CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Technische Universität Dresden: Qucosa

Großer Beleg

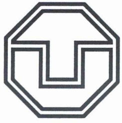
Graphical Support for the Design and Evaluation of Configurable Logic Blocks

Fredo Erxleben
Born 30th April 1987 in Blankenburg (Harz)
Matriculation number 33 00 664

29th April 2015

Technische Universität Dresden
Fakultät Informatik
Institut für Technische Informatik

Supervising Professor: Prof. Dr.-Ing. habil. Rainer G. Spallek
Supervisor: Dr.-Ing. Thomas B. Preußer



TECHNISCHE
UNIVERSITÄT
DRESDEN

Fakultät Informatik Institut für Technische Informatik, Professur für VLSI-Entwurfssysteme, Diagnostik und Architektur

Aufgabenstellung für den Großen Beleg

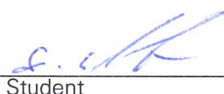
Student:	Fredo Erxleben		
Studiengang:	Informatik	Beginn:	03.11.2014
Matrikelnummer:	33 00 664	Abgabe:	02.05.2015

Thema: **Graphical Support for the Design and Evaluation of Configurable Logic Blocks**

Configurable hardware is a key technology to implement custom digital circuitry without having to cope with the immense costs of producing custom silicon on recent technology nodes. It is essential for making non-high-volume and flexible circuitry economically feasible.
Configurable logic blocks (CLBs) contribute heavily to the flexibility of configurable hardware. They form a repetitive structure that adopts a custom circuit by programming connection points, data-path multiplexers or truth tables.
The design of CLBs is a critical engineering challenge, which must trade off the occupied silicon area against the flexibility and computational strength provided by it. While the manufacturers of programmable gate arrays have to deal with this task naturally, the goal of this project is to make the basic exploration of CLB designs easily accessible for a broad audience by initiating an intuitive graphical design and evaluation tool. Essential features that should be implemented within the scope of this project are the graphical input of a combinational CLB circuit and the integration of an available SAT-based boolean matching flow for strength evaluation. Besides a statistical strength evaluation across a corpus of boolean functions, also the visualization of specific CLB configurations for concrete implementable functions shall be enabled by a suitable back annotation. The integration into an established tool like `qucs` is desirable.

1. Literature review on CLB evaluation and boolean matching.
2. Selection of a set of suitable tools for the implementation and integration of the envisioned solution.
3. Implementation of the CLB design and evaluation inside the chosen ecosystem of tools.
4. Demonstration and evaluation of the achieved functionality also identifying future enhancements.
5. Written documentation of the conducted research, the implemented design and its evaluation.

Betreuer:	Dr.-Ing. Thomas B. Preußer
Betreuender Hochschullehrer:	Prof. Dr.-Ing. habil. Rainer G. Spallek


Student

Betreuender Hochschullehrer

Verteiler: HSL, Betreuer, Student

Declaration of Academic Honesty

Hereby I declare, that I created this document at hand by myself, using only the mentioned sources and auxiliary means. All quotations are marked as such.

Dresden, 29th April 2015

Fredo Erxleben

Trademarks

All trademarks remain property of their respective owners. Unless otherwise specified, no association between the author and any trademark holder is expressed or implied. No endorsement of the author is expressed or implied by the mention of such trademarks.

Abstract

Abstract Developing a tool supporting humans to design and evaluate configurable logic block (CLB)-based circuits requires a lot of know-how and research from different fields of computer science. In this work, the newly developed application *q2d*, especially its design and implementation will be introduced as a possible tool for approaching CLB circuit development with graphical user interface (UI) support. Design decisions and implementation will be discussed and a workflow example will be given.

Thanks to...

... *Dr. Thomas Peußer*, for letting me occupy a chair in his office, providing a *C/C++* interface for Quantor using his own parser *wisent* and being a great supervisor.

... *Dipl.-Inf. Marcus Hähnel*, who offered a lot of advice regarding proper *C++* and Qt, did code reviews, beta tests and improved the applications build script to be more user friendly.

... *(Soon to be Dipl.-Minf.) Paula Schöley*, for lengthy discussions about UIs, human-machine interaction as well as human perception of shapes and colors.

... *my family and friends*, helping me to keep my feet on the ground

... *Krümelchen*, helping me to keep my head in the clouds.

