Visualization Support for FPGA Architecture Exploration

by

K. Nasartschuk, K. B. Kent and R. Herpers

TR 11-213, December 1, 2011

Faculty of Computer Science
University of New Brunswick
Fredericton, NB, E3B 5A3
Canada

Phone: (506) 453-4566
Fax: (506) 453-3566
Email: fcs@unb.ca
http://www.cs.unb.ca

Abstract

Nowadays Field Programmable Gate Arrays (FPGA) are used in many fields of research, e.g. to create prototypes of hardware or in applications where hardware functionality has to be changed more frequently. Boolean circuits, which can be implemented by FPGAs are the compiled result of hardware description languages such as Verilog or VHDL. Odin II is a tool, which supports developers in the research of FPGA based applications and FPGA architecture exploration by providing a framework for compilation and verification. In combination with the tools ABC, T-VPACK and VPR, Odin II is part of a CAD flow, which compiles Verilog source code that targets specific hardware resources. This paper describes the development of a graphical user interface as part of Odin II. The goal is to visualize the results of these tools in order to explore the changing structure during the compilation and optimization processes, which can be helpful to research new FPGA architectures and improve the workflow.

Contents

List of Figures

List of Tables

1 Introduction

Field Programmable Gate Arrays became and become more and more important during the last decades in different fields of hardware research areas. Reprogrammable hardware is needed in fields, where requirements often change and therefore adjustment in the circuits is needed. Usually hardware functionality is fix after it was produced. FPGAs can be changed at any point. Another field, where FPGAs are very useful is hardware prototyping. The possibility to test, debug and evaluate hardware before much money has to be spend on the production of hundreds or even thousands of devices is more than useful.

Hardware description languages such as Verilog HDL[1] and VHDL[2] were developed to describe hardware structures. Development of the languages is still an ongoing process as the requirements change. FPGA devices need to be described more in detail as they offer more functionality than original devices. Programmable structures are fully connected gates, which can be programmed according to desired functionalities.

So called netlist files are generated by compilers and used to represent the structure of an FPGA device. These netlists are read by optimization tools to increase the efficiency of a circuit. The resulting netlists are later used to evaluate the optimization.

Finding errors and locating them is another difficulty which has to be faced in the evaluation and development process. Though benchmarks and compilers can provide some information, which is needed in this process, the main complexity still has to be done by hand. For this, task simulations of the netlist file on a virtual FPGA device and/or by visual exploration tools are used.

Odin II is a tool, which supports developers in the research of FPGA based applications and FPGA architecture exploration by providing a framework for compilation and verification. In combination with the tools ABC, T-

VPACK and VPR, Odin II is part of a CAD flow, which compiles Verilog source code that targets specific hardware resources.

This report describes the development of a graphical user interface(GUI) for Odin II, which visualizes netlists, which are produced during the workflow Odin II is part of. The visualization aims to improve the productivity of development, support the evaluation process, improve the work flow and to help the research of new FPGA architectures.

The paper is divided into seven chapters. Chapter 2 introduces the fields of FPGA researcg and GUI development to provide an idea of basic structures and current developments in the fields. The tools, which are related to Odin II and the workflow it is part of are presented in Chapter 3.

The basic design, the structure of the application and the use cases, which should be fulfilled by the application are defined in Chapter 4. The implementation of this design, a detailed description of classes and methods can be found in Chapter 5. In Chapter 6 the current state of the GUI is evaluated according to the use cases and using a think aloud test.

2 Basics

2.1 FPGA

2.1.1 FPGA Introduction and History

During the 1990s the development of hardware became more and more risky as the production of chips ranged from \$20,000 to \$200,000[3]. Application Specific Integrated Circuits(ASICs) needs a lot of time to be produced and factories to produce a big amount of chips at the same time. The goal of the hardware industry was to reduce the costs of hardware prototypes and the time, which passes from programming to testing. A solution for this problem was to use a technology which was introduced in 1986 by Xilinx Inc. named "Field Programmable Gate Arrays (FPGA)"[3][4].

There are two main advantages, which lead to the conclusion, that FPGA technology is a very good solution for prototyping hardware. At first prototypes can be produced in small quantities and "no facility must be tooled to begin production of a mask-programmed device which incurs a large overhead cost"[3]. The second reason is that the programming process is finished within minutes and can be tested, "whereas mask-programmable devices must be manufactured by a foundry over a period of weeks or months"[3].

An FPGA is basically a collection of logic blocks on a hardware device, which can be programmed and connected. There are many different architectures, which place the logic blocks and possible interconnections. As can be seen in Figure 2.1 the basic design of an FPGA consists of logic units, interconnection ressources and I/O-cells, which are arranged as a two-dimensional array. The FPGA design is always "a trade off in the complexity and flexibility of both the logic blocks and the interconnection resources[3]". The most common design is called *island-style* and consists of evenly distributed I/O-cells surrounding a basic structure of logic blocks. The main advantage of this structure is the flexibility, which makes possible, that a single application, which is programmed in such a "design can be used in multiple products all with different capacity, pin count, package etc[5]". The structure can also be used in other applications by embedding the design with all its logic blocks and their interconnections[5]. Figure 2.2 shows an example for an *island-style* designed FPGA, which looks very much the same as the conceptual FPGA in Figure 2.1.

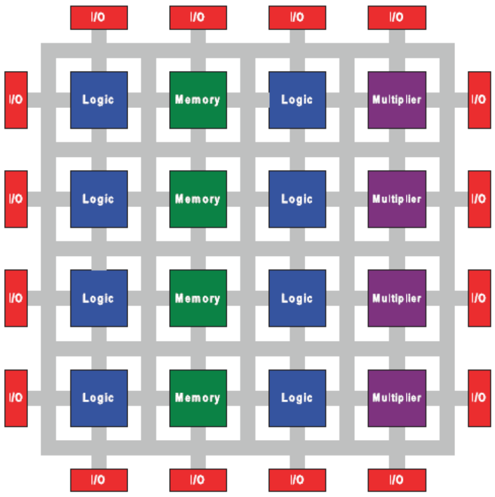

Figure 2.1: A Conceptual FPGA[6]

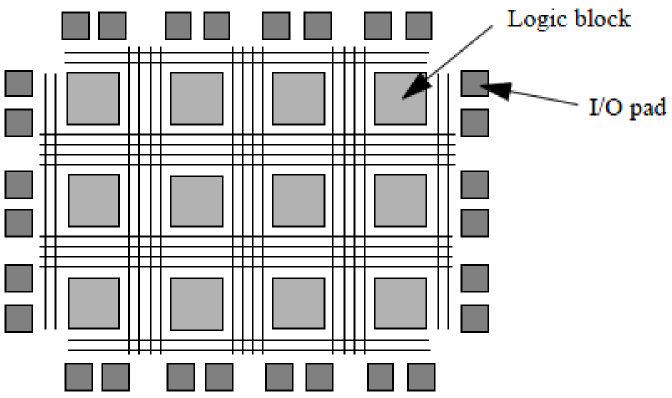

Figure 2.2: "Island-Style"[4]

There is a variety of manufacturers which produce FPGAs. The architecture and design differs very much. Logic units on an FPGA differ and can represent different levels of granularity. Predefined logic blocks are often embedded on FPGA devices. A more granular logic block is more effective in speed, but less flexible compared to a block with smaller granularity[7].

The first FPGA, which was presented by Xilinx contained "64 logic block and 58 inputs and outputs [6]". Nowadays the size of FPGAs grows every year and in the year 2007 it contained "approximately 330,000 equivalent logic blocks and around 1100 inputs and outputs [6]". This leads to different problems. One of the problems is addressed in the next Chapter, which is the architecture of such a device. Other challenges are the programming and understanding of processes that happen on such a device.

2.1.2 FPGA Architecutre

There are two main categories of architecture problems in the development of a FPGA, in which challenges for research in this field can be grouped. The main groups are *Logic Block Architecture* and *Routing Architecture.* Both contain trade offs which are responsible for results that are achieved by FPGA devices.

Logic block architecture

Programming of a FPGA, which is mostly done by a program considers different aspects, which influence the location of different logic blocks. The main structure of a work flow in this area is shown in Figure 2.3. Benchmark circuits and the FPGA architecture are used to create a circuit implementation, which can be measured by the benchmarks itself but also by such factors as area consumption, speed and power consumption. This is a practical approach, which is mostly used for logic block architecture purposes. It can also be done theoretically, but that is more useful and common in researching routing architectures[6].

Area-efficiency is an important metric on a FPGA as the size of a FPGA is not only a matter of costs[6], but also gives the maximal distance between two logic blocks, which could be connected to each other. This has to be

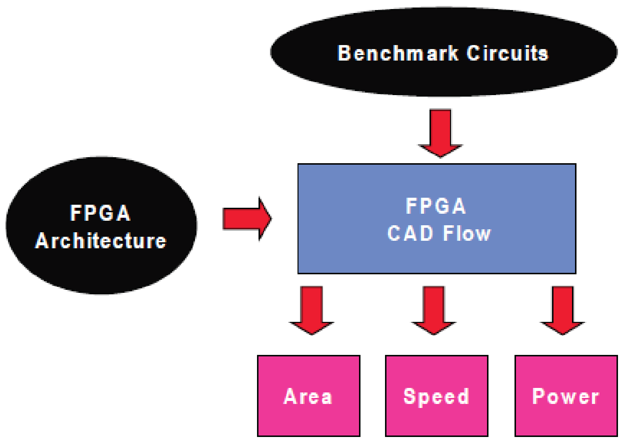

Figure 2.3: FPGA "Architecture Exploration Empirical CAD Flow[6]"

considered when a clock period is determined. Speed is an trade-off on an FPGA as the amount of higher logic blocks increases as the interconnection count is lower and functionalities are optimally located[6][3]. The price, which has to be paid for blocks with higher logic is the flexibility of a device as all parts remain on a FPGA device even if it is not used. Instead, it could be a logic block, which could be flexibly programmed. Another issue in integrating blocks with higher logic is the fact, that delays of blocks are not unified. A block, which contains a more complex logical structure needs more time to establish a stable output[6].

