Acta Cybernetica **22** (2015) 81–99.

Structural Information Aided Automated Test Method for Magic 4GL*

Ferenc Horváth† and Tamás Gergely†

Abstract

Nowadays testing data intensive, GUI enhanced applications properly on an easily maintainable way has become a more crucial part of the application life cycle. There are many evolving technologies to support automatized GUI testing in various environments. However there are hardly any methods that support 4GLs, especially Magic.

Fortunately, the characteristics of the 4^{th} generation of programming languages – like explicit definition of the relations between GUI elements and data – eliminate most of the problems raised during testing the GUI in the case of 3GLs. By utilizing these advantages we were able to develop a generalized testing method that supports 4GLs, and as a proof of concept a system for testing Magic xpa applications was built.

In this paper, a generalized testing method for 4GLs, our path and script generator algorithms, and their implementations for Magic xpa applications are presented. In addition, the cooperation of these components with existing solutions is demonstrated, and a test method that has been competed by the application of our tools (and which is an instantiation of the generalized method) is introduced as a possible use of the results.

Keywords: Magic, 4GL, automated testing, CFG, ASG, path generation, test script generation

1 Introduction

Nowadays the well-designed, nice-looking, and transparent GUI frontend that satisfies customers' usability requirements is a minimal requirement for most software. On the other hand, more and more software are developed through some non-linear development models (such as extreme programming), which results in frequent changes. In such environments cheap and easily maintainable testing becomes a more crucial part of the application development life cycle.

*This research was supported by the Hungarian national grant **GOP-1.1.1-11-2011-0039**.

†University of Szeged, Department of Software Engineering, E-mail: {hferenc,gertom}@inf.u-szeged.hu

There are many evolving technologies to support automatized GUI testing for data-intensive web or desktop applications in different platforms and environments. The first GUI tester tools were layout dependent and more like simple recorder and player of low-level user actions (mouse move, keystroke, etc.) [11, 13]. The main drawbacks of this solution were the poor verifiability and the sensitiveness to even the smallest modifications of the GUI layout. Also, these tools provided no real support for automation of other test activities than test execution.

Then, the tools got more sophisticated by supporting the identification of GUI elements in layout independent ways. These identification methods were usually related to the programming language the software were written in or to the execution environment [11, 13]. This identification allowed not only to record but also to generate the test cases, as the high-level interactions on the precisely identified GUI elements were captured. However, identifying the GUI elements and the possible actions usually required the source code and very complex methods to be used, and the automation of the test case generation were limited by the lack of direct connection between the GUI elements and the manipulated data (which resulted in the use of heuristic approaches even in the simplest cases).

Fortunately, the characteristics of the 4th generation of the programming languages eliminate most of these problems. Usually, the connection between the GUI elements and the data manipulated by the actual program is direct or follows well-determined pattern rules in software created using 4GLs.

For example, the most important elements of Magic 4GL 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 linked to forms, menus, help screens, and other GUI elements. Table elements are directly connected to database records, input or output fields of forms, and programs. The description of possible values are descriptive, instead of the algorithmic check implemented in 3GLs. Thus, Magic provides information about the data in a much straightforward way than 3$^{\text{rd}}$ generation languages.

Using this feature the developers can reorganize related data elements into small objects, and link these derived fields to database tables, graphic elements, or other contexts.

Therefore, the 'source code' of this kind of applications contains much more details about the GUI elements than the information that can be gained through complicated heuristic methods in case of the code written in 3GL. Despite of this advance, it is still hard to find any methods that directly supports test case generation of 4GL applications.

In this research, our goal was to provide a semi-automated testing toolchain for Magic xpa applications. To achieve this goal, we defined a general automated test process that is feasible for 4GLs, and implemented it in the Magic xpa environment. This implementation included several automation tools that supported different steps of the process. It is based on the work of Dévai et *al.* [5] to gain usable behavioral information about the program under test and Fritsi et al. [8] to run our assembled scripts. These two components have been connected and completed by developing a path and a script generator that process behavioral information and

assemble executable scripts for the test runner.

Path generation is a very complex task because one has to face obstacles like the exponential relation between the number of branches and the number of the possible paths, or the presence of loops in every real program which induces the number of paths to converge to infinity [3]. Therefore scalable algorithms are needed.

Our path generation strategy is based on well known algorithms like breadth-first and depth-first search. Considering the time and space complexities, these algorithms perform well enough to allow us to build upon them while developing our advanced strategies. However, we have to introduce some modifications to meet the requirements of our domain. These modifications affect the generation process in numerous ways. For example, an interface have been created that can be used to build a business model for the program being tested. This model defines constrains which are used by the generator algorithm as a form of selection method to narrow down the set of used program subcomponents (e.g. tasks which represent main functional components in Magic 4GL). This way the set of the generated paths can be reduced significantly and contains only the most relevant ones.

The path generation strategy is a crucial part of the test script generation. After the selection of the appropriate path set, the gained information have to be converted into interpretable test scripts. The number of the generated paths has a strong influence on the final number of test scripts as well as on which strategy is used to determine how values are selected and permuted among different variables.

