Acta Cybernetica **21** (2014) 419–437.

Designing and Implementing Control Flow Graph for Magic 4th Generation Language

Richárd Dévai[*], Judit Jász[†], Csaba Nagy[†], and Rudolf Ferenc[†]

Abstract

A good compiler which implements many optimizations during its compilation phases must be able to perform several static analysis techniques such as control flow or data flow analysis. Besides compilers, these techniques are common for static analyzers as well to retrieve information from source code, for example for code auditing, quality assurance or testing purposes. Implementing control flow analysis requires handling many special structures of the target language. In our paper we present our experiences in implementing control flow graph (CFG) construction for a special 4th generation language called Magic. While we were designing and implementing the CFG for this language, we identified differences compared to 3rd generation languages mostly because of the unique programming technique of Magic (e.g. data access, parallel task execution, events). Our work was motivated by our industrial partner who needed precise static analysis tools (e.g. for quality assurance or testing purposes) for this language. We believe that our experiences for Magic, as a representative of 4GLs, might be generalized for other languages too.

1 Introduction

Control flow analysis is a common technique to determine the control flow of a program via static analysis. The outcome of this analysis is the Control Flow Graph (CFG), which describes the control relations between certain source code elements of the application. A CFG is a directed graph: its nodes are usually basic blocks representing the statements of the code that are executed after each other without any jumps. These basic blocks are connected with directed edges representing the jumps in the control flow. A CFG is a useful tool for code optimization techniques (e.g. unreachable code elimination, loop optimization or dead code elimination). The first publications of using control flow analysis are from the 70s [1] and 80s [4, 10, 21], but since then most of the compilers have implemented this kind of analysis to construct a CFG and implement optimization phases by using it.

[*]FrontEndART Software Ltd, E-mail: `devai@frontendart.com`

[†]Department of Software Engineering, University of Szeged, Hungary, E-mail: `{jasy,ncsaba,ferenc}@inf.u-szeged.hu`

DOI: 10.14232/actacyb.21.3.2014.9

Although the basic structure of a CFG is quite common, the methods constructing it for applications are rather language dependent. Identifying control dependencies in special structures of the target language may result in special algorithms. Moreover, some program elements or applications may require minor modifications in the structure of the CFG (e.g. nodes like entry nodes).

In our paper, we present our experiences in implementing Control Flow Graph construction for a special language called Magic. This language is a so-called 4th generation language [22] because the programmer does not write source code in the traditional way, but he or she implements the application 'at a higher level' with the help of an application development environment (Magic xpa[1]). This unique programming technique has many differences compared to 3GLs which are the most common languages today (Java, C, C++, C#, etc.). Due to the programming style of Magic, we had to revise traditional concepts like program components, expressions and variables during the design of a CFG for Magic applications.

The main contributions of this paper are (1) development of a CFG construction technique for applications developed in Magic xpa, (2) identification of CFG implementation differences in a 4GL context as opposed to 3GLs.

Our work was motivated by our industrial partner who needed a tool set to perform precise static analysis for code auditing and to support their testing processes. In the case of code auditing, the CFG is an important input for static code checker algorithms, while in the case of testing, the CFG is an input for algorithms which generate test scripts for automatic UI testing. Our experiences in Magic, as a representative of 4GLs could provide a good basis to implement CFG construction for other 4GLs too.

2 Related work

Control flow is a widely used information container for example in the compiler programs of 3GLs. The method of a CFG construction is well defined in [17]. We need to discover and identify the statements, and define basic blocks by selecting leader statements. Key steps are to define the structures to handle control passing, and the elements for those items of logic which implicitly influence the behavior of the control flow.

Control flow analysis has many applications, such as program transformations or source code optimizations in compilers[2] [11], rule checkers of analyzer tools [6, 7, 20], security checkers [5], test input generator tools[3] [25], or program slicing [23]. Program dependence analysis approaches are also based upon control dependencies computed by control flow analysis [9].

The implementation of control flow analysis might differ for different languages. There are many papers published about dealing with higher-order languages (e.g. Scheme), for instance the work of Ashley et al. [2] and the PhD thesis of Ayers [3]

[1]Magic Software Enterprises Website: `http://www.magicsoftware.com`

[2]GCC Internals Online Documentation: `http://gcc.gnu.org/onlinedocs/gccint/`

[3]Prasoft Products: `http://www.parasoft.com/jsp/products.jsp`

both summing up further works too [10, 21]. An extensive investigation had been done for functional languages too, which was recently summed up by Midtgaard in a survey [16].

However, CFG solutions for 4GLs are really limited. E.g. ABAP, the programming language of SAP is a popular 4GL and there are only few published flow analysis techniques which mostly deal with workflow analysis [13, 24].

In our previous work [19] we implemented a reverse engineering tool set for Magic and we found a real need to adapt some of these techniques to the language. Besides our work, Magic Optimizer[4], as a code auditing tool also shows this necessity. This tool checks for violations of coding rules (i.e. 'best practices'), and it is able to perform optimization checks and further analyses to give an extended overview of every part of a Magic application.

3 Specialties of a Magic Application

In the early 80's Magic Software Enterprises introduced a 4th generation language, called Magic. The main concept was to write an application in a higher level meta language (using already existing solutions for instance for data handling and user management) and let an application generator engine create the final application. A Magic application was runnable on popular operating systems such as DOS and Unix, so applications were easily portable. Magic evolved and new versions were released, uniPaaS and lately Magic xpa. Latest releases support modern technologies such as RIA, SOA and mobile development.