The last and most vague trade-off is the power consumption in FPGAs. Power consumption on a device in general can be divided into "dynamic power and static power [6]". Static power is the power consumption of the device even if no signals are processed whereas dynamic consumption also contains the signals during computation.

Routing Architecture

Routing Architecture on a FPGA not only determines the speed of the user circuit, but also the variety of circuits, which can be implemented on this device. As the trade offs for logic blocks and the amount of interconnections is increasing the costs of the device, it also minimizes the routing minimization afford.

The main factors, which are determined by the manufacturers in global routing are the amount of wires, which are placed between the logic blocks and the amount of switches, which can be programmed to route the interconnections[6].

There are different global routing architectures used on devices. Two of them are *hierarchical routing architectures* and *island style architectures*. Hierarchical routing architecture groups logic blocks, which can be connected to each other at a very low level. Connections between groups are accomplished on a higher level. Such an architecture is shown in Figure 2.4, where a hierarchy based on three levels is demonstrated.

Island-Style routing architecture as shown in Figure 2.2 creates a two-dimensional mesh between logic blocks with switches on the intersections of wires. These switches can be used to create interconnections between logic blocks. ”Currently, most commercial SRAM-based FPGA architectures use island-style architectures[6]”. There are also effords to create an extended island-style architecture by adding an additional dimension to the wire mesh. This would allow more connections and decrease the area complexity of an FPGA, but according to Schmit ”commercial three-dimensional fabrication techniques, where the transistors are present in multiple planes, do not exist[5].”

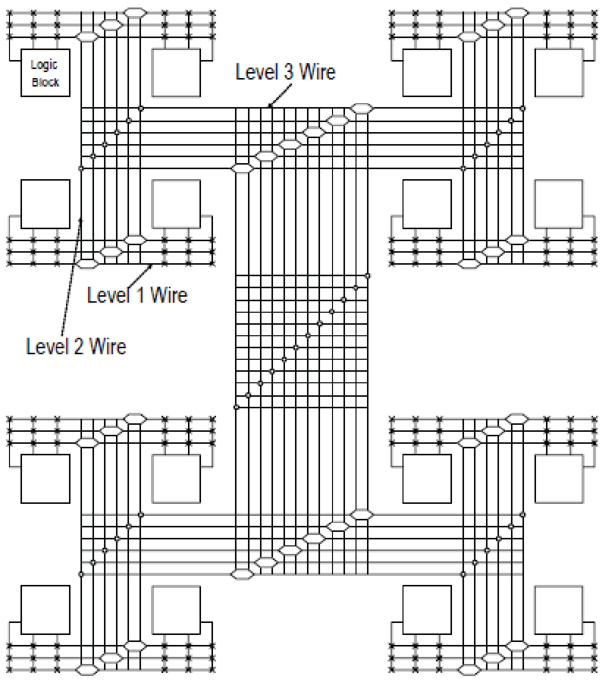

Figure 2.4: Hierarchical Routing Based on Three Levels[6]

2.1.3 FPGA Architecture Exploration

Exploration of FPGA architectures can be done in different ways. There are methods to explore an architecture based on benchmarks. Those can be used to assist during the development process. An example for such a framework is created in [8].

There are also theoretical analysis tools, which explore the architecture to assist developers as e.g. in [9] in order to find the "best multiprocessor on the FPGA for a target application and optimally map the application tasks and communication links to this micro-architecture"

During the path from a design description in a hardware description language such as Verilog to the logic circuit on the FPGA device different tools perform optimizations and changes, which cannot all be explained in regarding the results. To do so it is useful to see the results of processes between the computation steps and track their origin. This helps to understand the functionalities and to find room for improvement. Visual exploration support has the goal to provide such a tool, which can be used by human users to navigate in the logical structure, which is also a goal of this project.

There is a variety of exploration tools, which are available on the market created by the manufacturers of the devices. Those are for example Altera[10] or Xilinx[11] for FPGA. Those tools offer benchmarks and visual exploration tools, which are optimized on the devices. The main problem is, that they are only for commercial use with the devices they are made for.

Also Open source tools are available, which can be used to simulate and therefore explore digital designs. Some of them are: Icarus[12], Veriwell[13] and FreeHDL[14], "which mainly concentrate on simulation and analysis of digital designs[15]."

2.2 GUI Development

2.2.1 General Information

Computer applications perform actions or computations, which the user has ordered them to do. Most of the applications are very specific and users are capable of using them by entering commands on a command console. The basic flow of such an application can be seen in Figure 2.5a. It just receives the input parameters and terminates after results are delivered.

The advantages of such an application is, that no resources are needed

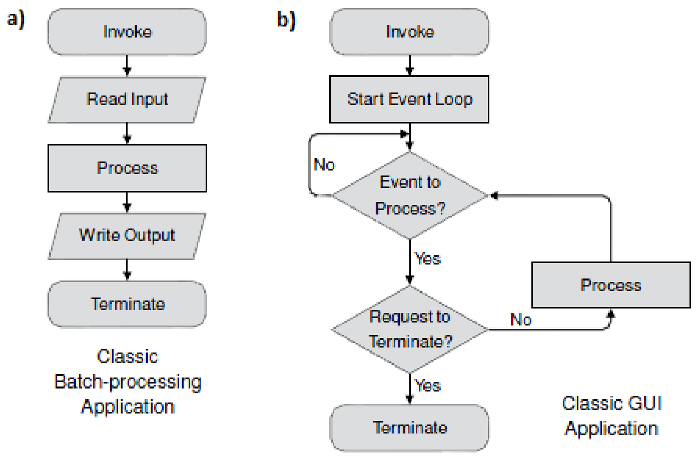

Figure 2.5: GUI Application Flow and Classic Batch-processing Application in Comparison[16]

in addition to the actual computation, that computations can be performed automated on servers etc. The disadvantages of classical batch-processing application is the interaction with a human user. The user has to learn how to address the program and how to interact with it. To simplify this process programs with high human contact are offering graphical user interfaces(GUI), which encapsulate the complexity of the program into a intuitive interface.

The structure of such an application differs from a batch application(Figure

2.5) as it adds an event loop, which awaits user commands and passes this command to the actual program. Such graphical user interfaces can be implemented using different languages and pre-compiled libraries. Some of them, which were considered in the development of this project are shortly introduced in this Chapter.

2.2.2 QT

The first public version of Qt(pronounced *cute*) was published in May 1995[17]. It was developed at the Norwegian Institute of Techonlogy in Trondheim by two students, Haavard Nord and Eirik Chambe-Eng. The development began in the late 90s, but was published after the students founded the company Quasa Technologies, which later became Troll Tech and is known today as Trolltech[17]. Trolltech is a subsidiary of Nokia.Graphical interfaces developed using Qt are "KDE, the web browser Opera, Google Earth, Skype, Qtopia and OPIE"[18].

The library consists of modules, which functionality is clear structured and separated to be able to use only modules, which are needed for a specific application. Dialogue based and also window based applications are possible[19].

The library is written in C++ and heavily utilizes on object oriented standards. Parts of the application implement classes, which already exist in the library started by the main class `QObject` and becoming more and more specific. Some exemplary classes are for example `QLine` or `QPolygon`.

Qt consists of 22 modules, which are used for different purposes. The `QtGui` module consists of classes, which are used to create graphical user interfaces, `QtCore` is a collection of core non-graphical classes, which are used by other models.

Xml, WevKit, OpenGL, Svg can also be handled using modules, which are provided by Qt. A full list of modules and an API can be found on the website of Qt[20].

Development in Qt in the basic version only allows C++, but many projects created bindings of the library. The reason was mostly to be able to use the functionalities of Qt in different programming languages. Qt bindings for more than 15 languages are available. Examples include PyQt(Python), QtRuby(Ruby) and QtLua(Lua).

3 Related Work

3.1 CAD workflow

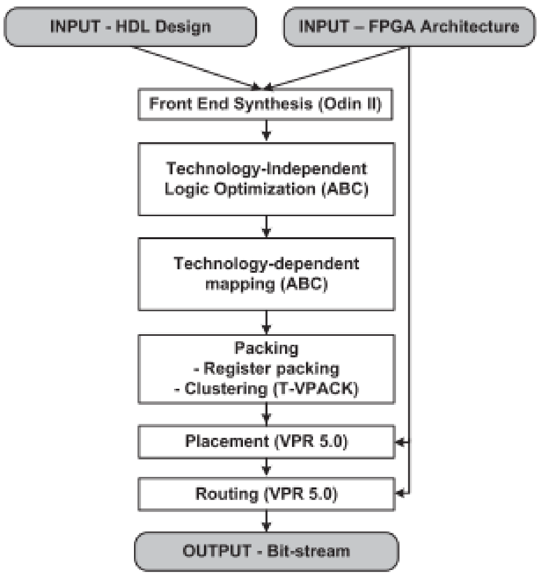
INPUT - HDL Design
INPUT – FPGA Architecture
Front End Synthesis (Odin II)
Technology-Independent Logic Optimization (ABC)
Technology-dependent mapping (ABC)
Packing
- Register packing
- Clustering (T-VPACK)
Placement (VPR 5.0)
Routing (VPR 5.0)
OUTPUT - Bit-stream

Figure 3.1: "A VPR5.0 CAD flow[6]"

Developers create FPGA applications using hardware description languages such as Verilog HDL[1] or VHDL[2]. This hardware description code is used to create an (in the best case) optimized digital circuit for a specific FPGA architecture.

Basically the goal of the CAD flow is to combine different tools in order to perform the basic steps(Figure 3.1):

- Front End Synthesis
- Optimization
- Placing
- Routing

The front end synthesis and compiling is done using Odin II, which is introduced in Chapter 3.2. Optimization and sub procedure *mapping* is part of the ABC tool, which "can optimize/map/retime industrial gate-level designs with 100K gates and 10K sequential elements for optimal delay and heuristically minimized area in about one minute of CPU time on a modern computer[21]". Placing and routing on the FPGA device is performed by the VPR 5.0 toolset. The structure and basic functionality of VPR 5.0 is part of Section 3.3.

3.2 Odin II

Odin II was introduced in [15]. It is "a framework for Verilog Hardware Description Language(HDL) synthesis", which is an improved version of Odin[22].