Contents

1 Introduction **1**

1.1 Forethoughts . . . 1
1.2 Theoretical Background . . . 2
1.2.1 Definitions . . . 2
1.2.2 Expressing Connections between Circuit Elements . . . 3
1.2.3 Global Context and Target Function . . . 3
1.2.4 Problem formulation as QBF and SAT . . . 4

2 Description of the Implemented Tool **7**

2.1 Design Decisions . . . 7
2.1.1 Choice of Language, Libraries and Frameworks . . . 7
2.1.2 Solving the QBF Problem . . . 8
2.1.3 Design of the Internally Used Meta-Model . . . 9
2.1.4 User Interface Ergonomics . . . 9
2.1.5 Aspects of Schematic Visualization . . . 10
2.1.6 Limitations . . . 12
2.2 Implemented Features . . . 13
2.2.1 Basic Interaction . . . 13
2.2.2 User-Defined Components . . . 13
2.2.3 Generation of Circuit Symbols . . . 14
2.2.4 Methods for Specifying Functional Behaviour . . . 15

3 Implementation Details **19**

3.1 Classes Involved in the Component Meta-Model . . . 19
3.2 The Document Entry Class and its Factory . . . 21
3.3 Model and View . . . 22
3.3.1 The Model Element Hierarchy . . . 22
3.3.2 The Schematics Element Hierarchy . . . 23
3.4 The Quantor Interface . . . 25

4 An Example Workflow **27**

4.1 The Task . . . 27
4.2 A Component Descriptor for Xilinx' LUT6-2 . . . 27
4.3 Designing the Model . . . 30
4.4 Computing the Desired Configuration . . . 30

5 Summary and Outlook **33**
5.1 Achieved Results 33
5.2 Suggested Improvements 34
References **35**
A Acronyms and Glossary **A – 1**
B UML Diagrams **B – 1**

List of Figures

2.1 Port and Signal Flow Visualization 12
2.2 An automatically generated component symbol 15

4.1 Circuit Diagram of a CLB Detail 28
4.2 Circuit Diagram of Xilinx' LUT6-2 28

B.1 unified markup language (UML)-Diagram for the Component Meta-Model . . . . . B – 2
B.2 UML-Diagram for the Document Entry and its Factory . . . . . B – 3
B.3 UML-Diagram for the *q2d*-Model . . . . . B – 4
B.4 UML-Diagram for the *q2d*-Schematic visualization . . . . . B – 5
B.5 UML-Diagram for the *q2d*-Quantor Interface . . . . . B – 6

List of Tables

2.1 Overview of Logical Operator Notations in *q2d* 16

List of Listings

2.1 A minimal component descriptor file 14
2.2 A descriptor file with a custom circuit symbol 16

4.1 The Component Descriptor for Xilinx' LUT6_2 29

1 Introduction

1.1 Forethoughts

Problem Statement The aim of this work is the creation of a tool that can support hardware designers in evaluating the capabilities of CLBs. To achieve this, it is necessary to find ways of easily modelling configurable components using a graphical UI. It is further required to offer a method to specify the intended behaviour of the whole design, at which point the tool should evaluate whether this specification can be implemented by the CLB, and by which configuration.

Outline of the Solution Initial research made clear that so far no tool exists for supporting this kind of CLB evaluation in a user friendly way. Attempts to extend already existing projects also proved to be difficult. As a consequence, a new application called *q2d* was designed and implemented for the outlined purpose. It supports project management, CLB-based circuit design and the evaluation of such. The user is presented with an appealing graphical UI and can create custom component types quickly, requiring only simple means.

Involved Areas of Computer Science In addition to the area of design and engineering of large-scale integrated circuits, several other fields of computer science are involved in creating an application capable of dealing with the posed task. As a result, the work at hand might also be worth reading for audiences engaged in these particular areas:

Computational Logic is involved when discussing the theoretical background of the problem and solving the resulting quantified boolean formula (QBF) problem in section 1.2.

UI Engineering will be discussed largely in sections 2.1.4, 2.1.5 and 2.2.1 where the presentation of information and user interaction will be discussed,

Software Engineering comes in mostly during the design phase, when the choice of tools and the internal program structure is explained in sections 2.1.1, 2.1.3 and its implementation in chapter 3.