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, he or she actually programs the Magic Runtime Application Environment (MRE) using its meta model. This meta model is what really makes Magic a 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) binded 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.

The main building blocks of a Magic application are defined in repositories. For example in the *Data Sources* repository one can define Data Objects. These are essentially the descriptions of the tables in a database. Using these objects Magic is able to handle several database management systems.

The logic of an application is implemented in the programs stored in the *Programs Repository*. Programs are the core elements of an application. These are executable entities with several sub tasks. Programs or their tasks interact with

[4]Magic xpa tools: `http://www.magic-optimizer.com/`

the user through forms to show the results of the implemented logic. Forms are also parts of tasks or programs.

Developers can edit a program with the help of the different views. The main views are the followings:

Data View. Declares which *Data Objects* are bound to the programs. The binding generally means some variable declaration, where these declarations can be real or virtual. A real declaration connects a variable to a data table column, while a virtual declaration stores some precomputed data.

Logic View. Defines *Logic Units* of a program. A task has a predefined evaluation order determined bz so-called execution levels, and *Logic Units* are the parts of a task to handle the different execution levels. E.g. *Task Prefix* is the first *Logic Unit* which is executed to initialize a task. Actually, *Logic Units* are the units where the developer can write 'code' like in a 3GL. We can define statements here to perform calculations, manipulate data, call sub tasks, etc. Statements appear as *Logic Lines* in the *Logic Unit*.

Form View. Defines the properties of a window (e.g. title, size and position). Elements of a window can be typical UI elements such as controls or menus. A window is represented by a *Form Entry* in which we can use many built-in controls or our custom controls too.

As it can be seen, a Magic 4GL application differs from the programs developed in lower level languages. Developers can concentrate on implementing the business logic and the rest is done by Magic xpa.

4 Control flow graph construction

In this section, we discuss the main definitions and steps of the control flow construction for 3rd generation languages and introduce the specialties of the control flow graph construction for Magic as a representative of 4GLs.

4.1 Definitions and general steps

A **control flow graph** is a graph representation of the computation and the control flow in a program, as it can be seen in an example in Figure 1. The nodes of a CFG are basic blocks represented by rectangles. Each basic block represents a set of statements that are executed after each other sequentially. Branching can only exist at the end of blocks, after the execution of their last encapsulated statement.

The first step in the control flow creation is to determine the starting points of basic blocks [17]. These statements are called as *leaders*, and a leader can be:

- the first statement of a program,
- any statement that is the target of a conditional or unconditional branch statement,

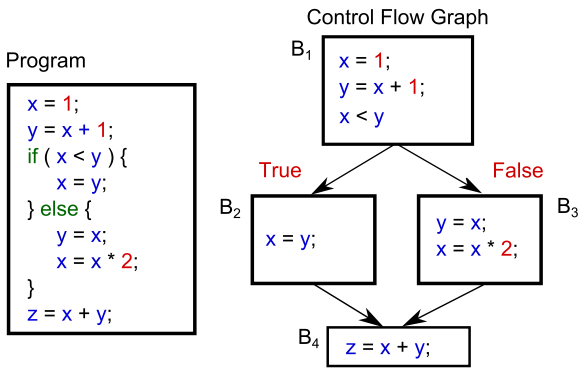

Figure 1: CFG of a simple conditional structure.

- any statement that immediately follows a conditional or unconditional branch statement,
- any statement that immediately follows a method invocation statement[5].

If we know the sequence of statements in a program and the leaders of basic blocks, we can determine the blocks by enumerating their statements from one leader to another, but not including the next leader or the end of the program. Compilers and source code analyzers first construct an intermediate representation of the source code, called abstract syntax tree (AST) that implicitly describes the sequence of statements. With the traversal of the AST we can determine the sequence of statements, and if we want to build the control flow with finer granularity, we can examine the evaluation order of the expressions. We will discuss finer representations under the examination of Magic expressions and call types in Section 5.

In general, the control flow information of methods, procedures or the subroutines of a program are represented individually. Due to technical reasons, each of these has two special kinds of basic blocks. The *Entry* block represents the entering of a procedure, while the *Exit* block represents the returning from a called procedure. The potential control flows among procedures are represented as call edges. A connected control flow graph of a procedure with the call information gives the so-called interprocedural control flow graph (ICFG) of a program. Figure 2 shows an example of the ICFG, where call edges are represented as arrow-headed dashed lines between the call site and the *Entry* block of the called procedure, and the *Exit* block and the return statement in the caller ICFG component. In some cases, detecting procedure boundaries is not an easy task, and a call target or a branch instruction cannot be determined unambiguously. The earlier situation commonly appears in binary codes [12], while the later is typical in the presence of function pointers or virtual function calls in higher level languages. The problems appeared in 4GLs are discussed in the rest of this section.

[5]Method invocations should not be basic block boundaries in all cases only if we need compute some summarized information at the call sites in our connected application.

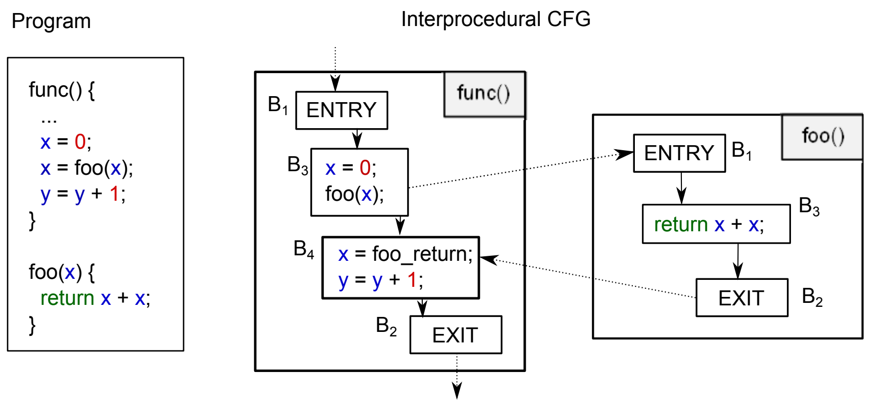

Figure 2: Example ICFG.