The co-domain for each variable is based on its type and control flow context. After all the influencing data properties (D/U information) are extracted, the co-domain of a variable have to be narrowed down based on the influencing expressions and statements which appear on the corresponding path. Definition/use (D/U) information [2] expresses the data flow relationships between expressions that modify a certain data item (i.e. definitions) and expressions that reference that data item without modifying it (i.e. uses). The D/U representation also provides information about the data related program elements (e.g. GUI inputs, database fields).

Choosing a good strategy for solving the aforementioned problems is essential in order to get an accurate and usable test script set as the output of the described script generator algorithm. Only a small enough test set size can be manageable even if the pre-configuration of the Magic xpa program and the execution of the scripts are totally automated.

In this paper, we present our generalized testing method for 4GLs, its instantiation for Magic xpa programs, and our path and script generator algorithms. In addition, we describe the implementation of the path and script generator algorithms for Magic xpa applications, and use these in the implementation of the Magic test process.

The rest of the paper is organized as follows. In Section 2 an overview on the related work is given. Section 3 presents the generalized test method, while in Section 4 the details and the implementation of the Magic test method are described. Finally, our achievements are summarized in Section 5.

2 Related work

There are more methods described in the literature that generates test cases or test data input for business models. Yan et *al.* [24], Yuan et *al.* [25], and Bakota et *al.* [3] presented methods for BPEL, but the approaches can be utilized for any high-level specifications annotated by test engineers.

Our approach uses business models to control the number of test cases, but the main source it relies on is the "source code" of the software. To generate concrete input values, our implementation also employs the well-known equivalence partitioning [18]. Also, we used the idea of test frames as the script generation module populates script frames with the generated input values to assemble the final test scripts.

Beside business models, there are other high level approaches available for test case generation. For example, Kim et *al.* [12] and Mingsong et *al.* [16] used UML diagrams as the source of the generation process. Their presented methods are also feature data flow and control flow analysis techniques.

Control flow graph (CFG) is a widely used representation of programs. It is used, for example, by analyzer and checker tools [6, 7, 14, 10], test input generator tools [21], or program slicing [19]. It is also applied in complex solutions like the timing model generator presented by Wenzel et *al.* [23] and the run-time error predictor tool presented by Weimer et *al.* [22]. However, while CFG is widely used in 3GL environments, CFG solutions for 4GLs are really limited: a few flow analysis techniques were published for the ABAP (the language of SAP) by Abramowicz et *al.* [1] and Vanhatalo et *al.* [20], and Dévai et *al.* [5] implemented a solution for generating CFG and D/U data of Magic programs. We rely one the latter results.

The implementation of our method for Magic xpa is based on several existing tools. First of all, we use a reverse engineering tool set to access the source files of Magic xpa applications, which was implemented in a previous work of Nagy et *al.* [17]. This toolset was able to analyze magic programs and produce the ASG. The test case generator tool produces test scripts that can be executed by the Magic test execution tool of Fritsi et *al.* [8].

3 The general test method

Several parts of software testing can be automated. Automation is most often done in test execution: prepared test cases are executed by some frameworks that executes and evaluates the test cases.

Despite the fact that automated test generation is a rapidly evolving field in software engineering [9, 26], these test cases are usually manually created: a test analyst designs the test input and preconditions, defines the postconditions and the expected output; then someone implements the test case according to the software interfaces and the test environment. A full automation of an adequate test case generation is infeasible. For example, even though Rodin [4] offers a method for complete formalization, the formalization itself still requires human effort. And

using the most precise documentation, the source code itself alone as the base of the test cases is also pointless as we cannot determine reliable and valid expected output from it.

Although full automation is not possible, semi-automatic methods can ease the work of the testers. The quality of the testing can be measured through different metrics. The most well known such metrics are the various code coverage metrics. In short, these metrics show in what degree the software code is examined through the test executions. White-box testing methodologies are based on code coverage and aims the definition of test cases that will execute previously not covered code segments.

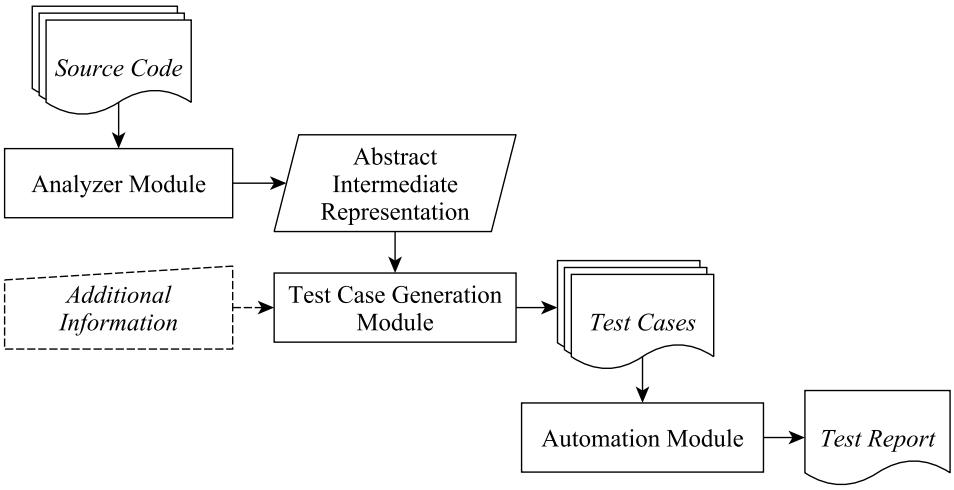

Figure 1: The flowchart of the generalized test process

A part of this work can be automated as can be seen on Figure 1. Given the source code (or some detailed representation of the program), we can examine it by an analyzer tool that builds a usable internal representation of the program. This representation contains the control and data flow of the program. (Note, that while computing control and data flow for imperative programs has many working solutions, doing so for event-driven programs and 4GLs is a much more difficult. One reason for it is that while in 3GLs instructions and their execution order are described, 4GLs usually employ higher level event-driven models and hides implementation details.)