The main purpose of ODIN and ODIN II is "the front-end conversion of a HDL design into a netlist of basic gates and more complex logic functions[22]"

In order to convert a Verilog HDL design to the according netlist Odin

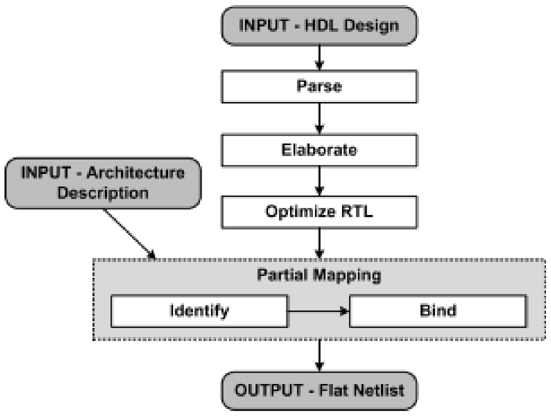

Figure 3.2: "The Processing Stages in Odin II[15]"

II performs steps, which are shown in Figure 3.2. Two inputs are needed during the procedure. The first one is the Verilog HDL design, which is parsed to create an "abstract syntax tree[15]. The second is the FPGA architecture description. As the architecture is changed, the infrastructure of the design is changed, which allows one to apply changes very easily[15]. In comparison to other CAD flows, the internal data structure of Odin II is not focused on saving memory but on speeding up possible changes. This means in particular, that wires are stored as objects and not embedded in the rest of the infrastructure. This allows to change interconnections without unnecessary changing of many lines in the netlist.

The grey highlighted section in Figure 3.2 shows the partial mapping step of Odin II, which needs the second input file. This hardware resources description file contains for example the amount of basic logic blocks, their interconnections and other resources. An example which is also mentioned by Jamieson et. al[15] is a 8x8 multipler, which is used in the design. The tool has to find the 8x8 multiplier in the circuit description and map them onto each other. This task "is even more challenging when there are a restricted number of hard circuits available on the target FPGA[15]."

3.3 VPR5.0

VPR([23]) is a toolset, which was developed at the University of Toronto in the late 1990s and is widely "used to perform FPGA architecture and CAD research[24]." The main purpose of VPR is to place logic structures on FPGA devices and to perform an optimized routing based on experimental results.

As major development on the VPR toolset stopped in the year 2000, different developers changed and modified VPR for their purposes. Those results are mostly not available as they have not been contributed to the project([24], p1). In [24], VPR 5.0 was introduced and includes the following key features:

- Single Driver Routing Architectures
- Heterogeneous Logic Blocks

- Optimized Circuit Design in released Architecture Files
- Robustness

Each of these features lead to different enhancements in the toolset, which also result in better benchmark performance. VPR 5.0 is used in combination with other tools as described in Section 3.1. In Figure 3.3 the position of VPR in the CAD flow is shown once more and the tasks, which have to be accomplished are demonstrated.

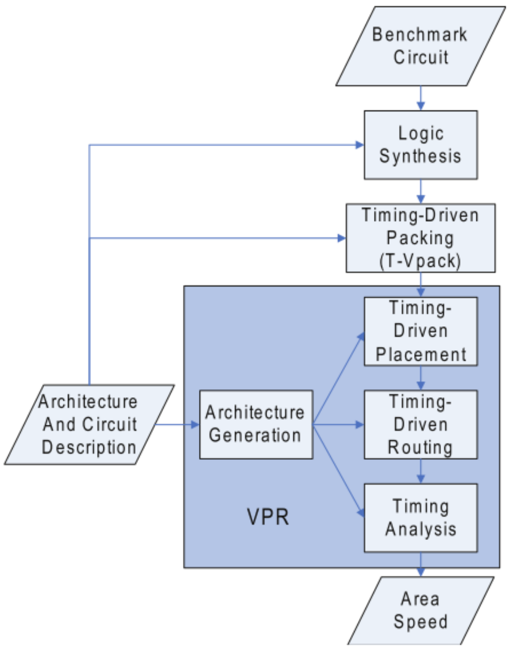

Figure 3.3: VPR Tasks in the CAD Flow[6]

After tools such as ABC or T-Vpack completed the mapping process, the BLIF structure file is passed over to VPR, which performs the placement and routing tasks. The placement and routing processes on the device are influenced by the critical path delay, which is used to measure the quality of an actual implementation circuit[24]. Each architecture, which is created during the computation process is tested by the *Timing Analyser* in order to find the best solution. The placement algorithm is very computation intensive, as it has to fulfil many different constraints.

4 Design

4.1 Problem Specification

The contribution of this project is a visual exploration support software, which can be used in the development and evaluation process of Odin II and its cooperating tools. The main functionality of this application is to visualize a boolean circuit, which is processed in the CAD flow.

The file structure, which is used to create a representation of the boolean circuit is the Berkeley Logic Interchange Format(BLIF[25]), which is used to pass results between Odin II, VPR 5.0 and ABC. This file structure describes modules and logic blocks, their inputs, outputs, interconnections and functionalities. The functionality of logic units is not important for the visual representation as no simulation has to be done by the software.

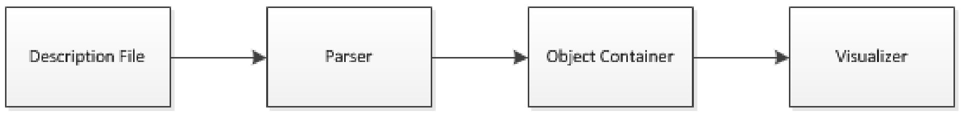

Figure 4.1: Basic visualizer workflow

The circuit visualizer is created to find information, which is stored in the BLIF file, to create objects according to it and visualize them on the screen. The useful information, includes the modules, their inputs, the name of the output and the interconnections between them. The user should be able to recognize those on the screen represented by nodes of a connected graph.

This R&D report describes the implementation and design, which was needed to create the basic structure of the software, its classes, factory-classes and communication process. Another aspect is to create the graphical user interface as intuitive as possible to offer a front end for the CAD work flow, which allows the user to become familiar with the results and sub results quick and easy.

The actual situation in the CAD flow is that developers use Verilog HDL designs and architectures to synthesize, optimize, place and route the modules on the device. The results of each step are given as a BLIF, which can grow very large. The file structure itself is not created to be human readable. The visualizer provides a tool, which supports the understanding of processes and can be used for debugging purposes.

4.2 Use Cases

The evaluation of the project follows a systematic evaluation of use cases. These use cases are defined to be able to show, which functionalities the software provides and which are not available at this time.

The list of use cases is structured in two main regions: GUI usability, which describes the functions, menus and functionalities of the graphical user interface. The main task of those use cases is to make sure, that the software is as user friendly as possible and can provide the needed functions intuitive. Visualization Functionality describes use cases, which describe the usage of the background functions of the GUI. This includes as well the file handling as also the visualization of modules, connections etc.

- GUI usability
  - Open a BLIF file per button.
  - Open a BLIF file in context menu.
  - Rearrange menus as needed.
  - Mark visualized modules.
  - Mark interconnections between modules.
  - Highlight modules or connections.
  - Rearrange modules using the mouse.
  - Zoom in/out of the graph.
  - Close the application properly.
  - Interconnections can be added and deleted.

  - Text can be added in the graph.
  - Items can be added to the graph.
  - Items can be deleted from the graph.
  - A new graph can be opened after another one was explored.

- Visualization Functionality
  - Class structure can be extended by new module types and shapes.
  - Receive error, if BLIF file is corrupt or not found.
  - Recognize the modules correctly.
  - Final State machine of file parser does not have *dead ends.*
  - Recognize the interconnections correctly.
  - Modules arranged without overlapping.
  - Reading in a description file has to be performed in a reasonable time.

4.3 Basic Design

The application mainly consists of two main modules. The first one includes everything, that is needed to visualize. That includes the GUI, the visualization graphic with all modules and functionalities. This module embeds the functionality of the actual computation part of the application, which is responsible for all inputs, which are needed by the application.

As shown in Figure 4.2, each of the modules include functionalities. The File-Parser is responsible for every interaction with the file structure. A *File-Container* is the storage of each module and each interconnection, which were found in the description. Communication between each is accomplished by the *Visualization Advisor* and the *Module Creator.*

Modules are placed in the graph according to their interconnections in the *Module Arranger* and drawn on the *GUI Graphics Module*, which is part

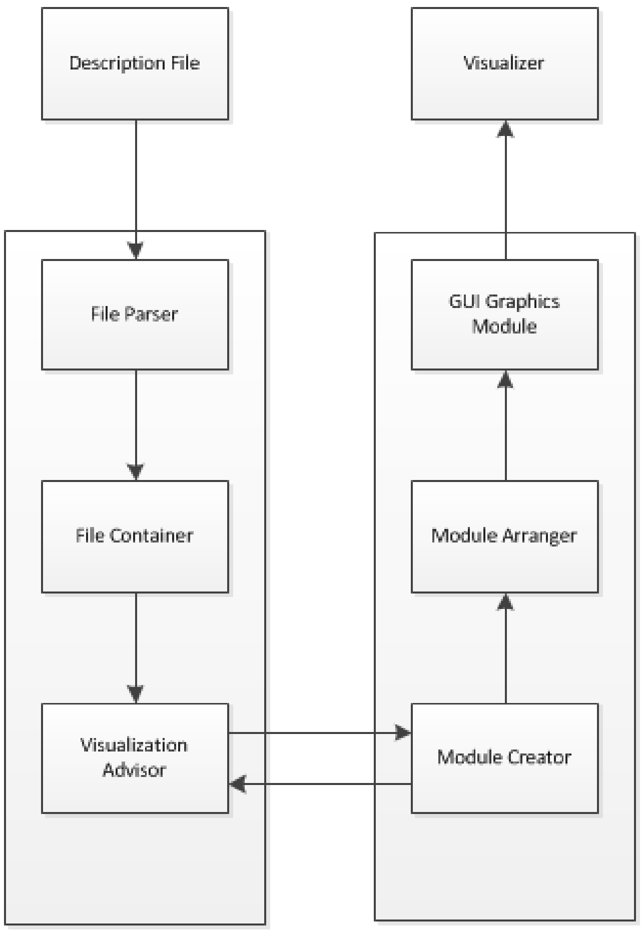

Figure 4.2: Design of Basic Modules in the Visualizer

of the Visualizer. This basic structure is the idea of the final application, whose implementation is described in Chapter 5.