4.2 Challenges in Magic

Like compiler programs or other software analyzer tools do it, our first step is also to create an intermediate representation of a Magic application. We call this representation the Magic Abstract Syntax Graph (ASG)and its structure is defined by the Magic Schema [18]. The ASG allows us to traverse and process every required element of a Magic application in a well-defined hierarchical graph format through an API to determine the execution order of Magic statements. Nodes of the ASG have all the necessary attributes that can affect the control flow. E.g. ASG contains the propagation information of `Event Handlers`, which can terminate the execution of other event handlers, or the wait attribute of `Raise Event`, which determines the execution point of the given event.

Developing an application in Magic requires a unique way of thinking since the programming language is unique itself. However this programming language preserves some of the main characteristics of procedural languages. Mostly, the main logic of an application can be programmed in a procedural way via control statements in programs and their subtasks. Programs can call each other and they can call their subtasks. Also, tasks can use variables for their computations, and they can have branches within their statements. These structures of the language make it possible to adapt the CFG construction of 3GLs to Magic 4GL. For example, for every potential target of the call sites of Magic (task, event handler, developer function) we make an intraprocedural control flow graph and we connect these graphs by call edges to get the ICFG. However, there are some structures in the language which make harder to construct the CFG of an application. Here, we discuss the challenges which we face in later sections.

Tasks architecture has a special event-based execution system. There are different task types for different operations. For example, online tasks interact with the user and batch tasks run in background without any user interaction. Each task type has its own levels (e.g. task, record) and the developer can operate these with the so-called *Logic Units*. A user action or a state change in a program can trigger predefined events that are also handled by the *Logic Units* of tasks. So, the statements (*Logic Lines*) of these *Logic Units* get executed if a certain event triggers them. The most challenging step to construct the CFG of a Magic program is to discover every circumstance that can change the flow of the control among *Logic Units* and *Logic Lines*. We have to understand and represent the effect of property changes which can influence the behavior of execution, and represent it in a well describing form.

A **Raise Event Logic Line** raises an event which is later handled by an **Event Logic Unit**. When an event is raised, the MRE immediately looks for the last available handler in the given task, and gives the control to the handler. This is the simplest case, the synchronous case. However, we could raise events asynchronously; or set the scope of handlers as they could be handled by parent tasks too, or only by the task which raised them; or every matching handler could terminate the chain of handlers if propagate property is set to 'no'. Describing the proper event handler chains within the CFG requires a complex traversal of logic units in the task hierarchy with respect to the influencing attributes. Our model is limited to those events which are raised by a code element or a form item.

Data access is supported with a rich toolset in Magic to access databases. Magic provides support to many database management systems (RDBMSs) by handling connection, transactions and generation of queries. In general, we can choose from two options to perform our transactions. In the Physical mode other DB users see our changes in RDBMS log and we use the locking system of the DB server. In the Deferred mode Magic xpa is responsible for storing our changes and committing them when we have assembled our transaction within a running task. Besides the transactional modes, we have to select the method of update process for the records we use in the transactions. Different strategies give us opportunity to handle concurrency and integrity on record updates. During the creation of the CFG we have to handle the different event handlers based on the selected transaction mode and update strategy.

Parallel task execution makes it possible to execute more programs in parallel. Parallel programs run in an isolated context where every loaded component of the main application are reloaded within the new context. In such context, a parallel program has its own copy of memory tables and its own database connections with some limitations (e.g. it cannot store data in the main program or communicate directly with other running programs). Tasks can raise asynchronous events in the context of another program to communicate, or they can use shared variables through proper functions in expressions. Parallel processes can run in Single or Multiple instance modes. In the Single mode the context is the same for each instance of the task, while the Multiple mode uses different contexts for each task. At the CFG construction we have to simulate all hidden data copying and the parallel execution of statements.

Forms have many uses during a program execution. In each case, we have to build the CFG according to the current use of forms. In a form a user can manipulate variable data, which appear in the running program as an assignment instruction, or the user can affect the running program behavior too.

5 Implementation details

The process of CFG building has several phases. First, with the traversal of the ASG we determine the sequence of statements and the evaluation order of expressions. During evaluation we collect information about calls, then we determine basic block leaders and finally, we build up the basic blocks for later processes. In our representation, each call site is a block boundary.

To determine the execution order of the contained statements and form elements of an analyzed code, we traverse its ASG from the root node step by step in the tree hierarchy and we refine the control flow information among the sub components. In each step, we define the execution order of the composed nodes of an investigated ASG node and we augment the execution sequence with additional expressions or statements, if it is needed. We do this since many semantic elements of a programming language do not appear explicitly in the source code and so in its ASG representation. Due to the hierarchical traversal, the control flow information of descendant nodes is refined after the traversal of their ancestors.

