A Layout Independent GUI Test Automation Tool for Applications Developed in Magic/uniPaaS

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

A good software development process involves thorough testing phases, that are usually expensive, but necessary to deliver a reliable and high quality product. Testing an application via its graphical user interface requires lots of manual work, even if some steps of GUI testing can be automated. Test automation tools are a great help for testers, particularly for regression tests. 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. With this technique, the already recorded tests scripts will be unaffected by minor changes in the GUI. It reduces the maintenance effort of very expensive regression tests where thousands of test cases have to be maintained by testing teams. The idea was motivated by testing an application developed in a fourth generation language, Magic/uniPaaS. In this language the layout of the GUI elements (structure of the window, position and size of controls, etc.) are stored in the code and it can be gathered via static code analysis. We implemented the presented approach for Magic/uniPaaS, and our Magic Test Automation tool is used by our industrial partner who has developed applications in Magic/uniPaaS 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 and play back these events to the application under test. This is a great help for regression tests, for example. However, there remains still a lot of manual work to be done. Testers still need to record the test case for the first time when they create it, and they still need to maintain the recorded test cases as the application may evolve and the user interface of the application or its implemented functionalities may change.

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). Since this information can be gathered via static code analysis, it is also available to support testing processes. 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 problem with regression tests for GUI applications is that even a minor change in the GUI may result in rewriting all the test cases. 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/uniPaaS applications.

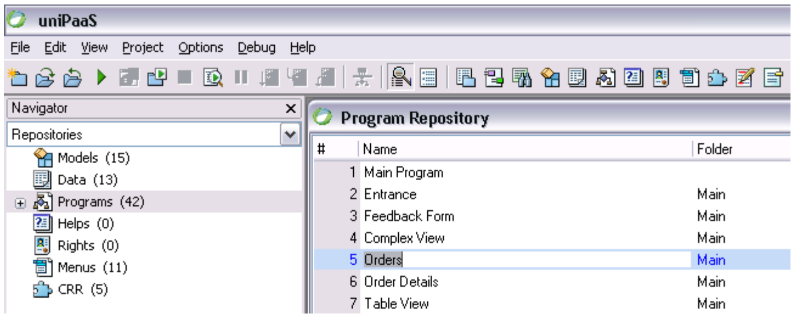
The paper is organized as follows. In Section 2 we give a brief introduction into the world of Magic 4GL. Next, in Section 3 we describe our method for layout independent automatic GUI testing of Magic applications. Here we present the most important steps of recording GUI events, playing them back to the application and evaluating the outcomes of the executed test cases. In Section 4 we present related work and tools. Finally, we conclude our paper and discuss future work in Section 5.

2 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. With this technique MSE could ship ready solutions for user management, printing, database management, etc., and the developers did not need to spend their time on working with these components. 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, which is called uniPaaS. The new version supports technologies such as RIA (Rich Internet Applications) and SOA (Service Oriented Architecture).

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 (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 business logic using logic statements, e.g. for selecting variables (virtual variables or table columns), updating variables, conditional statements.



#	Name	Folder
1	Main Program	
2	Entrance	Main
3	Feedback Form	Main
4	Complex View	Main
5	Orders	Main
6	Order Details	Main
7	Table View	Main

Figure 1: A screenshot of the uniPaaS application development framework.

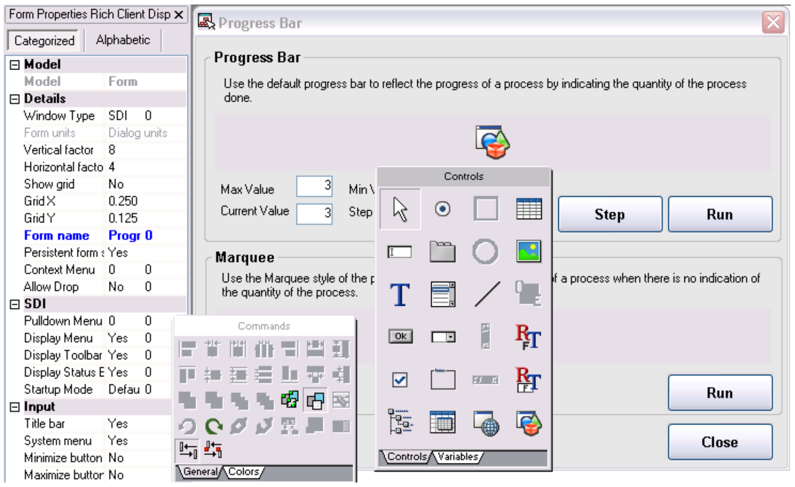
Figure 1 is a screenshot of the uniPaaS development environment. The major components of uniPaaS, as a 4th generation programming language are:

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 uniPaaS 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. uniPaaS has a form editor, where we can define the windows' attributes (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 a uniPaaS FormEntry. In the uniPaaS 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 uniPaaS form editor is shown in Figure 2.



