Martin Gijsen. Fifth generation scriptless and advanced test automation technologies, 2009.
10. Cem Kaner. Architectures of test automation, 2000.
11. Edward Kit. Integrated effective test design and automation software development. *Software Development online*, feb 1999.
12. Kanglin Li and Menggi Wu. *Effective GUI Test Automation*. SYBEX Inc., Alameda, CA, USA, 2005.
13. Yongzhong Lu, Danping Yan, Songlin Nie, and Chun Wang. Development of an improved GUI automation test system based on event-flow graph. In *Proceedings of the 2008 International Conference on Computer Science and Software Engineering - Volume 02*, pages 712–715. IEEE Computer Society, 2008.
14. A.M. Memon, M.E. Pollack, and M.L. Soffa. Hierarchical GUI test case generation using automated planning. *IEEE Transactions on Software Engineering*, 27(2):144 –155, feb 2001.

15. Csaba Nagy, László Vidács, Ferenc Rudolf, Tibor Gyimóthy, Ferenc Kocsis, and István Kovács. Solutions for reverse engineering 4GL applications, recovering the design of a logistical wholesale system. In *15th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 343 –346, march 2011.
16. Jan Peleska, Helge Löding, and Tatiana Kotas. Test automation meets static analysis. In *GI Jahrestagung (2)*, volume 110 of *LNI*, pages 280–290. GI, 2007.
17. Bruce Posey. *Just Enough Software Test Automation*. Prentice Hall PTR, 2002.
18. Ian Sommerville. *Software Engineering (9th Edition)*, chapter Software testing. Addison-Wesley, 2010.
19. Richard Strang. Data driven testing for client/server applications. In *Proceedings of the Fifth International Conference on Software Testing, Analysis and Reliability (STAR'96)*, pages 395–400, 1996.