Rectangles in the figures of this section represent nodes, or groups of ASG nodes. Parallelograms denote branches where the possible flow of control depends on an attribute of `Logic Units`, `Logic Lines`, controls, variables, etc. Black arrows denote the control edges of the CFG, while dashed lines represent the call edges among the intraprocedural CFG components. Since in our representation call instructions are basic block boundaries, we represent each call with two virtual nodes called `Call Site` and `Return Site`. In some cases, we introduce solutions of alternative program versions with the help of one figure. To distinguish the variations of these versions, we use black branching points on the paths where the behaviors of the different versions are differ.

In the following sections, we discuss the cases where we could create general algorithms to process group of nodes with the same base type. Finally we introduce some special solutions where the general algorithms are not able to describe precisely the real evaluation order of the descendants of the analyzed ASG node.

5.1 General algorithms

Tasks in the ASG represent either programs or their sub tasks. The final representation of a `Task` is influenced by the implementations of its `Logic Units`, and its variables, but first we have to concentrate only on the skeleton of the tasks, since the finer control flows of `Logic Units` are determined in later steps of the traversal.

When we reach a `Task` node in the traversal, first we create an intraprocedural CFG context for the `Task` node. Our second step is to collect the sequence of logic units that take part in the execution process of the task. These nodes are the child nodes of the `Task` node in the ASG. `Task`, `Group`, and `Record` are subtypes of the `Logic Unit`, but of course, the existence of these elements are only optional in each `Task`. `Prefix` and `Suffix` are sub categories of previous `Logic Unit` subtypes controlled by an attribute. The subtype and the selected attribute value determine

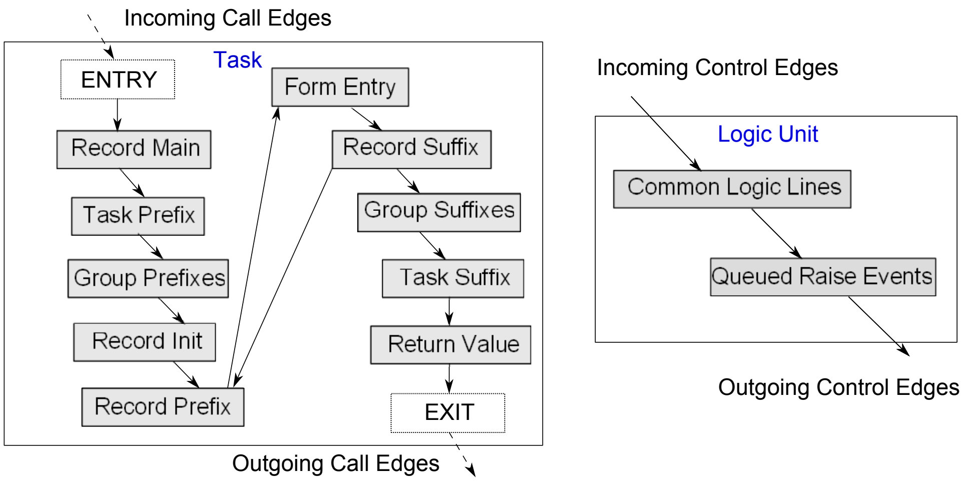

Figure 3: Evaluated control flow of a `Batch Task` and a `Logic Unit`.

the exact execution point and order of these `Logic Units`. So, we nominate the diversity of `Logic Units` with the addition of subtypes to their names as it can be seen in Figure 3.

We do not connect every `Logic Unit` subtype in this step, only the `Task`, `Group` and `Record`. For the `Event` and `Function` subtypes of the `Logic Unit` we associate a distinct intraprocedural CFG and handle them separately since these kinds of `Logic Units` can be triggered several times from distinct points.

Generated source codes and behaviors of MRE are different from the structure that we can see in Magic xpa while developing a `Task`, because variable declarations and initializations are also parts of the execution of logic, but defined in a separated view as we showed it in Section 3. The creations of variables and default value assignments are at the start point of a task execution. These commands are gathered by the `Record Main` node.

While `Task` and `Group` logic units have only two subcategories, `Prefix`, `Suffix` and `Record` logic units logically have three distinct in a loop of control. Each execution round of `Record` logic units could have an initialization part that does not appear in the code explicitly. Since it has an important effect on the control flow, we insert a virtual `Record Init` node into the flow of execution. If we do not find any initialization during the investigation of variables in the traversal of the record unit, or the task is not in 'write' mode and the initializations use real variables only, we can delete this `Logic Unit` from the CFG at the end of the traversal of the `Task`. During the traversal, we collect available data about `Forms`, their associated `Controls` and the logic related to them and map this information to a suitable structure of statements. This data is represented by the `Form Entry` node in the figure between the `Record` logic units as they handle the data initialization, pre- and post-processing of a record of a table. In the last step, we investigate the return expression node of the `Task`, and if it exits we connect it as the last item

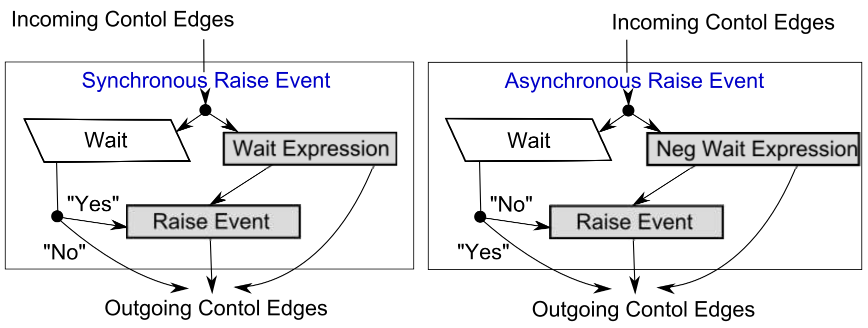

Figure 4: Control flow of `Raise Events`.

before the `Exit` block of the `Task`.