Using the control flow, we can extract possible execution paths whose number can be infinite for real programs. This phenomenon requires the definition of some constraints that prevents us from trying to generate new and new paths endlessly. Possible solutions include limiting the number of iterations a loop is allowed to execute or utilize high level business process descriptions to generate feasible or filter unfeasible paths [3]. The resulting paths contain many decision points, each with an assigned outcome value, which will add constraints to the variables used

in the decisions. In 4GLs, most of the variables are directly connected with some inputs (e.g. GUI elements or database fields). Using the determined constraints, we can automatically assign values to these inputs that will result in the execution of the desired path of the software. But defining the test input is far from a working test case. From the source code itself we cannot derive the expected outcome, which must be determined manually using the specifications. This part of the work cannot be automated.

After the test cases are ready, test scripts can be automatically generated (utilizing the beneficial attributes of 4GLs), and the generated scripts can be executed using some automated execution frameworks, which will produce the test report.

4 The Magic test process

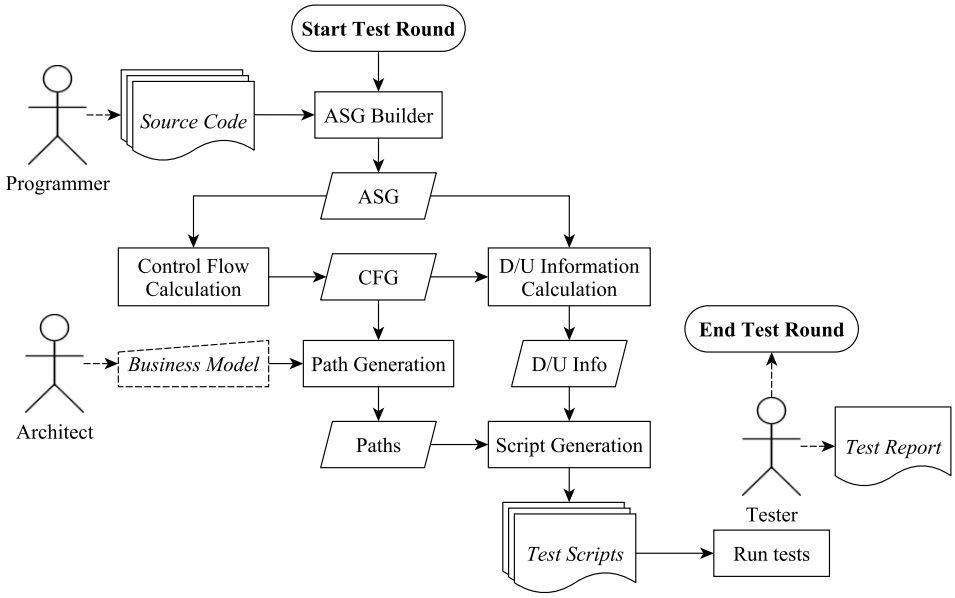

Figure 2: The flowchart of the Magic test process

The Magic test process is an instantiation of the generalized process. The refined and specialized process can be seen in Figure 2.

The process starts with the application of the Magic2ASG tool [17] on the source files of the program under test. This tool extracts important information about the actual application by examining its XML based source code. After parsing the relevant XML files the tool builds an abstract semantic graph (ASG) which is an intermediate representation of the actual Magic xpa application. This representation captures the structure of the application. It identifies the relevant language entities (e.g. data tables, columns, programs, tasks, logic units, etc.) and their

parent-child relations that determine the main structure of the application, like packages, classes and methods define the structure of a Java application. Finally, the ASG is stored in binary format for the subsequent steps of the process.

In the next steps, algorithms developed by Dévai et *al.* [5] are used to compute control flow graph (CFG) and definition/use (D/U) information is applied. These algorithms extract important details about the possible activation order of the program elements, about the different paths on which the data is processed, and about data-flow related elements (variables, parameters, GUI input/output elements).

The next step of the testing process is the path generation, which takes the CFG and extracts different feasible execution paths from it. In our approach, a high level business model is used to help filter out those paths that are allowed by the implementation but make no sense in real-life use of the software. This model can usually be derived from the functional specifications of the software. Although it cannot be expected that such a business model is always available in a proper format or with the required content, many of the 4GLs includes the ability to define high level workflows. In Magic, due to the characteristics of the language, the structure of a certain application reflects to the business level workflow in which the program is used. Therefore, the generation of this model is not very expensive, and even if it is, such an extra effort usually pays off due to the reduction of the number of test cases.