Application Use Cases The resulting application will operate on the register transfer level. It is designed to help the user focus on the design task he wishes to perform without distracting him with details regarding the concrete physical implementation of specific circuit elements. Therefore, *q2d* is best suited to develop, test and explain concepts of configurable circuit designs and demonstrate their flexibility and limits.

1.2 Theoretical Background

When looking at the posed problem from a mathematical perspective, it becomes clear that the solving of a QBF problem is required. Intuitively speaking, the designed circuit and its components are to be described by boolean formulae. For those, an interpretation is to be found, which lets all formulae become true. As such a satisfying interpretation defines suitable values for the configurable circuit elements, it will be called a *configuration*.

1.2.1 Definitions

For simplicity, a shorthand notation for the set of boolean values will be used as provided in equation 1.1.

$$\mathbb{B} := \{\bot, \top\} \tag{1.1}$$

When talking about circuits, the need to distinguish its elements from each other arises, since each of them establishes its own *context*. This will be achieved by marking symbols referring to circuit element ε with the index ε. If no index is given, the symbol refers to the circuit as a whole. This will also be referred to as the *global context*.

Further, it is useful to define three disjunct sets of variables. Each of these sets has a distinct semantic meaning.

C_ε Configuration variables represent programmable bits that define the temporary specific behaviour of an element.

N_ε Node variables represent the interfacing points of components that serve as ports for interconnecting the overall circuit.

X Input variables stand for the outside connections of the whole circuit that are driven arbitrarily during the circuit's operation by the surrounding world.

Let a partial Component Behaviour Descriptor (pCBD) be given as a boolean formula on a set of configuration and node variables as in equation 1.2.

$$f_{\varepsilon,i}(V_\varepsilon) \left| \begin{array}{l} V_\varepsilon \subseteq (C_\varepsilon \cup N_\varepsilon) \\ i \in [0,n)\,;\, n \in \mathbb{N} \end{array} \right. \tag{1.2}$$

An arbitrary large set of pCBDs can be used to describe the behaviour of a single circuit element such as a component or a wiring. All those pCBDs are combined in equation 1.3 into a set describing the whole circuit elements behaviour, called the Component Behaviour Descriptor (CBD).

$$F_\varepsilon := \bigcup_n f_{\varepsilon,n}(V_\varepsilon) \tag{1.3}$$

For any given element ε, its *local context* is defined in equation 1.4 as

$$\square_\varepsilon := (C_\varepsilon, N_\varepsilon, F_\varepsilon) \tag{1.4}$$

1.2.2 Expressing Connections between Circuit Elements

In the case of node variables, it may occur that two elements effectively refer to the same node.[1] Then, given the existence of two elements ε_a and ε_b sharing the same node, this common node can be expressed as a boolean formula $\sigma_{a,b}$.

$$\sigma_{a,b} = \{n_a \Leftrightarrow n_b | n_a \in N_{\varepsilon_a} \wedge n_b \in N_{\varepsilon_b} \wedge n_a \text{ and } n_b \text{ are connected}\} \tag{1.5}$$

Note that the equivalence can be transformed as shown in equation 1.6.

$$\begin{aligned} x \Leftrightarrow y &\equiv (x \Leftarrow y) \wedge (x \Rightarrow y) \\ &\equiv (\neg x \vee y) \wedge (x \vee \neg y) \\ &\equiv [\bar{x}, y]\,[x, \bar{y}] \end{aligned} \tag{1.6}$$

This will come in handy at several points in the future, especially when discussing conversions into clause normal form (CNF).

1.2.3 Global Context and Target Function

Creating the Global Context To obtain a holistic description of the circuit, it is required to create a global context for the whole described circuit. For this purpose, the sets of node and configuration variables are merged as shown in equation 1.7.

$$\begin{aligned} N &:= \bigcup_{\varepsilon} N_{\varepsilon} \\ C &:= \bigcup_{\varepsilon} C_{\varepsilon} \end{aligned} \tag{1.7}$$

In the global context, there also exist input variables that have to be mapped to node variables to express how the inputs are connected to the circuit. This mapping is quite similar to the definition of the σ-notation in equation 1.5 and consequently is called σ^*.

$$\sigma^* = \{x \Leftrightarrow n | x \in X \wedge n \in N \wedge x \text{ and } n \text{ are connected}\} \tag{1.8}$$

One might wish to ensure that all inputs to a circuit are connected by requiring

$$\forall x. \exists n. \, (x \Leftrightarrow n) \in \sigma^* \, (x, n) \left| \begin{array}{l} x \in X \\ n \in N \end{array} \right. \tag{1.9}$$

This, however is not necessary for the problem to be decidable and will not be enforced. As a result, a global set of formulae set F can be created as

$$F := \bigcup_{\varepsilon} F_{\varepsilon} \cup \bigcup_{a,b} \sigma_{a,b} \cup \sigma^* \tag{1.10}$$

[1] For example, one might think of a wire that is connected to the port of a component.