On the left side of Figure 3, we can see the execution order of a `Batch Task` or a `Browse Task`. This task contains variables, implements all possible `Logic Unit` subtypes, and defines a return expression.

After visiting all the nodes of a `Task`, we are able to build up its basic blocks and determine the control and call edges among them. With this information we can derive the exact execution order of the statements and expressions.

Each **Logic Unit** consists of `Logic Lines`. Generally, `Logic Lines` have two distinct kinds. First, the execution of the logic line does not depend on its properties or on the execution of other logic lines; we handle them as they can run sequentially in the order of appearance until further checks. We refer to these as `Common Logic Lines`. The second kind is the so-called `Raise Event` with an attribute called `wait` that we have to observe. With the `Raise Event` nodes we determine the asynchronously executed `Queued Raise Events` according to Figure 3, if the value of the wait attribute is 'no'. The wait attribute of the `Raise Event` can have a 'yes' or 'no' boolean constant value or the result of a boolean expression. Since the execution of these lines depend on the value of the wait attribute, we have two distinct cases. If this value is logically true the raise events are synchronous otherwise they are asynchronous. An illustration can be seen in Figure 4.

The execution of a `Logic Line` depends on a condition. If this condition evaluates to true, the flow of control goes into the statement, which describes the exact behavior of the logic line. Although this part of the evaluation of the logic lines is general, the behavior of the distinct subtypes of `Logic Lines` can be very different as we can see in the next section.

5.2 Specific algorithms

As it was mentioned in the last subsection, the **Function** and **Event Logic Unit** nodes are different from other logic units, but similar to each other. Since the execution of these units depend on their context, and their execution can be triggered from different points of the program, it is better to handle them in a similar way as we handled the `Task` nodes. Hence, for these nodes we created intraprocedural

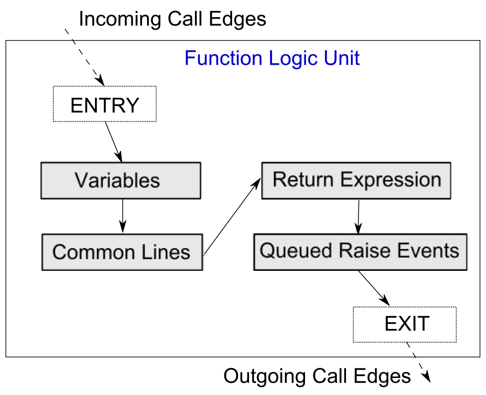

Figure 5: CFG of Function Logic Unit.

CFG representations which are callable from distinct program points. Next, we collect `Logic Lines` which are variable declarations from their contained `Logic Lines`, because they are not necessarily in order before all other `Logic Lines`, but executed collectively at the beginning of the execution of the `Logic Unit`. Next we have to perform an algorithm like we performed for `Logic Units`. The difference between `Function` and `Event Logic Units` is that the former could define a `Return Expression` declared by an attribute of the `Logic Unit` which is executed before the `Queued Raise Events` as it is shown in Figure 5.

Logic Lines are evaluated through the traversal by specific evaluators. These elements of logic are much more unique from the point of view of control flow processing than the `Tasks` and `Logic Units`. We introduce some of these to show the variety and the complexity of their processing.

A `Block` node is implemented by a `Logic Line` pair. A **While Block** with its related `End Block` declare the start and the end of the `Block`. These two encapsulate the body of the `Block`. When we find a `While Block` in the ASG, we have to search its terminating `End Block` node, because they are not connected directly in the ASG. The condition of a `While Block` can be a 'yes' or 'no' constant or an `Expression`. Nesting `Block` nodes make it harder to carry out this task. The left hand side of Figure 6 shows the evaluation of a while structure. The structure of an **If Block** is similar to the structure of a `While Block`. First, we have to search the corresponding `End Block` and `Else Blocks` for each `If Block` node. The multiple selection is implemented by the optional condition argument of an `Else Block` node.

The right side of Figure 6 shows a **Call** logic line which implements a call based on a Magic generated identifier of a program, a sub task or a public name, etc. A `Call` logic line node has an optional argument list and could receive a return value. The passed-by-reference arguments are updated after the control is given back to the `Return Site`. To implement this behavior in the CFG, we have to create update nodes for them. Before the actual call, we insert a `Call Site` node into the CFG, while after the execution of the `Exit Block` of the called CFG we

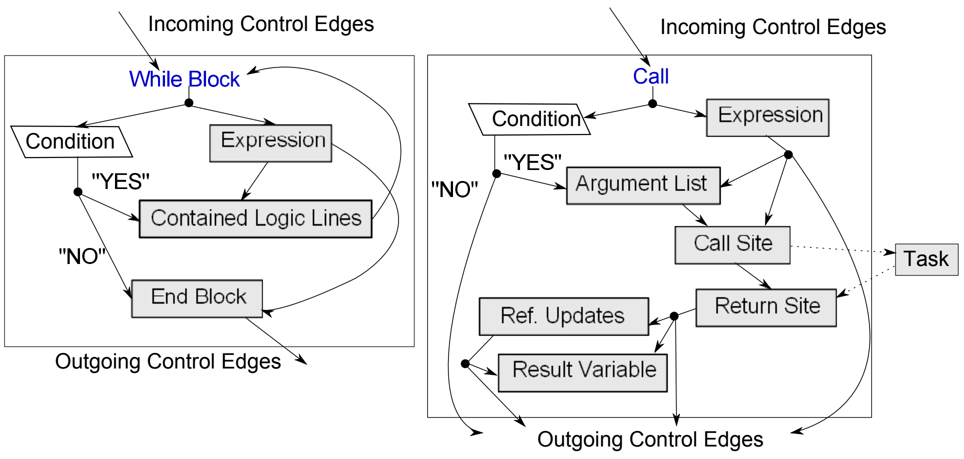

Figure 6: CFG of a While block and a general Call logic line.

nominate the return with a `Return Site` node.