5 Implementation

5.1 Software Engineering

5.1.1 Class Structure

The circuit visualizer is a collection of 6 classes, which combined provide the functionalities which are needed to fulfil the use cases defined in Chapter 4.2. In the next chapters the implementation, functionality and most important parts of the source code are described and explained to provide an idea of the development process, its challenges and functions, which are available at the current development stage.

The classes in Figure 5.1 can be divided into graphical and functional classes. Graphical classes represent objects, which can be seen by the user, whereas functional classes only perform tasks, which are used in the background of the application. The only pure functional class is `Container`, which consists of file-operations, module creation and arrangement computations. A detailed explanation of all classes is provided in Chapter 5.2.

5.2 Circuit Explorer Implementation

5.2.1 Object Classes

A boolean circuit consists of logic blocks, which are connected in order to combine their functionality to compute a result. In order to perform visualization support logic units and connections, which are found in the BLIF file have to be presented to a user. Object oriented principles, which are defined to provide methods of effective software development are used to maximize reuse of code passages and design a code structure, which can be changed without major code refactoring. The structure contains classes, which are created to represent objects. These objects are created according to the BLIF file. There is a direct connection between each object class and the BLIF file structure. Those classes are: `LogicUnit`, `Wire` and `DiagramTextItem`.

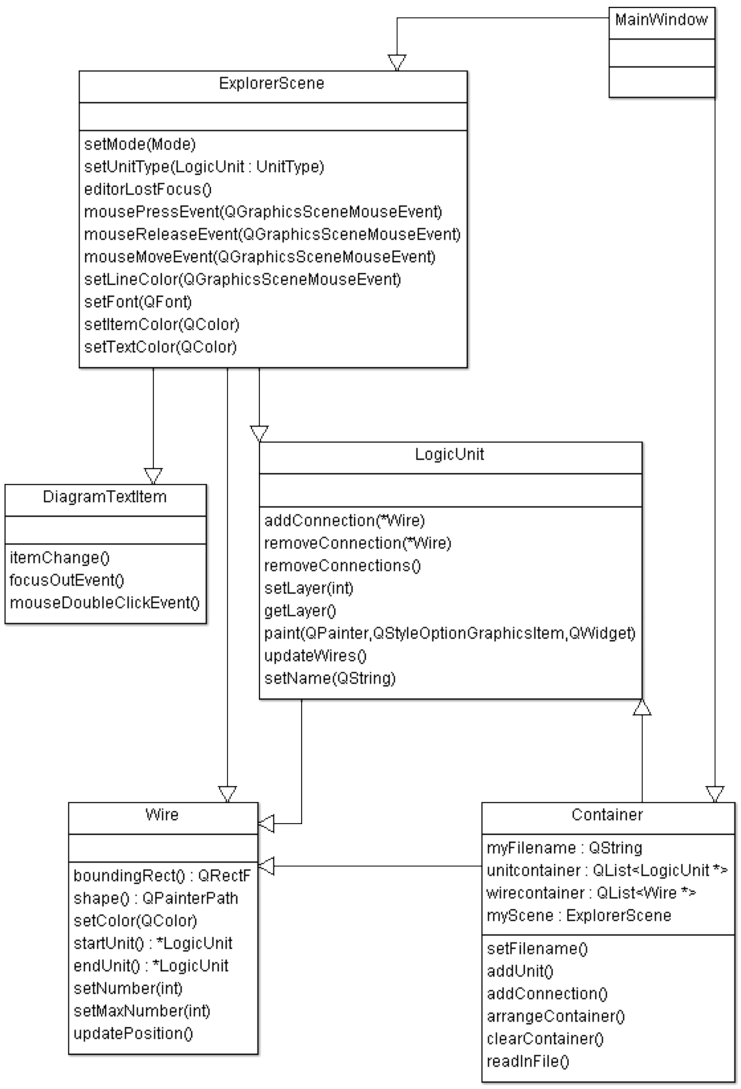

Figure 5.1: Class Diagram of the Application

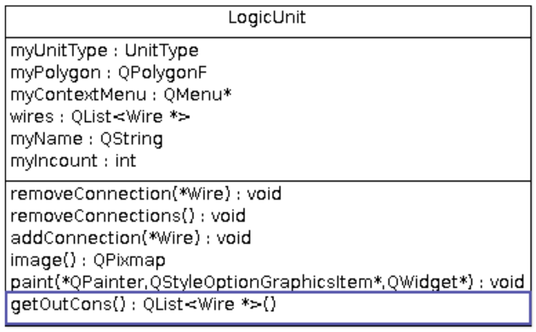
LogicUnit
myUnitType : UnitType
myPolygon : QPolygonF
myContextMenu : QMenu*
wires : QList<Wire *>
myName : QString
myIncount : int
removeConnection(*Wire) : void
removeConnections() : void
addConnection(*Wire) : void
image() : QPixmap
paint(*QPainter,QStyleOptionGraphicsItem*,QWidget*) : void
getOutCons() : QList<Wire *>()

Figure 5.2: LogicUnit Class UML

An instance of `LogicUnit`, stores information, which either is of interest for the user or is needed to visualize a logic block. The name and the input count are needed to identify the block, which was found in the BLIF file. The shape of an input object differs to other logic blocks in the circuit by its shape as shown in Figure 5.3. New shapes for different types of logic blocks can be added easily.

The basic shape, which is implemented to represent a logic block is a

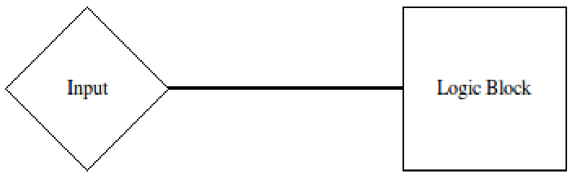

Figure 5.3: Different Shapes for Inputs and Other Logic Blocks

simple rectangle, which has specific dimensions. This is not a restriction, as each type of `LogicUnit` is able to define its own Polygon, which dimensions and shape can vary. Inputs for example can look different than outputs do.

Each `LogicUnit` object includes a container, with all connections, which are bounded to it. As in Qt only the object, which is actually changed (position, attributes) is being notified and all other objects stay in the original setting. This causes problems as the ends of connections have to be connected to the actual `LogicUnits`, where they start or end. If a `LogicUnit` was moved, the connection will not adjust.

This fact requires the `LogicUnit` to notify all instances of `Wire` in its container about a change of position, which causes them to be repainted. This procedure reduces the time which is needed to adjust to a position change. Another strategy is not to store any references to connections, but just iterate through all of them and check, if the ending points are still where they are expected to be. This would be more memory cost effective, but the computation time could grow in the worst case to $O(n^2)$ with respect to the block count, whereas the container strategy computation time only grows in $O(n)$. Another procedure, which favours use of a container is when an item is deleted, as all connections have to be erased, which is explained in more detail in Section 5.2.2.

Other functions, which are of interest are `image()`, `paint()` and `boundingRect()`. `image()` creates a bitmap presentation of the polygon of the actual `LogicUnit` type, which is later used to show the shape of it on a button in the graphical user interface. The size of the bitmap is always 250×250. Polygons, which are bigger may not be represented properly.

`paint()` and `boundingRect()` are methods, which are part of every instance and implementation of `QGraphicsItem`. Their function is to describe the behaviour and visual representation of each object. Both methods are closely related. The method `paint()` describes the shape of an object. This can be a polygon of any shape. Also complex Figures can be created using a combination of different shapes, colors, texts etc. In case of a `LogicUnit` the Figure is a rectangle, which has a solid surrounding and which filling color can be changed. The name of the object is displayed inside the rectangle.

`boundingRect()` returns a rectangle, which surrounds the whole figure. This is very important as this is the area, which is renewed when the object is changed, added or removed. The bounding rectangle is usually kept as small as possible, as the refreshing process takes time. At the same time it needs to be big enough to surround all parts of the object to prevent artifacts and to ensure that the whole object is visible.

The class `Wire` represents interconnections between `LogicUnits`. There

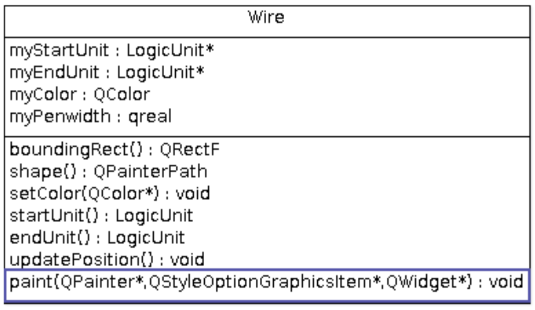

Figure 5.4: Wire Class UML

are different reasons, which led to the implementation of this class. There are two reasonable methods to create the data structure of a connection between two logic blocks. One method is to store references of `LogicUnits` in instances of `LogicUnits`. This is very easy and fast to implement and would not need much memory to store this information. The problem in such an implementation is the fact, that also all attributes, which can be seen by a user would have to be stored in `LogicUnits`. These attributes are for example the color of a connection, its width etc. One of the principles of object oriented programming is separation, which means, all object have to be created for only one purpose. A `LogicUnit` is only a representation of logic blocks. If interconnections are more than only references, they have to be extracted.

Also methods, which are needed for an interconnection would be less readable and functional using this method. A bounding rectangle of a `LogicUnit`, which has many interconnections would have to find the longest and create an area surrounding the whole figure. This would cause the application to refresh a very big area, which is not needed and may cause trouble during the development and extension process.

These arguments nullify the main argument to use references stored in `LogicUnits` and lead to the implementation of the class `Wire`. Two parameters are needed to create such a *Wire*: a starting and an ending `LogicUnit` which have to be connected. The advantages of such a data structure are beside the implementations of a small bounding rectangle also the information that can be stored in this object. Not only the color and width of the connection, but also the custom shape, input number of the wire in the ending point etc. can be stored and used to represent the connection. Another extension, which can be added later is the exact line in the BLIF file, which caused the creation of the connection to give the user additional exploration possibilities.