This can also be merged into a single formula F^* by conjugating all elements of F.[2]

$$F^* := \bigwedge_{f \in F} f \tag{1.11}$$

Comparable boolean description techniques of circuit behaviour can be found in [Saf+06] and [Lin+07].

Conclusively, the global context can now be represented by

$$\square := (C, N, X, F) \tag{1.12}$$

Including the Target Formula Once all contexts are established, the last thing left to specify is the *target formula*. It describes desired behaviour of the given circuit partially.[3] For that purpose, an indexed target formula t_i can be defined as

$$t_i(V) \left| \begin{array}{l} V \subseteq (X \cup N) \\ i \in [0, n)\,;\, n \in \mathbb{N} \end{array} \right. \tag{1.13}$$

To maintain a consistency with the ways of defining and naming things, the set of partial target functions is called T.

$$T := \bigcup_i t_i \,|\, i \in \mathbb{N} \tag{1.14}$$

Since all target functions need to be fulfilled to solve the problem, it is an obvious step to conjunct them into a general target function T^*.

$$T^* := \bigwedge_{t \in T} t \tag{1.15}$$

1.2.4 Problem formulation as QBF and SAT

With all prerequisites established, equation 1.16 then encodes the fact that there exists a configuration that for all input combinations, there is a node assignment so that $F^* \wedge T^*$ is fulfilled.

$$\exists c . \forall x . \exists n . \left(F^* \wedge T^*\right) \Big| \; c \in C;\; x \in X;\; n \in N \tag{1.16}$$

This formula establishes an *EAE-problem*, which is a special instance of a QBF. Many tools require a representation in CNF. It is possible to translate QBF-problems into satisfiability (SAT)-problems although the sets of clauses and variables will expand largely. [SaBa05]

Section 2.1.2 will discuss how this problem has been approached in the actual implementation.

[2] One may find that node variables could be removed at this point by merging functions into each other and creating one large transfer formula that describes the whole circuit. There are, however, no practical gains proceeding so since the automated solving of the problem might re-introduce them eventually. Also, solvers are capable of performing clause and literal eliminations by themselves.

[3] Given the case that there exists only one partial target formula it trivially equals the complete target formula.

Looking for Multiple Solutions Once the problem formulation is present in CNF, it can be evaluated, whether the problem is satisfiable. Should this be the case, the negated solution can be added as a clause to the problem. If this also turns out to be satisfiable, an alternative solution has been found. This process can be repeated until the problem is no longer satisfiable, at which point all possible solutions have been found.[4]

[4] In the actual implementation, the SAT solver might abort not only due to the problem being not satisfiable but also when running out of memory or exceeding a certain computation time. Usually solvers report the reason for stopping early. Still, only non-satisfiability can definitively negate the existence of further solutions.

2 Description of the Implemented Tool

2.1 Design Decisions

Before starting the actual implementation of the desired tool, it was necessary to make some fundamental design decisions regarding the work at hand such as

- required features,
- used code base,
- included external libraries and tools,
- applied programming and description languages,
- ways of directing the users workflow, and
- increasing the applications usability by appropriate visualization techniques.

Some of the choices made had to be revised during the implementation process and additional design cornerstones were added when required to achieve certain goals.

2.1.1 Choice of Language, Libraries and Frameworks

Research of Integration Possibilities into Existing Tools The initial idea was to integrate the use-case proposed by the problem statement into an existing tool. Qucs[BrJa08] seemed to be an obvious choice for that purpose due to its already existing visual design capabilities, availability of source code and sufficiently large community of users and contributors. When researching ways to integrate the requested features however, it was found that the code quality was low and documentation was largely missing. The developers were aware of these shortcomings, and at the time of writing, Qucs is subject to an extensive refactoring process.