Select Logic Lines defined in the Data View are separated from the code. Semantically these `Select Logic Lines` are executed in the `Record Main` and `Record Init` Logic Units during of a given task. Hence, handling expressions of `Select Logic Lines` is similar to the way we handle normal `Logic Line` types.

All **Expressions** of Magic are arranged into subtypes by categories in our ASG representation. An `Expression` can be a literal, a unary or binary operation or a Function Call that refers to a built-in function or `Function Logic Units`. `Literals` can make a reference to an identifier, a resource or a component, or they can contain a constant value.

The control flow of a `Function Call` can be built-up as a simpler `Call Logic Line`, the only difference is that its arguments cannot be passed by reference.

5.3 Associated control structures of Form logic

One primary motivation of our work was to support the UI testing of Magic applications, so it is essential to represent control dependencies arising via UI elements such as controls of Forms as they provide the main interface for user interactions. Magic programs basically follow a strongly event driven model and most of the events are generated by the UI elements of Forms.

In Magic, the structure of the UI or the layout of Forms and controls is readily available in the ASG, so thanks to the language, the connection between `Controls` and their related logic is also available (e.g. relation between an edit box and its related variable; or relation between a menu and the task to be executed). Based on this information, we can extend the CFG with UI elements and their control relations, so we can get a better view of the control flow than in an event-based context.

There have been several attempts to develop techniques for the generation of automated GUI tests with less or more success by abstract state machines, but most of the techniques are *ad hoc*, and mostly manual; in addition, there is also a great potential in modeling event interactions with directed graphs e.g. by modeling the event flows of applications as noted in [15]. Similarly, we represent events in the CFG as there is a great number of events built into Magic.

There are several control types which we group into the following two logical classes based on the control structures that we handle them with:

$Group_1$: **Push Buttons**, **Sub Forms**, **Menus**, Sub Menus and Context menus

$Group_2$: Input controls such as **Edit boxes** and Lists, Radio Buttons and Check Boxes

*Group*1 contains items which are responsible for process control and embedding, while items of $Group_2$ handle input data and could be wrapped into a validation context.

`Sub Forms` are useful to integrate a task form into the form of another task while maintaining the subform's task data handling and record cycle activities as independent of the parent task. This is a good solution for reusing data, logic and also their GUI parts.

Input controls (e.g. edit boxes) are useful for setting the value of a `Real` or `Virtual Variable`. These controls take input data to change the values of variables and there are several kinds of validators and programming logic (e.g. logic units) to handle their usage.

`Menus` can be used to navigate between different programs in an application through calls and events, so they provide access to a large variety of functionalities.

`Push Buttons` are good for triggering events to place several crucial behavior just in front of the user to ease navigation, and give an opportunity to stress the operations which should be emphasized. E.g. it is possible, but rather unusual to use a context menu in a calculator application to add numbers.

In a form the simple behaviour would be that all the controls are related to each other, since the sequence of control invocations is undetermined and we need to represent all possible sequences (e.g. imagine the user pushing the buttons randomly). Representing this would radically increase the number of edges in the CFG, so for simplicity, we introduce the so-called *entry* nodes which virtually join all the controls in a Form. A Form can have multiple exit points depending on which `Suffix` follows the Form in the program.

In Figure 7. we see the CFG part of a form structure with an example *entry* node, which we introduced before. The controls like `Edit Box` could be surrounded with different types of validation logic. In the case of $Group_2$ controls, there is one-to-one correspondence established by the language for variables, so to each `Control` a `Variable` with a `Variable` Logic Unit that is responsible for handling the changes of its value will be assigned. These Logic Units either receive the original value and the new one, or only one of them and they do an initialization, validation or checking step.

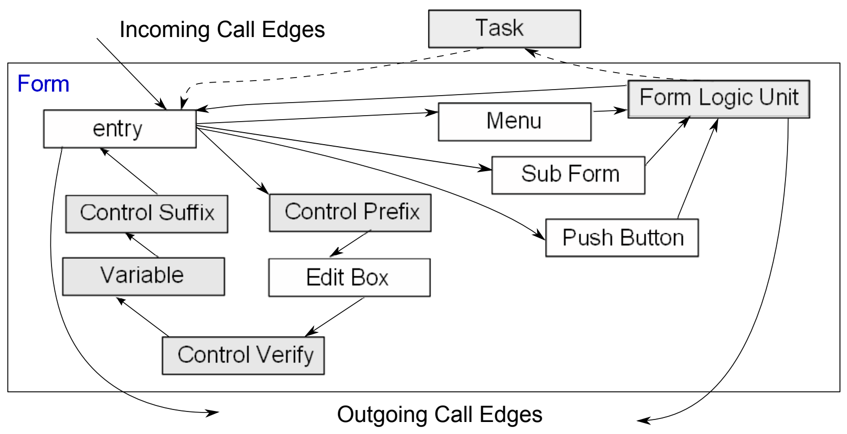

Figure 7: CFG context of a Form Entry and its encapsulated controls.