The shape of the wire is created to be rectangular. This is mostly used in representations of boolean circuits as wires that cross are rectangular and it is easier to determine which wire follows which path. An important piece of information for a connection is the number of inputs which lead into its ending point and the number, which is assigned to the input itself. Using this information it is possible to align all inputs of the `LogicUnit` equidistant to be visible. This allows to see how many inputs a logic block contains.

The last class, which represents an object is the class `DiagramTextItem`. This class has a cosmetic purpose and was part of the Qt tutorials, which were included in the development environment. This class allows the user to create text passages in the visualization in order to add notes or comments, which are helpful during the exploration process. Font, text size and color can be adjusted using the text tools in the tool bar.

5.2.2 GUI Classes

This section describes the functionality and tasks of all classes, which are responsible for the graphical user interface, providing their functions, layout and other interactions with the user. In particular the classes `MainWindow` and `ExplorerScene` are introduced.

The presentation part of the graphical user interface consists of two main parts, which are strictly divided into two classes. The class `MainWindow` as the name already implies, is a description of the whole window including its menus, buttons, layers etc. The structure of the application is required to be as intuitive as possible.

The main window has four areas. The first is the menu area, where the context menu and functional buttons are located, which offer the functions, which can be used. Those are mainly the functions to change colors, fonts, zoom level etc. The main function, which the program was developed for is the *open file* dialogue, which initiates the whole procedure of parsing, visualizing and leads to the ability to explore the architecture described by the netlist file.

Four methods are responsible for the appearance of the application:

- `createActions()`
- `createToolBox()`
- `createMenus()`
- `createToolbars()`

Those methods are called during the creation of the `MainWindow` object. Actions are Events, which are handled by the application. Each menu entry or button starts an event, which performs an action. Such events in case of the architecture explorer are *bring module to the front layer*, *open file*, or *exit program*. Qt offers a variety of parameters, which can be used for actions. An example declaration of an action is:

```
openFileAction = new QAction(tr("&Open BLIF"), this);
openFileAction->setShortcut(tr("Ctrl+O"));
openFileAction->setToolTip("Opens and visualizes a file in the BLIF format");
connect(openFileAction, SIGNAL(triggered()),
        this, SLOT(openFile()));
```

The last line defines, that the action `openFileAction` is bound to method `openFile()` and if this action is triggered, `openFile()` is executed. This structure allows the usage of existing methods for actions and also sharing methods for different actions. This is used for example if there is a menu entry to open a file and also a button which offers the same.

The toolbox in the application contains only two buttons at the moment as there is only one type of logic block implemented. One button allows to add a new instance of `LogicUnit` into the visualization, which name is per default *MouseAdd*. The second button is used to add notes or other text passages into the visualization. Some possible case to use this feature is to name a connection by writing a name on top of it.

The menus and the tool bar are very similar as they contain the same functions which can be used. The difference is, that menu items also provide short cuts and are more structured, whereas buttons can be accessed very fast and recognized by icons. This structure becomes very useful when the application is extended by additional functions and the menus get big. Only the main functions which are used very often have to be represented on the tool bar to save the user time.

Using Qt and its libraries interactions with the application is easier as tool bars can be dragged around if needed. Every button group can be rearranged or even detached from the main window if needed.

Another important function in `MainWindow` is `deleteItem()`. In the specific structure of this application and its object, it is necessary to make sure that all object are deleted correctly. It is not enough just to make them disappear from the visualization part of the graphical representation.

In case the object, which has to be deleted is an instance of `Wire` it has a starting and an ending object(`LogicUnit`). These objects are aware of the connection and always try to notify them about positioning changes. If such an object just disappears, it cannot be notified any more and would lead to a crash of the application. That means, all references to a `Wire` object have to be removed once a delete command is executed for a connection.

The same argument can be used in case a `LogicUnit` has to be deleted. If there are `Wire`s connected to it, they will still be there if the graphical representation disappears. This means also the connections of the `LogicUnit` have to be deleted. The order in which the objects are deleted is fixed, as connections can only be deleted while the starting and ending points are still existent.

To get a better overview of the graph, the zoom level of the scene can be changed. This function is located in the button section of the GUI. There are different zoom levels predefined, which can be chosen by the user. A change of zoom level causes the method `sceneScaleChanged()` to adjust the parameters.

```
QMatrix oldMatrix = view->matrix();
view->resetMatrix();
view->translate(oldMatrix.dx(), oldMatrix.dy());
view->scale(newScale, newScale)
```

Modules, connections and text items in the shown graph can be marked and highlighted using different colors. The button menu contains drop down menus, which are responsible for each of the objects. If a color is changed in

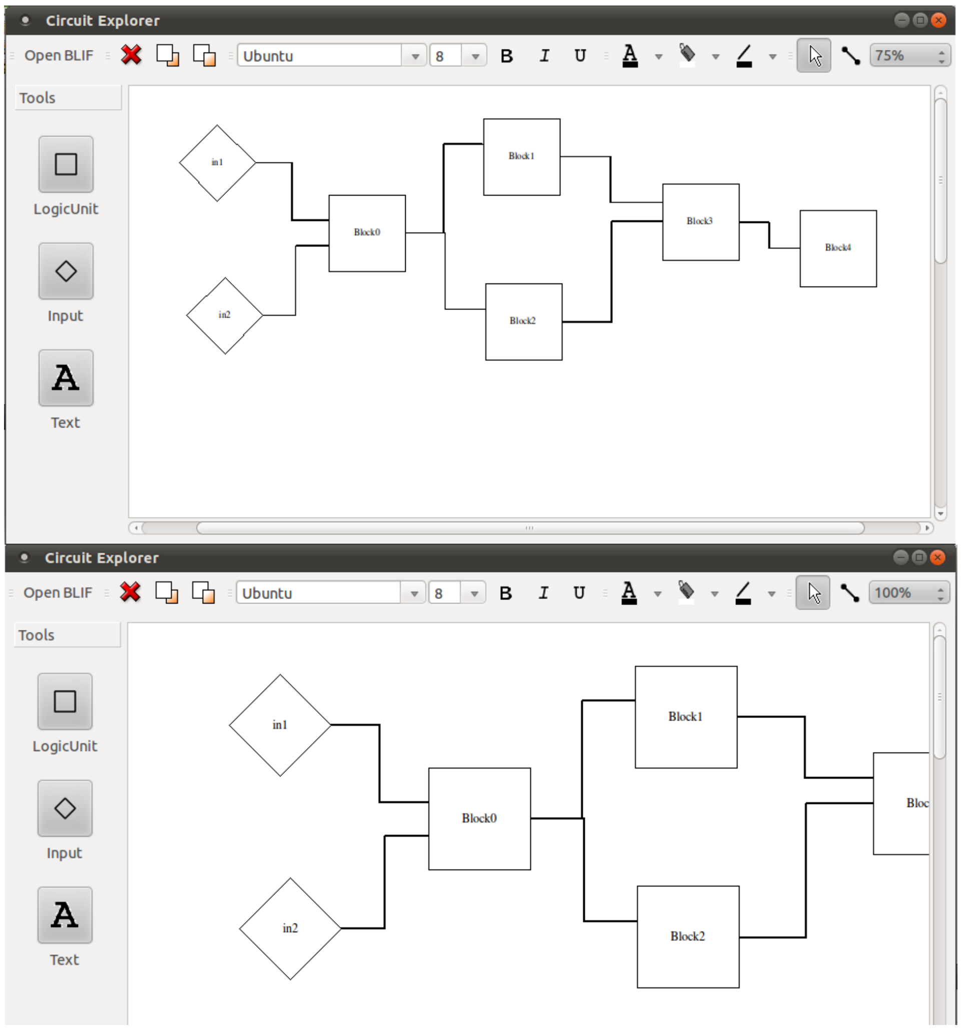

Figure 5.5: Different Zoom Level of One Graph

one of the menus and items are currently selected, the color of those items is changed. The new color is set to the default color. New items, which are inserted manually into the visualizer will use it, until it is changed again. A function, which is created to change the name of a module can be accessed using the right click context menu.

The method, which combines the graphical user interface with the functional class is `openFile()`. It is connected to `openFileAction`. The BLIF file which the user wants to be visualized is chosen using an open file dialogue which opens the home folder of the user logged in per default. It also offers a navigation bar on the left side to be able to access other important locations on the computer such as the root path. To increase usability the file browser shows only supported file formats (*.blif at the moment). This feature can be changed to show all files inside folders.

The functions that create and visualize `LogicUnits` are provided by the class `ExplorerScene`, which inherits `QGraphicsScene`. `QGraphicsScene` is used to manage 2 dimensional objects on the `QGraphicsView` in the application. The actions, which change, move, create and delete any kind of items in the visualization are passed to an instance of `ExplorerScene`.

All actions in the main window, which were described earlier are passed to `ExplorerScene` to modify existing or to create new items. The simple functions `setLineColor`, `setTextColor`, `setItemColor`, `setFont` and `setUnitType` are used if the events require so.

Mouse events are also handled in this class. Most of the interaction between the application and the user are done using the mouse. As the mouse only provides two buttons and a position, a mouse click can mean different things depending on the location of the cursor and the objects which are located on this position.

There are three mouse handling methods which perform case sensitive ac-

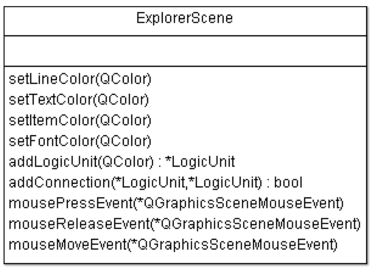
ExplorerScene
setLineColor(QColor)
setTextColor(QColor)
setItemColor(QColor)
setFontColor(QColor)
addLogicUnit(QColor) : *LogicUnit
addConnection(*LogicUnit,*LogicUnit) : bool
mousePressEvent(*QGraphicsSceneMouseEvent)
mouseReleaseEvent(*QGraphicsSceneMouseEvent)
mouseMoveEvent(*QGraphicsSceneMouseEvent)

Figure 5.6: ExplorerScene Class UML

tions. If no case is fulfilled the events are passed on to perform their usual functionalities. The first one is executed if the mouse is moved during a left click. The method `mouseMoveEvent()` checks if the user intends to create a manual connection between two `LogicUnits`. If this is the case, a solid line is painted starting at the origin of the movement and ending at the actual position of the mouse to show which items are to be connected.