Some attempts were made to extend the code base, allowing the extraction of the internal model in a netlist format for further processing by other tools. The components used in the schematic turned out to be hard-coded into the tool. Due to the points mentioned above, it quickly turned out to be a time-consuming, difficult and error-prone effort to achieve even minor results. This situation led to the decision not to expand Qucs as proposed by the initial task. Research for other tools that could be used for integrating the desired functionality did not reveal any alternatives that were deemed viable. Developing a new tool for the intended use case therefore was regarded as the best way to proceed.

Used Languages and Frameworks The experiences made earlier when investigating Qucs yielded the insight that it would be a good choice to use a combination of a well-established high-level

language in combination with a framework that could provide powerful tools and abstractions to reduce the effort needed to implement the UI.[1]

A suitable combination was found in using the *C++* programming language together with the Qt-framework. While the *C++14* standard was not yet commonly supported by development tools at the time of writing, *C++11* had been well established and offered features that were found desirable for the task at hand. Due to the large set of features offered by Qt, it turned out not to be necessary to include further frameworks to achieve the intended results. As a consequence, the major part of the UI is described using the extended markup language (XML).

It was aimed at using a very simple, human-readable format to allow the user easy access to the stored data for modification and debugging purposes. Therefore, JavaScript object notation (JSON) has been employed to represent user-defined components and data persistence. Since it is easy to parse by a program[2] and can also be fairly well edited by a human user with a simple text editor, this format proved to be a good choice during implementation. This will also become an important aspect in section 2.2.1 and 2.2.2.

2.1.2 Solving the QBF Problem

In section 1.2, it was established that solving QBF is one way to compute a configuration in order to be able to evaluate whether a given target function can be implemented by a provided circuit design. Applying a dedicated QBF solver instead of transforming the problem into SAT first has some clear advantages:

- It eliminates the need for intermediate representations and expansion logic between QBF and SAT representations.[3]
- Established QBF solvers are very well optimized and, therefore, perform much better than any implementation of non-specialized programmers can be reasonably expected to do.

An extensive comparison of currently available QBF solvers was composed by NARIZZANO ET AL. [Nar+06].

From this, Quantor was selected as backing tool for this task, providing the capabilities mentioned above.

In contrast to most other tools, it not only decides whether a problem is satisfiable, but also yields a variable assignment for the outermost existential quantor, if there is any. It thus computes the desired configurations.

Quantor is provided as source code and can be easily included either by directly compiling it into the application by or generating a linkable library.

Interfacing Quantor The input to Quantor has to be provided in CNF. So it is necessary to create an interface that transforms the formulae specifying the internal model and target formulae in an appropriate way for the QBF solver to use and interpret the returned results to obtain a conclusive solution that can

[1] Experience shows that UI implementations, especially for graphical ones, consist of huge portions of configuration code that can be generated automatically. It is a noticeable relief for the developer to do so. [Sch+10] [Eng+02]

[2] Qt already offers means of writing and parsing JSON, which simplifies the implementation considerably.

[3] A QBF problem's size will grow exponentially, when transformed into SAT as noted in section 1.2.4.

be presented to the user. For doing so, the computed result must be interpreted and prepared for visual representation. The concrete implementation of the classes responsible for this step can found in section 3.4.

2.1.3 Design of the Internally Used Meta-Model

While researching how Qucs handled components, it was discovered that the wide variety of available building blocks used for circuit design were directly included as classes in the source code. This might be a sound decision for the tool in question but it also was evident that thereby the maintainability and introduction of new components was notably inhibited. From the experiences made arose the desire to use a different approach in *q2d*.

A *component meta-model* has been designed to represent and categorize a hierarchy of templates, from which the actual building blocks for the actual model can be derived.

Abstraction into Component Types When designing schematics, it becomes obvious very quickly that most components do have a lot of things in common and share structural and behavioural similarities. Commonly used hardware design languages (HDLs) handle this by introducing types or type-like semantics of which a component can be. This approach was adopted in *q2d* by introducing a meta-model which describes a hierarchy of component categories and component types. From the latter, concrete component instances can be created. Categories, on the other hand, are intended mainly for helping with keeping an overview, even if a lot of component types are present. In the following the term *component library* will be used as a generalization of any category containing component types. Further, component libraries are not restrained from containing sub-libraries, unless explicitly stated otherwise in the context.