Model		
Model	Form	
Details		
Window Type	SDI	0
Form units	Dialog units	
Vertical factor	8	
Horizontal facto	4	
Show grid	No	
Grid X	0.250	
Grid Y	0.125	
Form name	Progr 0	
Persistent form :	Yes	
Context Menu	0	0
Allow Drop	No	0
SDI		
Pulldown Menu	0	0
Display Menu	Yes	0
Display Toolbar	Yes	0
Display Status E	Yes	0
Startup Mode	Defau	0
Input		
Title bar	Yes	
System menu	Yes	
Minimize button	No	
Maximize buttor	No	

Use the default progress bar to reflect the progress of a process by indicating the quantity of the process done.

Max Value 3 Min \
Current Value 3 Step

Use the Marquee style of the p f a process when there is no indication of the quantity of the process.

Figure 2: A screenshot of the uniPaaS form editor.

3 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/uniPaaS. 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 tested application.

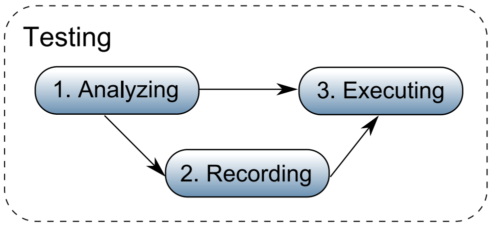


Figure 3: Main steps of automatic GUI testing of a Magic application.

We discuss these steps in the further sections in more detail.

3.1 Magic Static Analysis

During the analysis of a Magic application we extract information from the lowest level description of the application, from its source code. As the result of the analysis, a graph describing the structure of the program is created, which is called an Abstract Semantic Graph (ASG). The creation of this graph is based on the Columbus Magic Schema. The schema is defined by a UML class diagram which includes all the important elements of the language (see Figure 4).

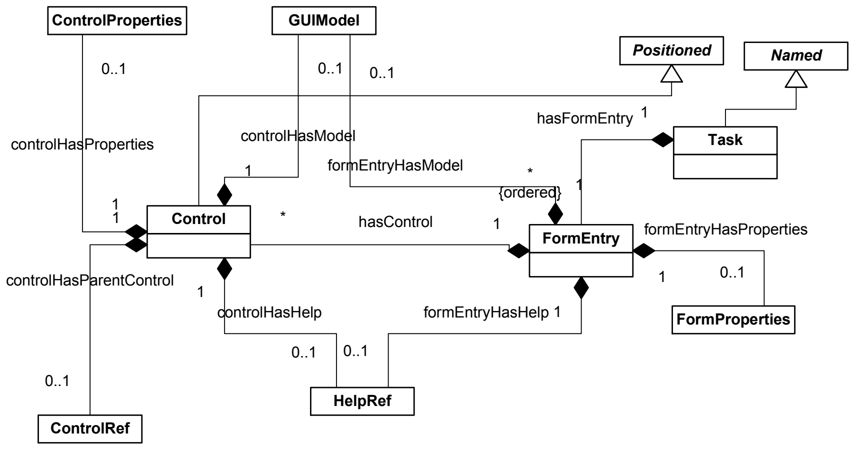


Figure 4: A simplified part of the Magic Schema describing Tasks, Forms, Controls and relations among them.

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 is a structured text file, but in the newer versions such as uniPaaS, this is an XML file. The older DOS-based Magic version did not include complex graphic elements, so do the new versions. Hence the Magic schema had to be created in a way that it contains all specialties of all Magic language versions.

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.

Figure 4 is a simplified part of the Magic Schema, which highlights the GUI elements important for test automation. The figure shows the relationship between the Magic Tasks and the GUI elements. A FormEntry corresponds to a graphic window. A FormEntry is part of a Task, and a Task can contain more than one FormEntries. The properties of a FormEntry (caption, size, etc.) are stored by the FormProperties node. With the hasControl edge we can get the GUI elements of a form. All GUI elements are described by Control nodes, and the properties of a GUI element (title, type, position, etc.) are stored by a ControlProperties node.