The methods `mousePressEvent` and `mouseReleaseEvent` in combination can perform three different actions. They can create a connection between different `LogicUnits`, where the `mousePressEvent` saves the starting point of the connection and `mouseReleaseEvent` finds the ending point. A mouse click can also mean to create a `LogicUnit` or a `TextItem` on the actual position of the mouse cursor.

Those methods are useful only if the user is creating modules and connections manually. The usual case in the visualizer is to create those modules automatically. To do so there are two methods, which can be used to create, name and place items. The methods are used in the functional class `Container`, which also processes the BLIF file. One method is developed to create LogicUnits of different types and another to connect them.

5.2.3 Functional Classes

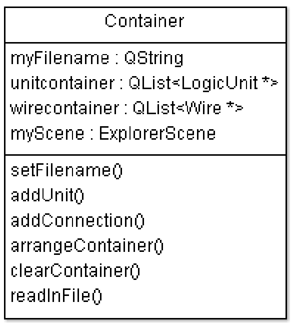

Figure 5.7: Container Class UML

The only functional class is called Container. It is created to perform the file parsing, arrangement of modules and other functions which imply dealing with the structures outside of the application itself. The class can be compared to a user who is creating a graph in the GUI by clicking buttons and creating logic blocks, naming them, creating interconnections and arranging them to be able to explore the structure. The only difference is that this virtual user works fast and automated.

Although all functions can be used as they are needed the usual structure of functions is shown in Figure 5.8. After the user has chosen a file which he wishes to explore, the parsing function of a Container processes the file. It parses twice through the file searching for keywords. Those are in case of *.blif files:

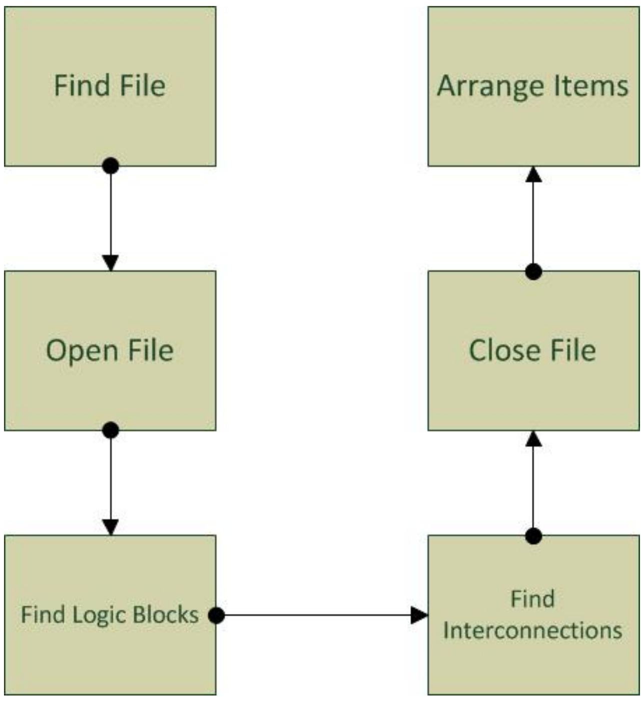

Figure 5.8: File Handling Work Flow

- `.inputs`
- `.names`
- `.subckt`
- `.model`

During the first parse the file is processed to find the logic blocks and models(subcircuits), which are created and connected to each other. The reason for it is that connections need to have existent logic blocks which they connect, otherwise an error occurs and the connection is not created. The second time the file is read is only to find all connections. This procedure just searches for the keyword `.names`, which defines a logic block, its inputs and its name using the following structure:

```
.names input1 input2 input3 ... "unit name"
```

To increase the speed of this process the application uses a hash table, which combines the name of every logic block in the container to a reference to the actual object. This is useful as the the creation of a connection does not require to search every `LogicUnit` for it's name in order to find the starting and ending point of the connection.

There are file structures, which are not recognized yet. A model is part of the parameters of every `LogicUnit` but it does not influence the visualization. Another part of the blif structure is the keyword `.latch`, which allos to create a delay element in a model or to connect a logic block's output with its input. This functionality is also used to arrange the items as all of them are created in the same position and have to be spread to provide an overview on the graph.

The arrangement function of `Container` is a very simple one and needs to be improved. The strategy is to find a straight forwarded structure to make sure the connections always connect logic blocks in a flow from the left to the right. Circles in a circuit influence each other in the positioning process. To avoid an infinitely big graph the iteration count of the arrangement algorithm is limited based on the interconnection and logic block count.

The procedure creates the layers of logic blocks, which are aligned from the right to the left to create a flow, which symbolizes the flow of data. Afterwards each layer is iterated and the logic blocks in each layer are ordered on a vertical line using a small drift which is created to decrease overlapping of connections. The main goal of the current algorithm is to spread all logic block and to show the user the main flow of signal within the circuit. Crossing interconnections are not considered during this process in the current state of the application.

The class also offers functions to delete logic blocks and to clear the whole container if the user wishes to explore another BLIF file. The structure of the class allows to change different steps without the need to change any other passages in any other classes as long as the input parameters and the class definition of `LogicUnit` and `Wire` remain the same.

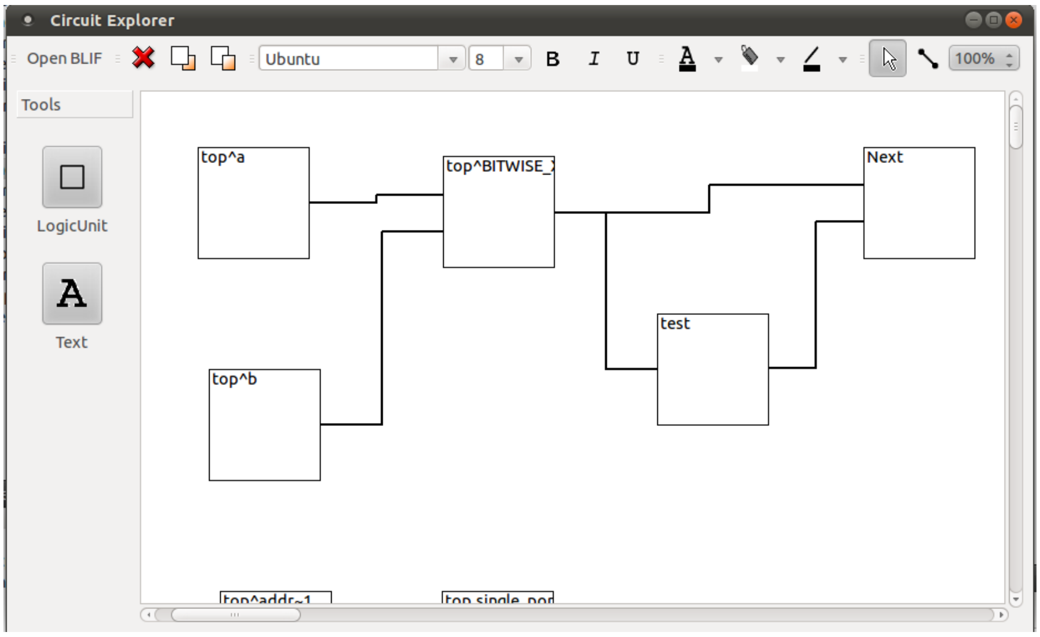

Figure 5.9: The Graphical User Interface Showing a Sample Boolean Circuit.

6 Evaluation

6.1 Use case based evaluation

This chapter covers the tests of the graphical user interface according to the use cases which were introduced during the planning phase in Chapter 4.2. All of them are evaluated regarding the implementation and how well they fit the requirements. The use case list also dictates the structure of this chapter as they are explained and evaluated in detail to show which parts are already sufficiently developed and where room for improvement still exists. At the end of this chapter a table with all results is given to provide an overview of the results. In Chapter 6.2 the usability of the GUI is verified with the help of potential users.

Open BLIF file per button, Open BLIF file in context menu

There are implementations for three different ways to open a file, which the user desires to explore. It is possible to click a button on the tool bar labelled *Open BLIF* as well as use the context menu in order to perform this action. The third possibility that is part of the context menu implementation is to use a hot key `Ctrl+O` at any time during the runtime of the application. There is no further input necessary in order to see the graph as the rest of the process happens fully automated. A possible improvement which is connected to this use case is to implement the possibility to open the file format, which is natively used in Odin II to be able to interact with Odin II and to provide more exploration options.

Rearrange menus as needed

The Qt library creates all menus and tool bars that is used during implementation. The buttons are grouped according to their functionalities in five groups:

- Open file tools
- Edit tools

- Font tools
- Colour tools
- Mouse tools

Each group can be dragged from the initial position and moved to another location. It is possible to dock the tool bars on each side of the application. There is a variety of possibilities to order and place the tool bars, even having them docked off the application and outside of the application window is possible.

Mark visualized items, interconnections between modules and highlight modules or connections

Each module and interconnection can be marked using the mouse. There is a predefined count of colors for interconnections and logic blocks. If more colors are needed, they can be added to the context menu. A better solution if many colors are needed would be to implement a color interface, which allows to choose a color using the whole RGB spectrum.

Rearrange items using the mouse

All modules can be dragged and dropped using the mouse. The interconnections which are attached to the logic block are adjusted to the actual position during the movement process. This allows the user for example to rearrange the graph to see the interactions he is interested in easier.

Interconnections, text and logic blocks can be added and deleted

If needed, logic blocks, text and interconnections can be added to the explorer. Logic blocks and text items are represented on the left side tool bar. Logic units are named *MouseAdd* when added, but the name of every item can be changed using the right click menu. To add interconnections, the mouse tool tool bar on the top of the GUI provides a connection tool, which allows the user to paint a line in order to connect two logic blocks.