Then, test cases are generated by combining the extracted paths with the available D/U information using various methods for data binding, and these test cases are exported as test scripts.

After the path and the script generation stages are completed the final phase of the method is to run the scripts that were assembled. In our implementation, the scripts are executed by the test runner tool developed by Fritsi et *al.* [8].

In the followings, the test generation process is detailed in Section 4.1, and Section 4.1 describes the test case and script generation phase.

4.1 Path generation

The path generation relies heavily on the additional information that can be extracted from the business model, which is usually derived from the specification of the actual software. In our case the business model is a high-level declaration of the possible use cases and work flows of the application. The elements of this high level business model are linked with the items of the abstract representation of the Magic programs. (In our implementation the connection is defined in a separate XML file as can be seen in the example Listing 1, but this was an arbitrary design decision.)

The model itself and the connection between it and the ASG can be defined according to the schema in Figure 3. The elements of this schema are the following.

- *model* – It is the root node of the schema. It has to contain at least one *scenario* element.

- *scenario* – This item is the equivalent of the various scenarios, use cases of the business processes. It is composed of a sequence of various functions and it is assigned with an ASG node of the appropriate Magic program component where the scenario, and thus the path generation starts from (i.e. entry point).
- *function* – It represents the functions that "implement" some part of the actual scenario. In other words, a function is a set of Magic program components that play a role in a given work-flow.
- *component* – This is the basic element of the model, a part of a function. Its one and only mandatory attribute is called *nodeIdRef*, and holds the identifier of the ASG node of applicable Magic program elements thus provides the connection between the model and the abstrace representation.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE filter SYSTEM "model.dtd">
<model>
  <scenario entryPoint="3015" name="Print Invoices">
    <function name="Show Invoices Manager">
      <component nodeIdRef="3293" /> <!-- manager task -->
    </function>
    <function name="List & Select Customers">
      <component nodeIdRef="3316" /> <!-- lister task -->
    </function>
    <function name="Select Orders">
      <component nodeIdRef="6698" /> <!-- min. order id selector -->
      <component nodeIdRef="6700" /> <!-- max. order id selector -->
    </function>
    <function name="Generate Invoices">
      <component nodeIdRef="3369" /> <!-- calculator task -->
      <component nodeIdRef="3323" /> <!-- pdf generator task -->
    </function>
    <function name="Close Invoices Manager">
      <component nodeIdRef="6702" /> <!-- close button -->
    </function>
  </scenario>
</model>
```

Listing 1: An XML file of an example business model.

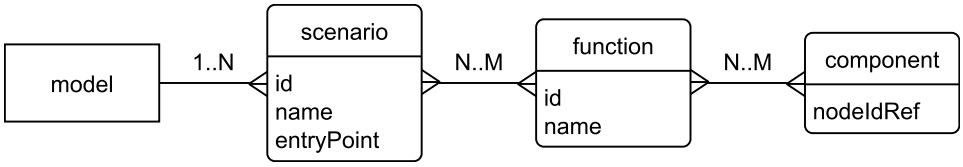

Figure 3: The schema of our business model.

An example business model instance can be seen in Listing 1. This example was created based on an application called RIA Demo [15], which is a demo program

that is bundled with Magic xpa Studio, the IDE for Magic. (This particular model represents only a part of the whole demo application.) The model has one scenario in which invoices are generated into a PDF file. This scenario is broken down to five different functions that must be performed sequentially. The first function is responsible for displaying the invoice generator. In the second one, the user can list customers and select one of them whose orders will be processed. After that, the user can select the range of order items to be exported using the third function. The actual invoice generation is done in the next function which is implemented by two components: one which calculates the totals and an other which generates the PDF file. Finally, the last simple function is responsible for closing the dialog. Both the third and fourth functions have two components, while all the others have only one component. These components are connected to the nodes in the ASG by giving their node identifiers 3293, 3316, 6698 and 6700, 3369 and 3323, and 6702. The entry point of the scenario is a special Magic task that starts the whole demo and is referenced by the identifier 3015 in the ASG.

Two similar yet different approaches have been developed to control the beginning of the path generation process based on the extra information provided by the business model. Each of these methods introduce constrains to the algorithm that can either be used separately or together as well. They are needed because generating all the possible execution paths would either be impossible due to loops in the CFG or be ineffective if the infeasible paths must be dropped later. So, these constrains stand for some kind of guidelines or restrictions during the beginning of the path generation process. Consequently, less effort is needed and simple (but sightly modified) graph algorithms become applicable to finish path generation.

The first method defines the main search space in the CFG by determining a template for the path generation algorithm. The template is created by specifying intermediate start and end points based on the functions of a scenario. The template-based method is detailed in Section 4.1.1. The other method has an effect on the path building process too, however, it was designed to allow the definition of dependencies that can restrict the generation process on-the-fly or can be used to filter the generated temporal paths before they are persisted. The filtering method is described in Section 4.1.2. The finishing step of the path generation is presented in Section 4.1.3.

4.1.1 Path template

In this case the path generation is based on Algorithm 1. The paths are generated as sequences of smaller sections, according to the functions of the scenarios. The first section of each path is generated by finding the shortest path between the entry point of the actual scenario and a component of the first function. Then the algorithm continues by extending the current path with the next path segment between the end of the actual path and the next function.

This algorithm has some special features. First, it requires all components of the scenario to be reachable from the entry point, otherwise the algorithm finds no paths. Then, the algorithm works by populating the set of generated paths, while

Algorithm 1 Template based path generation