For further details about reverse engineering Magic applications please refer to our previous work [13].

3.2 Recording GUI Events

During the manual test of the graphical user interface of an application, the tester simply goes through certain steps of a test case (click somewhere, write some text in a text box, etc.). Then she/he verifies if the application produces the expected behavior. In order to replace this manual work with automatic processes, first we define a test script format in which we describe the GUI events that the program should execute. Manually writing such a script file is possible, but it can be difficult, because in a complex test case there are many complicated GUI events. Therefore, the Magic test automation tool must be able not only to playback these scripts, but to record them as well. Figure 5 illustrates the process of the recording.

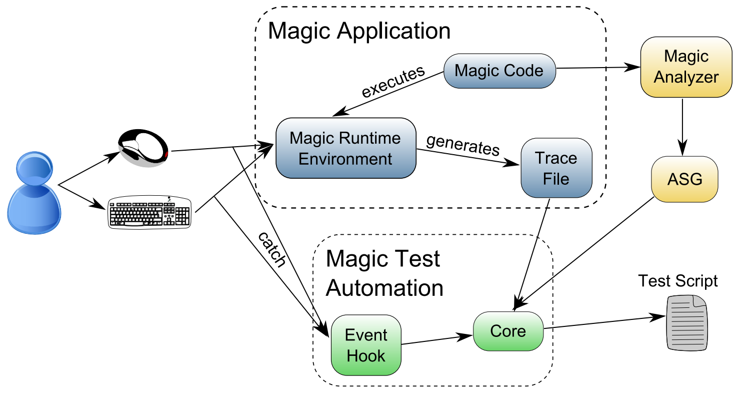
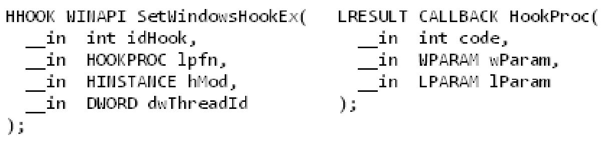


Figure 5: Recording GUI Events.

Before we record or execute test scripts, first we must analyze the application being tested to produce the ASG of it. This step has to be done only once for one version of the application, and once we have the ASG we can use it for all recordings and playbacks. Recording is based on the fact that Windows allows us to catch the mouse and keyboard events, and to obtain information about the GUI element on which an event occurred. The way of setting up the GUI event observation is that the Windows API contains a function named *SetWindowsHookEx* which takes the type of the observation as a parameter. We can not only observe the mouse and keyboard events, but we can observe the messages between windows and we can observe shell events too. This function takes a callback function as its parameter with a specific prototype (*HookProc*). When we call the SetWindowHookEx function, Windows will store the given callback function and when a mouse or keyboard event occurs, Windows will call our function and pass information to it about the actual event. Figure 6 shows the prototypes of the *SetWindowsHookEx* and the *HookProc* functions.



```
HHOOK WINAPI SetWindowsHookEx(   LRESULT CALLBACK HookProc(
  __in  int idHook,                __in  int code,
  __in  HOOKPROC lpfn,             __in  WPARAM wParam,
  __in  HINSTANCE hMod,            __in  LPARAM lParam
  __in  DWORD dwThreadId         );
);
```

Figure 6: The prototypes of the SetWindowsHookEx and the HookProc functions.

The first step of recording a test script in a Magic application is to execute the application under test in the Magic Runtime Environment (MRE). This environment can be configured to generate a trace file which contains dynamic runtime information. The trace file tells us which tasks started or ended and when. We use this information to determine which task was running and which form was active at the moment when a user event occurred.