If the origin and the end of the line connect two different logic blocks the interconnection is added to the graph.

Items can be deleted from the graph

Items in the graph can be deleted at any time. If a logic block is deleted, also all incoming and outgoing connections are deleted. It is also possible to delete multiple objects. Those can be selected one by one using the `Ctrl` key or by selecting all using `Ctrl+A` and pressing the delete button. Class structure can be extended by new module types, shapes etc.

Receive error, if BLIF file is corrupt or not found

Per default, the file choosing dialogue only shows folders and files with the ending `*.blif`. To choose a file without this extension, the user has to use the extension drop down menu. If the file which was chosen could not be opened or no structures are found in the description file, a warning is displayed which informs the user and offers to open another file.

Final State machine of file parser does not have *dead ends*

At the current state of work the state machine has only two states:

1. searching for logic blocks
2. searching for interconnections

Both of those steps are straight forward and end if the end of the BLIF file is reached. This means that no loops are possible and as long as the file can be read correctly the process always finishes.

Items arranged without overlapping

The graphs are arranged according to their inputs/outputs. This way logic blocks outgoing connections are always on the right side and incoming on the left side of the block. The arrangement algorithm creates a mesh of modules

Node count	Node Time	Conn. Count	Conn. Time
885	0.01s	1137	0.02s
945	0.01	7709	0.12s
1195	0.03s	10204	0.17s
1565	0.02s	4864	0.08s
2379	0.03s	9270	0.1s
2893	0.04s	8329	0.14s
12563	0.14s	38469	1.78s
20610	0.24s	67588	6.8s
39387	0.47s	168179	20.14s
82779	1.14s	224862	148.83s

Table 1: Evaluation of File Processing and Visualization Times. Times are divided in time, which is required for creation of logic blocks and time which is needed for creation of connections. All evaluation circuits are part of the Odin II benchmark set[26]

using this strategy. The problem with this alignment is that the interconnections are passing modules on their way to the ending point. There is no collision detection of interconnections and modules in the current state of the application.

Reading in a description file has to be performed in a reasonable time

The opening times that the application needs is divided into time which is needed:

1. to find and create all logic block.
2. to create all interconnections.

The BLIF files which were used to measure the time are part of the evaluation set of Odin II[26]. The machine, which was used during the evaluation had an Inter Core i7 processor and 4GB of RAM.

Table 1 shows that most of the time, which the user has to wait is needed to create the interconnections. The reason for this behaviour is that all logic blocks have to be existent in order to create a connection. The hash table of `LogicUnit` references makes sure, that the speed of find the ending points is high.

A circuit with 82,779 nodes and 224,862 connections is being processed for 148.83 seconds until the user is able to see something. This is a long time, but considered the count of object it is still acceptable. A graph, which shows the growth of both processes can be seen in figures 6.1 and 6.2.

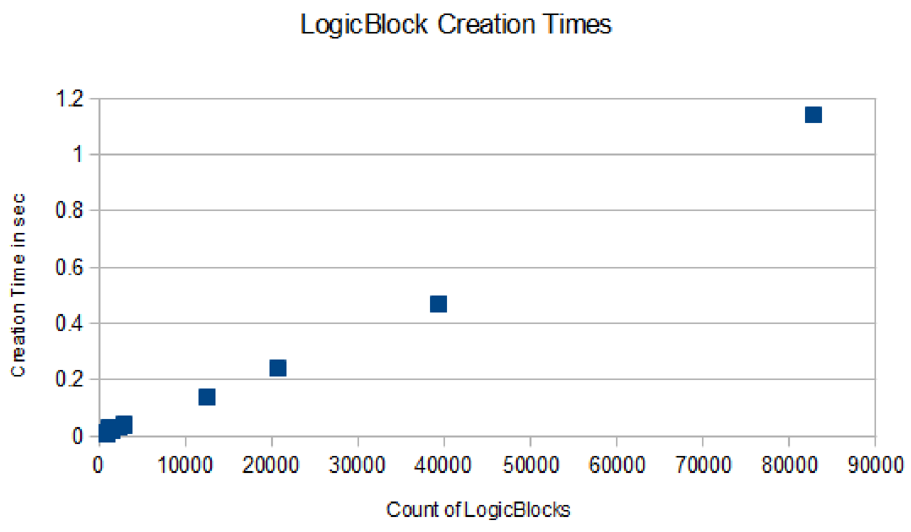

Figure 6.1: Creation times of Logic Blocks during file processing. Each dot shows the connection count and the creation count of one of ten test circuits.

6.2 Usability Evaluation

To verify the usability of the application, five computer science students have been asked to participate in a so called ”think aloud test”[27]. These students have never used the application before and were not involved in the development of this application. The goal of this evaluation is to make sure that all parts of the layout, all functionalities and the *look and feel* of the

Use Case	Implemented	Comments
Open BLIF file per button	Yes	
Open BLIF file in context menu	Yes	
Rearrange menus as needed	Yes	
Mark visualized modules	Yes	
Mark interconnections between modules	Yes	
Highlight modules or connections	Yes	Only single item at once
Rearrange modules using the mouse	Yes	Only single item at once
Zoom in/out the graph	Yes	
Close the application properly	Yes	
Interconnections can be added and deleted	Yes	
Text can be added in the graph	Yes	
Items can be added to the graph	Yes	
Items can be deleted from the graph	Yes	
A new graph can be opened if after another one was explored	Yes	
Class structure can be extended by new module types, shapes etc.	Yes	
Receive error, if BLIF file is corrupt or not found	Yes	
Recognize the modules correctly		
Final State machine of file parser does not have *dead ends*	Yes	
Recognize the interconnections correctly	Yes	
Modules arranged without overlapping	Yes	
Reading in a description file has to be performed in a reasonable time	Yes	

Table 2: Use Case Evaluation Summary

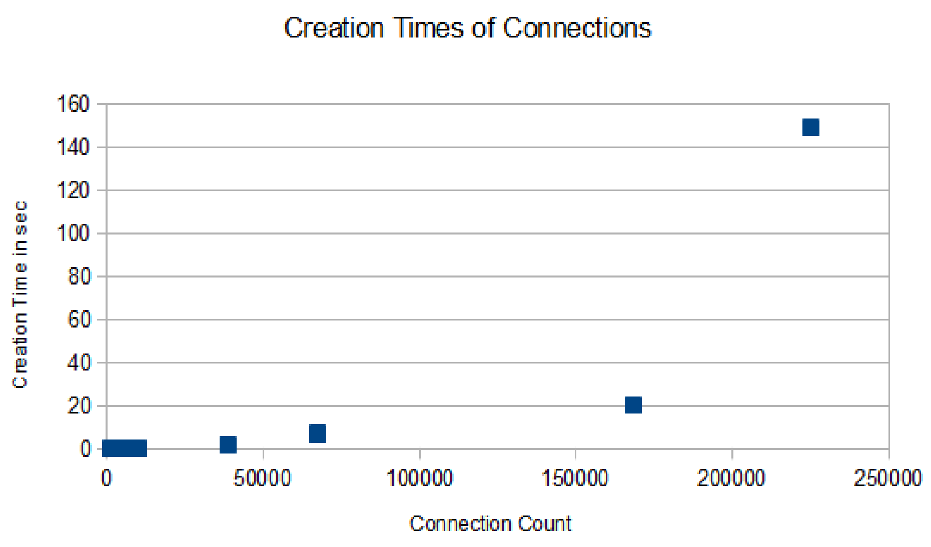

Figure 6.2: Creation times of Connections between Logic Blocks during file processing. Each dot shows the connection count and the creation count of one of ten test circuits.

GUI is intuitive enough to start using it without the need of any instruction.

The structure of the test requires every user to complete a predefined list of tasks. While the user is solving the tasks, he is advised to say everything he/she thinks. All thoughts which are connected to the application, it's functions etc. had to be communicated loudly. The list of tasks was the following:

1. Open application
2. Open specific BLIF file
3. Adjust zoom level to see the whole graph
4. Reorganize items as desired
5. Highlight one connection

6. Highlight the logic blocks which are connected by the highlighted connection
7. Open second BLIF file
8. Add a logic block manually
9. Rename the new logic block
10. Add the new logic block to the circuit by connecting it
11. Add a note to the new logic block saying it is a manually added node
12. Close the application

The result of the think aloud test shows that the main structure and layout of the application can be used very intuitively. The opening and exploring process could be performed by the users without bigger problems. Two out of four persons wished the zoom level could be adjusted not using predefined zoom levels but by typing in the desired percentage into a field. One person also tried to adjust the level using the combination of holding `Ctrl` and using the mouse wheel.

Item movement and handling was performed by all users without any missing points except for renaming a logic block. The renaming function can be found in the right click context menu or in the application menu in the *Item* section, where items also can be deleted and their layer can be adjusted. As no other task lead the user to these menus, two of them needed about 40 seconds to find it and one did not find the function and went on with the next task. Highlighting items was a task, which also was confusing in some cases as users tried to mark multiple logic blocks and to change the color. The application only changes the last of the items, which can be improved in future.

Another request by the users was to save the graph as an image or in any kind of format to be able to see the exploration results and the notes, which were added into the graph for later use. One of the users also asked if it

Task	**Completed**	**Avg Time in sec**
Open application	5	1
Open specific BLIF file	5	1.4
Adjust zoom level to see the whole graph	5	3
Reorganize items as desired	5	2
Highlight one Connection	5	5.2
Highlight the logic blocks, which are connected by the highlighted connection	5	10
Open second BLIF file	5	1
Add a logic block manually	5	5.8
Rename the new logic block	4	12
Add the new logic block to graph by connecting it	5	5.4
Add a note to the new logic block saying it is a manual added node	5	3
Close the application	5	1

Table 3: Results of the Think Aloud Tests

is possible to see the circuit simulate, which is not part of the application functionality at the moment. A summary of the 5 *think aloud tests* can be seen in Table 3.

Since the usability evaluation is completed the visualization software is used by developers in the Odin II project. According to their feedback the software was useful in order to improve the functionality of Odin II as the structure can be explored. The most useful part of the visualizer is the ability to see which nodes are connected. This allows to find missing connections, which were created, but got lost during the export process as well as for finding parts of the circuits, which are not used, as they represent subcircuits, which are not connected to the main circuit.