Allowing User-Defined Component Types No matter how many pre-defined component types are delivered with the application, users will most likely find that some component they need for their design is missing. Since most hardware producers maintain their own catalogues of components, which are updated frequently, it is not feasible to keep the application up-to-date on a regular basis. Especially, if components were hard-coded into the application, any changes to the component libraries would require a rebuild of the whole application and a new release. As a result, it was deemed far more practical if users were allowed and enabled to create their own components and component libraries. To do so, each component type is specified by a *component descriptor file*, which itself is only a simple JSON file following a defined structure. This will be discussed in depth in section 2.2.2.

2.1.4 User Interface Ergonomics

The way a user interface is designed has a strong influence on how the user approaches the task they wish to accomplish. As a consequence, the UI should, amongst other things:

- Offer sufficient tools to solve the given task while avoiding to become overburdened with features that have little to no use or can be reasonably achieved otherwise. The latter often increases the complexity of a program noticeably while offering little value in return.

- Follow semantics that the user can anticipate and comprehend. Preferably, these do require little to no explanation and feel intuitive.
- Attempt to discourage creations by the user that might be regarded as flawed or undesirable.

In the following, examples for design decisions regarding these points are given.[4]

Disallow Connecting Output Ports to Each Other When designing circuits, it is technically possible to connect multiple output ports of one or several components to the same input port, as shown in figure 2.1a. This design method is often used to increase the driving strength of output signals. This practise tends to lead to problems in the resulting circuit, should the output ports assume different signal levels at the same time. In the worst case, the physical implementation might be damaged. For this reason, it was decided to disallow the driving of an input port by multiple sources and to have a connecting element to be connected to at most one output port.[5] This is achieved by only allowing to start drawing a wire via dragging forth from an output port and only end this procedure by either dropping over an input port to create a connection or by dropping it on a blank area, thus cancelling the process.

Draw Connections Only in Signal Flow Direction As a consequence of the restrictions mentioned above, the user is forced to always draw connections between components according to the signal flow direction. It was theorized, while testing the application, that this restriction might force the user to plan his designs ahead in more detail and reduce the likelihood of design errors. A separate study would be required to measure these effects though.

Disallow Mixed-Direction Ports In the early stages of the application design, it was considered to not only support *input* and *output* ports in components but also a variant that could change its direction, similar to *inout* ports known from most HDLs. This would contradict the decisions made before and noticeably complicate the implementation, especially with respect to the interface of the QBF solver. Since those mixed-direction ports can be safely emulated by using a separate port for each direction, it was judged to explicitly support only single-direction ports. Also, it has to be noted, that it is highly unusual for CLBs to be produced with mixed-direction ports.

2.1.5 Aspects of Schematic Visualization

Humans are very focused on optical perception of information, which naturally puts an emphasis on the way things are represented visually and can be interacted with. Implementing appropriate means of aiding the user perceiving important information faster turned out to be very work-intensive. While a huge amount of visualization techniques exists, only a subset could be considered for the actual implementation[Bern10].

[4] It would exceed the extend of this document to elaborate all of the design decisions that were made before and during the implementation in detail.

[5] An alternative approach would be to introduce formal verification to ensure all drivers always carry the same signals at the same time. Any solution output by Quantor would ensure the outputs carry the same signals in a static state. However, timing behaviour has to be considered, which would make this approach become very complex quickly and create the need for a lot of very specific details like timing behaviour of the actual components used in the physical implementation. Any formal verification could only be made for a specific setup and had to be re-evaluated whenever the circuit design changes.

Aiding Orientation on Larger Schematics As soon as a schematic contains more than a handful of components and connections, it becomes notably harder to maintain the overview and the available screen space might no longer suffice to display everything at once. The most common solution for this problem is to add scrolling capabilities to the view area. Since the scrolling elements on their own require space on the view as well, it was concluded that a *scrolling on demand* approach posed a good compromise. To further aid the user's orientation on the view area and helping with the alignment of the schematics elements, a dot grid was added. It was found to be a better alternative to a line-based grid, which was perceived as more distracting and creating the illusion of a slightly darkened background.