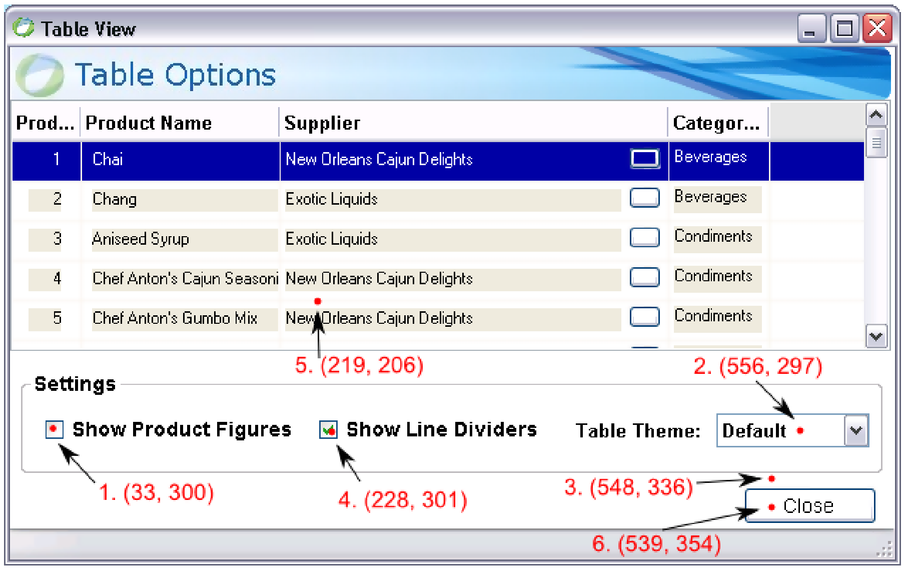
The observation of user actions is done by a separate component of the Magic Test Automation tool, called the Event Hook DLL. A tester starts the Magic application in the MRE and starts the Magic Test Automation tool where she/he chooses the record mode. From this point the Event Hook DLL is activated and monitors all GUI events of the application being tested. The tester can start using his mouse or keyboard and Windows will send all the events to the Event Hook DLL, which will pass them to the core of the Test Automation tool. The Test Automation tool will monitor the trace file and as it receives a GUI event it will determine the Windows control on which the event occurred. Then, it determines the ASG node to whom the control belongs to. The linking between the GUI event and the corresponding Windows control is also done through the Windows API, because the GUI event contains the handle of the control on which the event occurred. We can use this handle to obtain information (size, position, text, parent window) about the control for example with the *GetWindowRect*, *GetWindowText* and *GetParent* functions.

As soon as we obtain all the necessary information about the Windows control, our next task is to connect it to the corresponding node of the ASG. As we mentioned before, a FormEntry is always linked to a Task. So we read from the trace file that at the moment when the GUI event occurred which Task was active and then we go through the ASG and search for these Task nodes. Once we found these nodes we query from the ASG the FormEntry which belongs to this Task. Windows stores the controls of a window in a tree structure, and so does the Magic ASG. So we start walking through the tree of Windows' controls and the tree of the controls in the ASG starting from the active FormEntry. We compare the properties of Windows' control and the control of the ASG such as their size, position and text. With this method we can find the Control node in the ASG that belongs to the given Windows control.

To identify a control in the ASG, we use its scoped unique identifier. All nodes in the ASG have a unique scoped identifier which uses the type and name of its parent node and a separation sequence. For example the *Pj Test::Pr Main::Fe Settings::Ct Close* identifier belongs to a Control (Ct) named Close, which is on a FormEntry (Fe) named Settings, and this FormEntry is contained in a Program (Pr) named Main, which is in the Project (Pj) named Test. In most cases, this identifier remains unique between different versions of the application under question.

The last step of recording a test script is to save the gathered information into a script file when the tester stops recording his actions. When we save the script file we can choose between two formats:

- The Magic Test Automation tool's own format. This script is less extensible, but the Test Automation tool manages this type of script directly.
- A Python script format. If we choose this format then the Test Automation tool manages the script via a Python interpreter. One can use all features of the Python programming language to write more advanced scripts here.



Prod...	Product Name	Supplier		Categor...
1	Chai	New Orleans Cajun Delights	☐	Beverages
2	Chang	Exotic Liquids	☐	Beverages
3	Aniseed Syrup	Exotic Liquids	☐	Condiments
4	Chef Anton's Cajun Seasoni	New Orleans Cajun Delights	☐	Condiments
5	Chef Anton's Gumbo Mix	New Orleans Cajun Delights	☐	Condiments

Settings

Show Product Figures
✓ Show Line Dividers
Table Theme: Default

Figure 7: An example window of a uniPaaS application with example steps for testing its GUI.

Figure 7 shows a window of a uniPaaS demo application which demonstrates the way of using table controls. In the figure we illustrated the possible steps that a tester would perform when testing this window. We recorded the illustrated steps with the Magic Test Automation tool and saved the script in Python format. Figure 8 shows the resulting Python script.

```python
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)
```

Figure 8: A layout independent Python script for the steps in Figure 7.