`Menus`, `Push Buttons` and `Sub Forms` are different as they are responsible for navigation and encapsulation. Menus can mainly possess two different types of behavior as they can trigger events, or they can call other Programs. To represent this behaviour, we created a virtual Logic Unit called `Form Logic Unit`. The purpose of this virtual logic unit is to group together virtual raise event statements and call the statements which are stored in the ASG under the corresponding task. These virtual raise event statement simulate how the control of a form can actually raise an event in the control flow (see the `Form Logic Unit` node in Figure 7). Such a logic unit can have incoming control edges from $Group_2$ controls.

`Push Buttons` are similarly handled as Menus except that they cannot call other tasks only raise an event.

A `Sub Forms` is an embedded form in another main Form. The content of the Sub Form is provided by a sub task of the task of its main form. In order to represent this structure in the CFG, Sub Forms are connected through a task `Call` to their main Form.

Program control can leave the form context through events related to task termination or user actions. This is symbolized by outgoing control edges in Figure 7.

6 Evaluation

We implemented our technique in C++ and verified it through result validations and performance tests. For this verification we created a testbed with 105 Magic applications which were specifically designed to implement special control structures in Magic. We created a simple batch script to construct the ASG and then the CFG for each application in this testbed. Then, we manually compared the constructed CFGs to the program code. To perform this comparison, we exported the constructed CFG to graphML format which can be easily visualized with yED[6].

[6]yED Graph Editor: `http://www.yworks.com/en/products_yed_about.html`

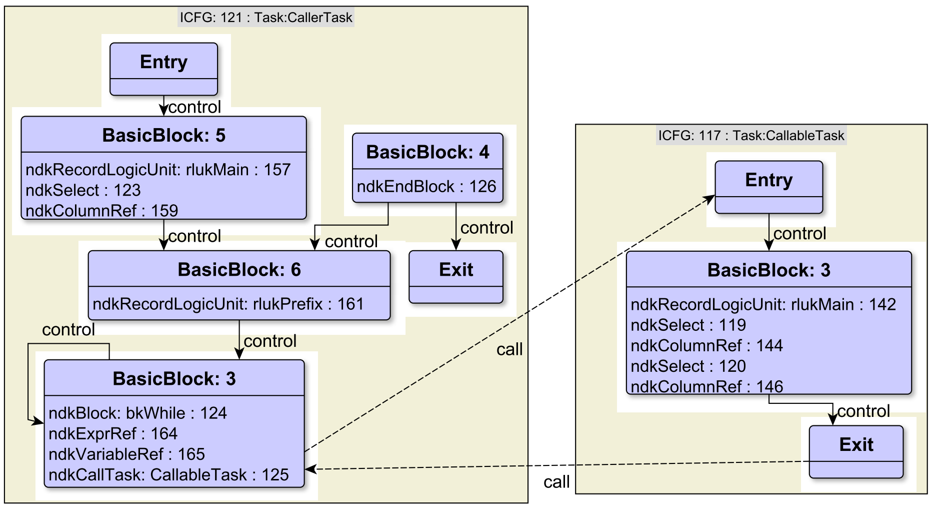

Figure 8: Visualized ICFG by generated graphML dump.

An exported picture of a sample graphML can be seen in Figure 8. The original code contains an infinite `While Block`. This information is shown in the figure too, where basic block with id 4 is unreachable. This information could be easily retrieved by API calls during the traversal of the CFG. Of course, in this case this possibly malformed control structure is recognized by Magic xpa too, it warns the programmer about the existence of the infinite loop. The example of the figure contains a call from the body of the `While Block`. This call also appeared in our ICFG. We compared all the resulting dumps with the original source code manually, and we found that each ICFG gave a good description of the possible execution paths of the original code.

After manually evaluating all the constructed CFGs of the testbed, we evaluated our implementation on a larger application too (the demo application released with Magic xpa). Not only did we construct the CFG and check its consistency, but with profiling we gathered run-time statistics of our algorithms too.

To verify the usability of our algorithms, we ran our implementation on an Intel XENON E5450 @ 3GHz 32 GB Windows Server 2008. As performance results on a medium sized sample project with nearly 200.000 nodes and about 500.000 attributes we got a 0,598 seconds runtime of the ICFG computation. The ICFG computation was carried out in an affordable time, so it was adaptable in any approaches based on this information.

7 Limitations of the approach

Besides the shown advantages of our technique, there are a few limitations too. Here we describe two main limitations.

Our event handling does not handle all the possible specialties of a Magic application. Currently, the implementation is able to follow the events that are raised and handled inside the code with a raise event statement or a certain logic unit. The internal events of Magic xpa (such as hotkeys) are not yet supported unless raised by a raise event statement.

Our recent CFG model does not support the representation of parallel task executions given by section 4.2. To improve our model, we should investigate previous work about the limitations and possible application forms of CFG for parallelism support e.g. [14].

8 Summary and Future Work

In our paper, we presented an application of CFG concepts for a specific 4th generation language, Magic 4GL. We used a static analysis approach to gain information from the generated Magic source code and to build a CFG with fine granularity. We created a reusable library for further use of our model which makes it possible to perform further analyses and process the CFG and ICFG structures which we created. We created a textual and an XML based graphML dump to make it easy to get an overview of the processed information.

Our evaluation showed that the approach implemented is applicable for middle-sized Magic applications. The method presented had an affordable space requirement and it constructed the CFG fast enough to analyze large projects too.

Besides, we showed that implementing control flow analysis for a higher-level language, such as Magic, was possible via adapting 3GL techniques, but the unique structures of the language may result in special methods and structures in the CFG as well. For example, the use of Events enabled us to gather more precise information compared to 3GLs where these structures are mostly dynamic.