```
P = ∅                                              The set of the generated paths
for all s ∈ model.scenarios do
  u = spath(s.entryPoint, s.functions[0])          Shortest path
  for i = 1 to s.functions.length() do
    u = u + path(u.tail(), s.functions[i])         Normal path
  end for
  P = P + u
end for
```

processing the scenarios one after the other.

Considering the previous example in Listing 1, the path generation starts from the entry point *3015*, which is a Magic task called *Entrance* in the CFG, and builds a path that reaches the invoice generator task *3293*. Next, the algorithm tries to find the second section of the path. This search is considered successful if the task *3316*, which is responsible for the listing and selection of customers is found. And so forth through nodes *6698* and *6700* of the third, then the nodes *3369* and *3323* of the fourth function, while the end of the path does not reach the one and only component (*6702*) of the last function.

4.1.2 Path dependencies and filtering

In this case, the business model is used to create dependencies that a filtering algorithm can manage while the path generation process is running. During the execution of the path generator or before a path can be persisted, a filtering module enforces that the dependency structure which has been built for each scenario is not violated by the actual state of the generator.

Dependencies are built by connecting the components of each subsequent functions in the actual scenario, starting from the last and moving towards the first. Specifically, the components of the n^{th} function have all the components of the $(n-1)^{\text{th}}$ function as a dependency. So, the algorithm must manage the components of the first function separately.

Algorithm 2 Filtering algorithm