7 Conclusion

This report presented the development of a graphical user interface which is part of Odin II in order to support the exploration of netlist files, which describe the hardware architecture and the boolean circuit which was created for a FPGA device.

The report provided an introduction into the field of research in FPGA and development of GUIs as well as presented the tools of the workflow it is part of and their functionality. Improvement of the workflow as well as the development of the GUI, which is described in this report is still an ongoing process. The evaluation shows, that the basic structures and functionalities exist, but there is still work to do in order to provide a tool which can be used by developers for exploration and verification purposes.

The future development of the application will improve comfort functions of the applications as well as direct the missing points, which were found during the evaluation. A goal, which should become more importatnt is a closer linking of the visualization to Odin II. One of the advantages of it is the ability of Odin II to simulate a circuit. This simulater could interact with the visualization to provide input/output states, which could be visualized. That way also the functionality of a circuit could be explored and verified.

Different types of logic block in the circuit can be highlighted using different shapes to provide new functionalities. At the moment inputs are highlighted using a different shape than other logic blocks. This feature could be extended to highlight different kinds of logic blocks or hard blocks to structure visualization.

Unstructured BLIF files, which does not contain any placement and routing informations can be visualized in the moment. The main focus of this report was to visualize the BLIF files which are created of Odin II. Odin II is the first step in the workflow. In future also the outputs of the next stages in the workflow could be visualized to show the changing structure of a circuit. Also the visualization of structured BLIF files wich contain placement and routing information could be used to create a virtual representation of the whole FPGA device could be implemented.

Another yet missing function is a save operation, which creates a BLIF file of the actual graph which is shown by the application.

References

[1] D. Thomas and P. Moorby, *The Verilog hardware description language.* Springer Netherlands, 2002.

[2] P. Ashenden, *The designer's guide to VHDL.* Morgan Kaufmann, 2008.

[3] S. D. Brown, *Field programmable gate arrays.* Kluwer Academic, 1997.

[4] P. Chow, J. Rose, K. Chung, G. Paez-Monzon, and I. Rahardja, "The design of an SRAM-based field-programmable gate array. I. Architecture," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 7, pp. 191–197, June 1999.

[5] H. Schmit, "Extra-dimensional island-style FPGAs," *New Algorithms, Architectures and Applications for Reconfigurable Computing*, pp. 3–13, 2005.

[6] I. Kuon, R. Tessier, and J. Rose, "FPGA Architecture: Survey and Challenges," *Foundations and Trends in Electronic Design Automation*, vol. 2, no. 2, pp. 135–253, 2007.

[7] P. Leong, W. Luk, and S. Wilton, "Floating-Point FPGA: Architecture and Modeling," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 17, pp. 1709–1718, Dec. 2009.

[8] J. Babb, M. Frank, V. Lee, E. Waingold, R. Barua, M. Taylor, J. Kim, S. Devabhaktuni, and A. Agarwal, "The RAW Benchmark Suite : Computation Structures for General Purpose Computing," *Components*, p. 134, 1997.

[9] Y. Jin, N. Satish, K. Ravindran, and K. Keutzer, "An automated exploration framework for FPGA-based soft multiprocessor systems," *Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis - CODES+ISSS '05*, p. 273, 2005.

[10] Altera Corporation, "FPGA, CPLD and ASIC from Altera." `http://www.altera.com/`, Aug. 2011.

[11] I. Xilinx, "FPGA, CPLD, and EPP Solutions from Xilinx, Inc." `http://www.xilinx.com/`, Aug. 2011.

[12] S. Williams, "Icarus Verilog." `http://iverilog.icarus.com/`, Aug. 2011.

[13] I. Geeknet, "VeriWell Verilog Simulator." `http://sourceforge.net/projects/veriwell/`, Aug. 2011.

[14] F. P. Group, "The FreeHDL Project." `http://freehdl.seul.org/`, Aug. 2011.

[15] P. Jamieson, K. B. Kent, F. Gharibian, and L. Shannon, "Odin II - An Open-Source Verilog HDL Synthesis Tool for CAD Research," *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 149–156, 2010.

[16] M. Summerfield, "Rapid GUI programming with Python and Qt: the definitive guide to PyQt programming," *October*, 2008.

[17] J. Blanchette and M. Summerfield, *C++ GUI programming with Qt 4*. Prentice Hall PTR, 2006.

[18] T. Sommer, "Physics for a 3D Driving Simulator Torsten Sommer Bachelor Thesis," *Physics.*

[19] A. Sharma, "White Paper Merits of QT for developing Imaging Applications UI," *System*, pp. 1–8, 2008.

[20] Nokia Corporation, "Qt Reference Documentation." `http://doc.qt.nokia.com/4.7-snapshot/index.html`, Aug. 2011.

[21] A. Mishchenko, "ABC: A System for Sequential Synthesis and Verification." `http://www.eecs.berkeley.edu/~alanmi/abc/`, 2011.

[22] P. Jamieson and J. Rose, "A verilog RTL synthesis tool for heterogeneous FPGAs," in *Field Programmable Logic and Applications, 2005. International Conference on*, pp. 305–310, IEEE, 2005.

[23] V. Betz and J. Rose, “VPR : A New Packing , Placement and Routing Tool for,” *Technology*, pp. 1–10, 1997.

[24] J. Luu, I. Kuon, P. Jamieson, T. Campbell, A. Ye, and W. M. Fang, “VPR 5 . 0 : FPGA CAD and Architecture Exploration Tools with Single-Driver Routing , Heterogeneity and Process Scaling,” *Computer Engineering*, pp. 133–142, 2009.

[25] U. Berkeley, “Berkeley logic interchange format,” tech. rep., Technical report, Technical report, University of California at Berkeley, Aug. 1998.

[26] J. Luu, O. Densmore, R. Rubin, C. W. Yu, K. B. Kent, P. Jamieson, A. J., and R. J., “The VTR Project: Architecture and CAD for FPGAs from Verilog to Routing,” *Accepted for ACM International Symposium on Field Programmable Gate Arrays (FPGA) 2012, Monterey, USA, February 21-23*, 2012.

[27] D. M. Turner-Bowker, R. N. Saris-Baglama, K. J. Smith, M. a. DeRosa, C. a. Paulsen, and S. J. Hogue, “Heuristic evaluation of user interfaces,” *Telemedicine journal and e-health : the official journal of the American Telemedicine Association*, vol. 17, no. 1, pp. 40–5, 1990.

[28] A. Mishchenko, S. Chatterjee, and R. K. Brayton, “Improvements to Technology Mapping for LUT-Based FPGAs,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, pp. 240–253, Feb. 2007.

A Appendix

B Comparison between BLIF file and Visualization

This example shows a Circuit with 10 nodes and 25 connections. It is supposed to give an idea how hard it can be to understand the structure of a BLIF file even if a limited amount of nodes is described. BLIF files in the Odin II evaluation set have up to 451,788 nodes.

```
.model testcase5
.inputs top^a top^b

.names top^a top^b top^BITWISE_XOR~0^LOGICAL_XOR~8
00 1
11 0
01 1
10 0

.names top^a top^b top^BITWISE_XOR~0^LOGICAL_XOR~8 top^AND_OR_5
-0- 1
0-- 0
0-1 1

.names top^BITWISE_XOR~0^LOGICAL_XOR~8 top^AND_OR_5 top^XOR_7
-1 1

.names top^BITWISE_XOR~0^LOGICAL_XOR~8 top^AND_OR_5 top^XOR_7 top^OR_3
0-1 0
01- 1

.names top^a top^AND_OR_5 top^XOR_7 top^OR_3 top^RES_0
000 0
001 1
010 1
011 0
101 0
110 0
111 1

```

```
.names top^b top^AND_OR_5 top^XOR_7 top^OR_3 top^RES_1
1-1- 0
-1-1 1

.names top^a top^b top^RES_1 top^BITWISE_XOR~0^LOGICAL_XOR~8 top^RES_2
0-1- 0
-1-0 1

.names top^RES_0 top^RES_1 top^RES_2 top^RES_3
1-1 0
-1- 1
.end
```

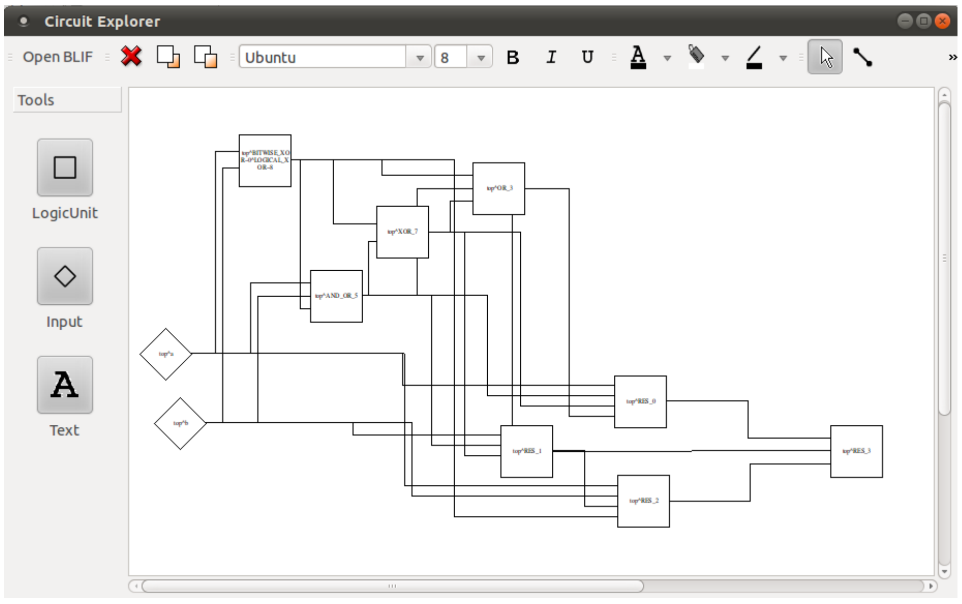

Figure B.1: Test case circuit with 10 nodes and 25 connections