Conceptually, the presented technique could be applied to other 4GLs too. The core elements of the CFG should be the same in a language independent way (e.g. UI handling), but special constructs of the language should require special solutions (e.g. events and raise event handling and the task record loop).

We applied our work in a research project which was carried out in cooperation with our industrial partner to automatically generate test cases and test input for a GUI test automation tool for Magic [8]. Additionally, we targeted the development of a trace analyser tool to support coverage measurement purposes based on our CFG solution. In this project, we created a path analyser and generator tool for traversing the CFG and generating potential execution paths for the test scripts. As we expected, the growth of path space was exponential [15], so we had to apply several pre-filtering techniques over the CFG before or during the generation, although post-filtering was also possible, it was inefficient or very limited. We created an XML based filtering technique where we could select a sub-component of the CFG with the help of the work flow descriptions of the application analyzed (a work flow described a certain functionality with its related programs and tasks). After the filtering we could execute the script generator tool to create test cases for the Magic XPA application that we analyzed. Our results were promising, hence our CFG technique seemed to be useful in supporting automatic UI testing with test script generation and validation via test coverage measurements.

Acknowledgements

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

References

[1] Allen, Frances E. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, July 1970.

[2] Ashley, J. M. and Dybvig, R. K. A practical and flexible flow analysis for higher-order languages. *ACM Trans. Program. Lang. Syst.*, 20(4):845–868, July 1998.

[3] Ayers, Andrew Edward. *Abstract analysis and optimization of Scheme.* PhD thesis, Cambridge, MA, USA, 1993.

[4] Cousot, P. Semantic foundations of program analysis. In Muchnick, S.S. and Jones, N.D., editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

[5] D., Anupam, J., Somesh, L., Ninghui, Melski, D., and Reps, T. Analysis techniques for information security. *Synthesis Lectures on Information Security, Privacy, and Trust*, 2(1):1–164, 2010.

[6] Ferenc, Rudolf, Beszédes, Árpád, and Gyimóthy, Tibor. Fact Extraction and Code Auditing with Columbus and SourceAudit. In *Proceedings of the 20th International Conference on Software Maintenance (ICSM 2004)*, page 513. IEEE Computer Society, September 2004.

[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, October 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 2011)*, pages 248–259, 2011.

[9] Horwitz, S., Pfeiffer, P., and Reps, T. Dependence analysis for pointer variables. *SIGPLAN Not.*, 24(7):28–40, June 1989.

[10] Jones, Neil D. Flow analysis of lambda expressions (preliminary version). In *Proceedings of the 8th Colloquium on Automata, Languages and Programming*, pages 114–128, London, UK, UK, 1981. Springer-Verlag.

[11] Kennedy, K. and Allen, J. R. *Optimizing compilers for modern architectures: a dependence-based approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[12] Kiss, Ákos, Jász, Judit, Lehotai, Gábor, and Gyimóthy, Tibor. Interprocedural static slicing of binary executables. In *Proc. Third IEEE International Workshop on Source Code Analysis and Manipulation*, pages 118–127, September 2003.

[13] Kowalkiewicz, M., Lu, R., Bäuerle, S., Krümpelmann, M., and Lippe, S. Weak dependencies in business process models. In Abramowicz, Witold and Fensel, Dieter, editors, *Business Information Systems*, volume 7 of *Lecture Notes in Business Information Processing*, pages 177–188. Springer Berlin Heidelberg, 2008.

[14] Lam, M. S. and Wilson, R. P. Limits of control flow on parallelism. *SIGARCH Comput. Archit. News*, 20(2):46–57, April 1992.

[15] Memon, Atif M. An event-flow model of gui-based applications for testing: Research articles. *Softw. Test. Verif. Reliab.*, 17(3):137–157, September 2007.

[16] Midtgaard, Jan. Control-flow analysis of functional programs. *ACM Comput. Surv.*, 44(3):10:1–10:33, June 2012.

[17] Muchnick, Steven S. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[18] Nagy, Csaba, Vidács, László, Ferenc, Rudolf, Gyimóthy, Tibor, Kocsis, Ferenc, and Kovács, István. Complexity measures in 4GL environment. In *Proceedings of the 2011 international conference on Computational science and Its applications - Volume Part V*, pages 293–309. Springer-Verlag, 2011.

[19] 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 the 15th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 343 –346, 2011.

[20] Rech, J. and Schäfer, W. Visual support of software engineers during development and maintenance. *SIGSOFT Softw. Eng. Notes*, 32(2):1–3, March 2007.

[21] Shivers, O. Control flow analysis in scheme. *SIGPLAN Not.*, 23(7):164–174, June 1988.

[22] The Institute of Electrical and Eletronics Engineers. IEEE standard glossary of software engineering terminology. IEEE Standard, September 1990.

[23] Tip, Frank. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.

[24] Vanhatalo, Jussi, Völzer, Hagen, and Leymann, Frank. Faster and more focused control-flow analysis for business process models through sese decomposition. In Krämer, BerndJ., Lin, Kwei-Jay, and Narasimhan, Priya, editors, *Service-Oriented Computing – ICSOC 2007*, volume 4749 of *Lecture Notes in Computer Science*, pages 43–55. Springer Berlin Heidelberg, 2007.

[25] Visser, W. and Păsăreanu, C. S. and Khurshid, S. Test Input Generation with Java PathFinder. *SIGSOFT Softw. Eng. Notes*, 29(4):97–107, July 2004.