```
function Filter(node, path) returns boolean
  for all d ∈ node.dependencies do
    if not contains(path, d) then
      return false
    end if
  end for
  return true
end function
```

An important part of the filtering module can be seen on Algorithm 2. Based

on this feature the generator can make decisions in special situations during its execution. On the one hand, it can decide whether it has to terminate the generation at an actual point and backtrack. This can happen when the algorithm reaches a part of the CFG where the constrains declared by the dependencies are violated. On the other hand, when the post generation filtering method is used the algorithm can drop paths that do not fulfil the constrains.

A dependency structure, which was built based on an example model that is presented in Listing 1, can be seen in Table 1. This structure is utilized, for example, when the path generator algorithm tries to find the third section of an actual path which is between the *customer selection* and the *order selection* steps of the example scenario. Because of the algorithm drops all the paths on which the *order identifier selector* is not preceded by the *customer selector* task. Considering how many ways the elements of even a simple GUI can possibly be enumerated, this technique saves a lot of effort during the path generation process by locally eliminating unusable sections.

Node	**Dependencies**
3293	*3015*
3316	3293
6698	3316
6700	3316
3369	6698, 6700
3323	6698, 6700
6702	3369, 3323

Table 1: Dependencies built based on the example model in Listing 1.

4.1.3 The base algorithm

Aside from the constraints handling portions, the path generator algorithm features a fairly simple design. The final algorithm consist of only two phases and its core is based on a customized depth-first search algorithm. These customizations are the following.

- Nodes which were already covered by one of the sub spanning trees of the previously visited children of a node can be visited again while visiting an other child of the same node. In other words, when the algorithm starts to backtrack nodes are removed from the actual path, so the algorithm can visit them again while traversing an other possible path.
- Loop edges are dropped. This compromise had to be made because the nature of the original algorithm implies that the traversal of a loop edge increases the number of possible paths exponentially, meanwhile the coverage level remains almost the same. It results in paths that iterate a loop zero or one time only.

The first of the two phases is the initialization phase where the algorithm categorizes the edges of the CFG and calculates priorities for each of them based on their category. The categories i.e. normal, forward, cross and back are given by the labelling of the depth-first search algorithm.

When each edge is categorized the algorithm adjusts the priorities in a way that prefers the normal or forward edges over the other ones. This is necessary because with these settings the algorithm tends to find the longest possible paths first. Consequently the number of generated paths can be reduced.

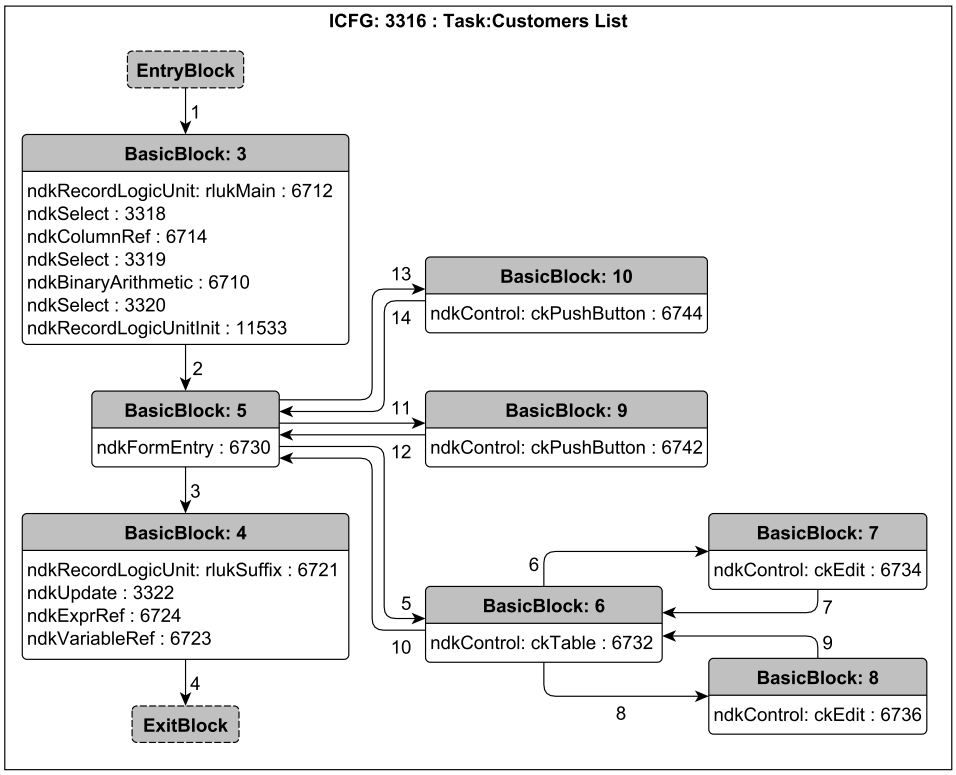

Figure 4: The CFG which represents the second function of the example scenario.

A part of the graphic visualization of the complete control flow graph can be seen in Figure 4. This part which represents the second function of the example scenario is consists of an entry- and an exit block, 8 basic blocks and 14 edges. The former are presented by rectangles that contain the consequent ASG nodes and their identifiers. The latter are presented by arrows indicating the direction of the control flow.

Note, that the edges are labelled by numbers. These numbers are the indexes that the depth first search algorithm gives to the edges during the initialization phase of the algorithms. In other words, this is the visit order of the edges. The

edges 7, 9, 10, 12, and 14 are labelled as *back* edges, all the others are *normal* edges. There are no *forward* or *cross* edges in this example.

Path	**Basic blocks**
1	3-5-6-7-6-8-6-5-9-5-10-5-4
2	3-5-6-7-6-8-6-5-9-5-4
3	3-5-6-7-6-8-6-5-4
4	3-5-6-7-6-5-4
5	3-5-4

Table 2: Paths built based on the CFG in Figure 4.

The real path generation is done during the second phase. The algorithm decides which edge it will visit next based on mainly the priorities. However, the algorithm has to consider the category of the edges as well, because each category requires different actions to be taken.

If we find a *back* edge in the CFG it means that the graph contains a loop, so these edges are handled specially. The algorithm remembers them to prevent itself from visiting infinite loops. Then the generation process is restarted recursively to allow the generator to continue its work after a loop has been found. The algorithm handles this situation as if it had been started from the node where the loop rejoins the path, besides it stores the previous path and continues the generation from its end. Thus paths are generated with and without the loop and its subsequent nodes in parallel.

The *normal*, *forward* and *cross* edges are handled similarly as the typical actions of the depth-first search are taken. However, there is a difference between the call and control edges, because for the handling of call edges further calculations are needed.

The call and control types are built-in types of the CFG, so this categorization is given explicitly, yet further classification have to be done on the call edges: those call edges that point from an *Exit* block to an other block will be labelled as return edges.

The general rule is that a call from a basic block must return to a child of the caller block itself, where the child relation is determined only by considering the control edges. So, when the algorithm reaches an *Exit* block it determines the next block by calculating the intersection of the return sites of the actual block and the child nodes of the last caller.

Although it is uncommon, the generation process can be started from a block that has incoming call edges. This means that, the algorithm will not know the last caller when it tries to do the calculation mentioned above. So, in this case the generation continues with all the possible return sites.

Special operation takes effect also when a node in the CFG has call and control edges too. This is possible because during the build process of the CFG some fake control edges are inserted between call and return sites in order to keep coherence

in the absence of call edges. Therefore, real paths can be generated only if we drop the control edges and continue the process on the available call edges.

4.2 Script generation