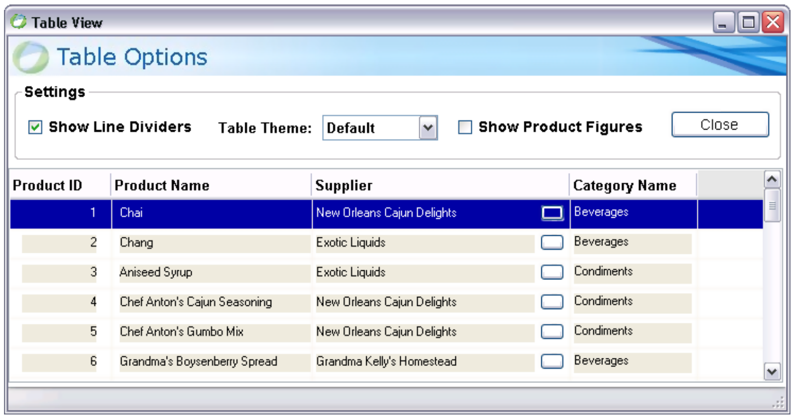
It can be seen that the Magic Test Automation tool connected the Magic code with the ASG and generated the script using the obtained identifiers. The Magic Test Automation tool can also be used with non Magic applications, so when the ASG is not available we can use it as a generic test automation tool and generate the GUI events in the script file by coordinates. We performed the previous test events with this method and saved the Python script again. Figure 9 shows this Python script.

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 10. The application works as before, but the controls are 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.

```
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)
```

Figure 9: A position based Python script for the steps in Figure 7.



Settings

☑ Show Line Dividers Table Theme: Default ☐ Show Product Figures

Product ID	Product Name	Supplier		Category Name
1	Chai	New Orleans Cajun Delights	☐	Beverages
2	Chang	Exotic Liquids	☐	Beverages
3	Aniseed Syrup	Exotic Liquids	☐	Condiments
4	Chef Anton's Cajun Seasoning	New Orleans Cajun Delights	☐	Condiments
5	Chef Anton's Gumbo Mix	New Orleans Cajun Delights	☐	Condiments
6	Grandma's Boysenberry Spread	Grandma Kelly's Homestead	☐	Beverages

Figure 10: A rearranged window of the example uniPaaS application (see Figure 7).

3.3 Playback Recorded GUI Events

3.3.1 Executing Events

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 GUI elements. During the playback we must execute the Magic code 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 11.)

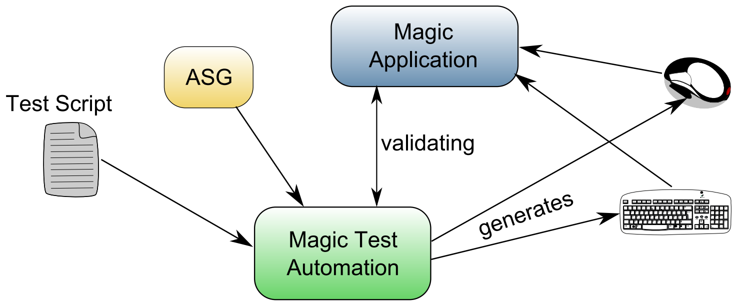


Figure 11: Running Recorded GUI Events.

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 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 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. We have to know the exact location of where to click. The Magic Test Automation tool generates script files where we store it as a relative position to the identified Magic control.

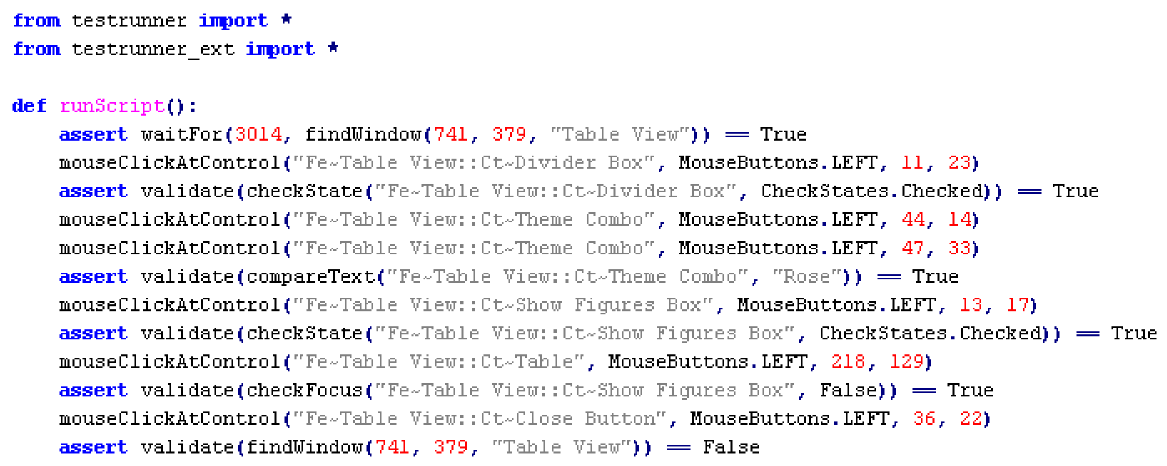
After we obtained the handle of the GUI element on which we have to perform a keyboard or mouse event and we have the proper relative coordinates we walk up the Windows control tree and summarize these relative coordinates until we get to the top of the tree. As we have calculated the absolute coordinates we use the Windows API to generate the keyboard or mouse event and send them to the application.