In the following the capabilities of the test script and input generator modules, and the role of the input generator module in the script generation process are presented. In addition, it is briefly described how the input generation process handles the different types of control elements of Magic xpa.

The script generation takes place in the test process after the path generation step. Its input is a set of paths provided by the path generator, and it uses the D/U information for the input generation. The first task of the generator algorithm is to select the appropriate Magic xpa form components (e.g. push buttons, menus, edit boxes) from each path separately. The algorithm handles two types of components.

On the one hand, there are simple Magic xpa controls (e.g. lists, check boxes, buttons) which can be covered by simple script events (e.g. clicks, mouse moves, short key combinations). The generation of these script events is the responsibility of the script generation module. The details of this part of the generation module will be described below in Section 4.2.1.

On the other hand, there are some form elements (e.g. edit boxes) that allow the user to enter different values. This is where the features of the input generator module are utilized. First, the related variable has to be found for each form element by the help of the data contained in the ASG.

Then, by searching for the instructions and expressions which have an effect on the actual variable during the execution the input generator narrows down the range of the possible values of the variable based on the available D/U information. How the generator deals with the different type of variables is going to be the topic of Section 4.2.2.

After the set of possible values have been specified for each variable the final value can be selected based on these intervals using different algorithms. Finally, the chosen value is returned to the script generator which then determines how the actual value will be represented in the script i.e. it translates the value to script events. The resulting sequence of events serves as the input of the Magic test runner tool which is responsible for controlling the playback of the scripts.

4.2.1 Control element processing

In case of Magic xpa programs that have graphical user interface the data is displayed on the so called forms and the user can manipulate the data through form elements called controls. The next section elaborates on how the script generator module handles the most important control elements of the Magic xpa framework.

The purpose of the *Menu* elements is to provide an interface for executing different operations. The two subtype of menus can trigger events or start programs. Because menu elements do not take any input data it is the responsibility of the

script generator to create the suitable script events for mimicking the appropriate navigation flow.

In terms of input generation *Edit box*es are the most complex form elements. The value that will be inserted is based on the variable related to the actual edit box. Therefore the input generator module must be utilized. After the input value has been selected various script events are generated which will do the following:

1. Select the edit box using the mouse based on the coordinates that can be retrieved from the ASG.
2. Clear the edit box if needed.
3. Type in the value that was calculated by the input generator.

Like the first type of menu elements, *Button*s are used for triggering events. The generation of their script is very similar to the menus as well. The script generator creates only a mouse click script event together with some mouse move events based on the position information stored in the ASG.

Usually *List* is used to show results of queries. In this case the script generator works almost identical to the case of the *Button*s except that it generates key stroke events instead of mouse script event.

*List box*es and *Combo box*es display a set of selectable items. The main task of the script in this case is to activate the drop-down function and navigate to the appropriate item based on the information available.

4.2.2 Variable processing

A main feature of the variables is that they have a mandatory format attribute which is called picture in Magic xpa terminology. This attribute is usually used to provide different appearance for each type of variables. In addition, by using a special set of characters one can define constrains which can determine what kind of data can be stored in a variable. Therefore, as well as various instructions and expressions the picture attribute can also affect the possible values of a variable.

In the Magic xpa framework a so called *Alpha* type is responsible for the handling of strings. In terms of the input generation it is a relatively unobstructed type, so the biggest challenge here is to manage the localization options correctly. Currently, our implementation supports a Hungarian and a US English virtual key set for this purpose.

Date and *Time* types are representing the usual values based on the actual region settings. The difficulty of parsing their picture attribute lies in the fact that they can hold partial values (e.g. just the years and months or the hours) too. The picture of these values are run through extensive processing as they are being bind to regular values that the system can manage commonly by defining dynamic value converters for them. These converters are designed to handle the customized pictures and they are used also during input generation because they can convert values back and forth.

Logical type variables are using the well know true–false co-domain, but their picture can change their appearance between numeric and character forms. Furthermore, associative values (e.g. Yes–No, Active–Inactive) can be mapped to them by using special domains.

Unlike the numeric types of the 3GLs the *Numeric* of Magic xpa is a more universal variable type. It can hold integer and real values as well. The actual type can only be determined by analyzing the picture attribute.

5 Conclusion

In this paper, a general white-box based testing method was presented that supports automated testing. The method can be used for any programming language but it is especially aligned with 4th generation languages because of their specific attributes, like direct links between input fields and stored data, descriptive constraints of data fields, etc. The input of the method are the source code and some business models. The source code is analyzed and different execution paths are extracted using the built CFG and the business model. Test inputs are assigned to these paths (and this is the point where the method relies on the properties of the 4GLs). Finally, a script generator transforms this data into executable test cases which are then run in a test execution framework.

The implementation of this method for testing Magic xpa applications was also presented. A source code analyzer and a test execution framework had been available for this platform from previous works. These were connected to the test case generator module, which included execution path, test input, test case, and test script generator parts.

The toolchain was evaluated on some pilot projects with promising results (e.g. it took only 4 seconds to generate a test suite which covered 100% of the tasks and the logic units related to the example scenario presented in Section 4), however, there are no extensive experiences about its performance. In the future, we are going to evaluate the performance regarding the needed computational effort and the achieved coverage. In addition, it is planned to deploy the toolchain in real environment.

References

[1] Abramowicz, Witold and Fensel, Dieter. *Business information systems [electronic resource]: 11th international conference, BIS 2008, Innsbruck, Austria, May 5-7, 2008, proceedings.* Lecture notes in business information processing. Springer, 2008.

[2] Allen, Frances E. and Cocke, John. A program data flow analysis procedure. *Commun. ACM*, 19(3):137–, March 1976.

[3] Bakota, Tibor, Beszédes, Árpád, Gergely, Tamás, Gyalai, Milán, Gyimóthy, Tibor, and Füleki, Dániel. Semi-automatic test case generation from busi-

ness process models. In *Proceedings of the 11th Symposium on Programming Languages and Software Tools (SPLST'09) and 7th Nordic Workshop on Model Driven Software Engineering (NW-MODE'09)*, pages 5–18, Tampere, Finland, August 26-28 2009.

[4] Coleman, Joey, Jones, Cliff, Oliver, Ian, Romanovsky, Alexander, and Troubitsyna, Elena. *RODIN (Rigorous Open Development Environment for Complex Systems)*. University of Newcastle upon Tyne, Computing Science, 2005.

[5] Dévai, Richárd, Jász, Judit, Nagy, Csaba, and Ferenc, Rudolf. Designing and implementing control flow graph for Magic 4th generation language. In Kiss, Ákos, editor, *Proceedings of the 13th Symposium on Programming Languages and Software Tools (SPLST'13)*, pages 200–214, Szeged, Hungary, 2013. University of Szeged.

[6] Ferenc, Rudolf, Beszédes, Árpád, and Gyimóthy, Tibor. Fact extraction and code auditing with Columbus and SourceAudit. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, ICSM '04, pages 513–, Washington, DC, USA, 2004. IEEE Computer Society.

[7] Ferenc, Rudolf, Beszédes, Árpád, Tarkiainen, Mikko, and Gyimóthy, Tibor. Columbus – Reverse engineering tool and schema for C++. In *Proceedings of the 18th International Conference on Software Maintenance (ICSM 2002*, pages 172–181. IEEE Computer Society, 2002.

[8] Fritsi, Dániel, Nagy, Csaba, Ferenc, Rudolf, and Gyimóthy, Tibor. A layout independent GUI test automation tool for applications developed in Magic/uniPaaS. In *Proceedings of the 12th Symposium on Programming Languages and Software Tools (SPLST'11)*, pages 248–259, 2011.

[9] He, Nannan, Rümmer, Philipp, and Kroening, Daniel. Test-case generation for embedded Simulink via formal concept analysis. In *Design Automation Conference (DAC)*, pages 224–229. ACM, 201.

[10] Jha, Somesh, Datta, Anupam, Li, Ninghui, Melski, David, and Reps, Thomas. *Analysis Techniques for Information Security*. Morgan and Claypool Publishers, 2010.

[11] Kaner, Cem. Architectures of test automation. *STAR West*, 2000.

[12] Kim, Young G., Hong, Hyoung S., Bae, Doo-Hwan H., and Cha, Sung-Deok D. Test cases generation from uml state diagrams. *Software, IEE Proceedings* -, 146(4):187–192, Aug 1999.

[13] Kit, Edward. Integrated, effective test design and automation. *Software Development*, 7(2):27–41, 1999.

[14] Leung, Nelson K. Y., Nkhoma, Mathews, and John, Blooma. *Proceedings of the 4th International Conference on IS Management and Evaluation: ICIME 2013*. Academic Conferences and Publishing Limited, 2013.

[15] Magic Software Enterprises Ltd. Magic xpa Application Platform RIA Demo. `http://demo.magicsoftware.com`, 2015. [Accessed 10-Mar-2015].

[16] Mingsong, Chen, Xiaokang, Qiu, and Xuandong, Li. Automatic test case generation for uml activity diagrams. In *Proceedings of the 2006 International Workshop on Automation of Software Test*, AST '06, pages 2–8, New York, NY, USA, 2006. ACM.

[17] Nagy, Csaba, Vidács, László, Ferenc, Rudolf, Gyimóthy, Tibor, Kocsis, Ferenc, and Kovács, István. Solutions for reverse engineering 4GL applications, recovering the design of a logistical wholesale system. In *Proceedings of CSMR 2011 (15th European Conference on Software Maintenance and Reengineering)*, pages 343–346. IEEE Computer Society, March 2011.

[18] Richardson, Debra J. and Clarke, Lori A. A partition analysis method to increase program reliability. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 244–253, Piscataway, NJ, USA, 1981. IEEE Press.

[19] Tip, Frank. A survey of program slicing techniques. Technical report, CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands, The Netherlands, 1994.

[20] Vanhatalo, Jussi, Völzer, Hagen, and Leymann, Frank. Faster and more focused control-flow analysis for business process models through SESE decomposition. In *Proceedings of the 5th International Conference on Service-Oriented Computing*, ICSOC '07, pages 43–55, Berlin, Heidelberg, 2007. Springer-Verlag.

[21] Visser, Willem, Păsăreanu, Corina S., and Khurshid, Sarfraz. Test input generation with Java PathFinder. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04, pages 97–107, New York, NY, USA, 2004. ACM.

[22] Weimer, Westley and Necula, George C. Finding and preventing run-time error handling mistakes. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 419–431, New York, NY, USA, 2004. ACM.

[23] Wenzel, Ingomar, Rieder, Bernhard, Kirner, Raimund, and Puschner, Peter. Automatic timing model generation by cfg partitioning and model checking. In *Design, Automation and Test in Europe, 2005. Proceedings*, pages 606–611 Vol. 1, March 2005.

[24] Yan, Jun, Li, Zhongjie, Yuan, Yuan, Sun, Wei, and Zhang, Jian. Bpel4ws unit testing: Test case generation using a concurrent path analysis approach. In *Software Reliability Engineering, 2006. ISSRE '06. 17th International Symposium on*, pages 75–84, Nov 2006.

[25] Yuan, Qiulu, Wu, Ji, Liu, Chao, and Zhang, Li. A model driven approach toward business process test case generation. In *Web Site Evolution, 2008. WSE 2008. 10th International Symposium on*, pages 41–44, Oct 2008.

[26] Yuan, Xun and Memon, Atif M. Using gui run-time state as feedback to generate test cases. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 396–405, May 2007.