These possibilities are still not enough for simulating a manual GUI testing task, since there are many applications in which we for example enter a text in a text box and during the typing a background search process gets executed to display some result in another GUI element. The next GUI action can be performed only after the search function is completed. With manual testing this is not a problem, because a human can notice when the search process ended. With a test automation tool we can support it by inserting delays for time intervals or until certain events. For instance, one can make the automation tool to wait for exactly 10 seconds or to wait until a GUI element loses or gets the focus or to wait until a GUI element is activated.

3.3.2 Evaluating an execution

Evaluating the results of an execution is also very important. Some steps of this 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 title of a specified text.
- 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.



```
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
```

Figure 12: Examples for validations in a Python script.

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 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 12 illustrates the Python script shown in Figure 8, 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 Table-View::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.

3.4 Drawbacks of the technique

We introduced many benefits of our layout-independent technique out of which the most important was that the recorded test script will be ignorant to minor changes in the GUI. However there are some important drawbacks as well 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.

4 Related Work

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 [3, 4, 5, 10, 15]. Here we elaborate on the literature and on related tools focusing on those that are orthogonal to our work.

Testing automation frameworks are usually divided into 5 generations [8, 9]. First 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. Third 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 data-driven testing [5, 16]. 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 [1, 2, 5]. 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 or developers [7].

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 automatic GUI testing in other approaches, e.g. for generating test scripts [6, 11, 12, 14].

There are a number of automatic GUI testing tools available for software engineers. Just to mention some examples, Selenium[1] 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[2] is a product of SmartBear Software Inc, which is also widely used in the world of Magic. HP offers solutions for automatic GUI testing via its Quality Center Software, and its Quick Test Professional (QTP) tool is also a widely used tool for applications written in 4GL. Microsoft also provides automated GUI testing solutions for instance via GUI Automation of the .NET Framework. An example for a keyword-driven test automation tool is TestArchitect[3] developed by LogiGear Inc.

5 Conclusions and Future Work

In our paper we presented an approach for layout independent automatic GUI testing 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/uniPaaS, 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.

[1]http://seleniumhq.org/

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

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

Using GUI information stored in the source code during automatic GUI testing is a novel approach for Magic 4GL, to our best knowledge. 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-1.1.2-07/1-2008-0007, GOP-1.2.1-08-2009-0005, OTKA K-73688, and by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences.

References

[1] Hans Buwalda. Automated testing with action words, abandoning record and playback. In *Proceedings of the EuroStar Conference*, 1996.

[2] 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.

[3] 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.

[4] Elfriede Dustin, Jeff Rashka, and John Paul. *Automated software testing: introduction, management, and performance*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[5] Mark Fewster and Dorothy Graham. *Software test automation: effective use of test execution tools*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1999.

[6] 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.

[7] Jeff Hinz and Martin Gijsen. Fifth generation scriptless and advanced test automation technologies. http://www.testars.com/docs/5GTA.pdf (accessed on 2011 August), 2009.

[8] Cem Kaner. Architectures of test automation. http://www.kaner.com/pdfs/testarch.pdf (accessed on 2011 August), 2000.

[9] Edward Kit. Integrated effective test design and automation software development. *Software Development online*, feb 1999.

[10] Kanglin Li and Menggi Wu. *Effective GUI Test Automation*. SYBEX Inc., Alameda, CA, USA, 2005.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] Bruce Posey. *Just Enough Software Test Automation*. Prentice Hall PTR, 2002.

[16] 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.