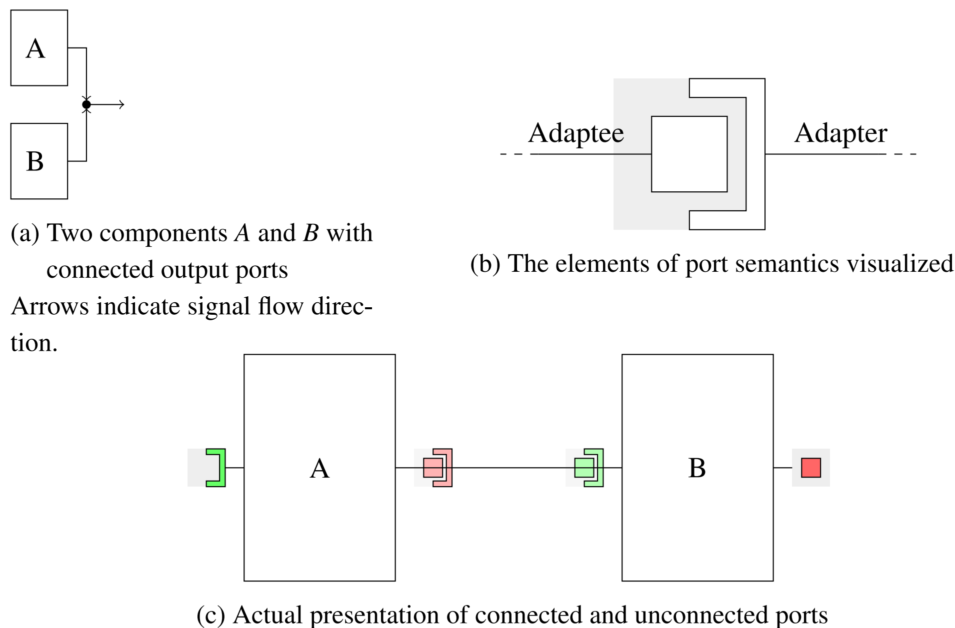
Providing Details on Demand Attempting to provide all available information at once tends to clutter visual representations rather quickly. A set of so called *details on demand*-techniques is available to counter information overload. In the application at hand, tool tips are employed to obtain information about certain elements. If this is not sufficient to display all available details, double-clicking an element will yield all information in a separate window. Furthermore, connections will highlight when hovered over so that it is easier for the user to see which elements they are linking.

Visual Representation of Ports Ports are associated with information regarding their signal flow direction, the elements, between which they are interfacing, and their connection status. In the first versions of the application, ports were displayed as circles with a filling color depending on their signal flow direction with respect to their owning component. However, this led to confusion as they tended to be mistaken for negations. While in hand-drawn schematics, simple forms dominate since they can be produced quickly and easily, on computer-generated views, it is feasible to employ more complex shapes, which have the potential to change depending on their state and thus can convey more information. It was therefore decided to enhance the representation of ports following a semantic of *adapters and adaptees*.

An *adapter* is a port that receives a connection, which is usually the case for input ports. *Adaptees*, on the other hand, provide a signal, which can be received by its counterpart. Both elements are displayed in figure 2.1b. Additionally, when connecting ports, the endpoints of the connection will complement the existing port symbols with their counterparts to create the visual impression of a plug connection. These semantics go well with the restrictions of connection drawing discussed in section 2.1.4. Regarding the basic shape of these elements, circular forms, looking similar to the *lollipop*-shapes known from UML were discussed and rejected as still looking to similar to negations. For the same reason triangular versions were discarded[6], and shapes based on squares were chosen. The color scheme devised in earlier versions has been kept, assigning a green color to input ports and using red for output ports. A slightly grey *active region* surrounds the port symbols to give the user a hint that special interaction options, like drawing connections, are available. Symbols of ports that are already connected are set to be more transparent, thus reducing the amount of dominant visual elements and helping the user find ports that are still unconnected. The actual representation as implemented is shown in figure 2.1c.

Module Interfaces To aid the understanding of the following sections, the concept of a *module interface* will be introduced briefly. A whole circuit, as contained in a single document is synonymously called a *module*. Special elements that act as an interfacing point between the module and the outside world are consequently named *module interfaces*. If such a module interface conveys data from the

[6] They looked too close to the DIN-symbols for ports with polarity indicators for negated logic, as well as clock driven ports.



(a) Two components *A* and *B* with connected output ports
Arrows indicate signal flow direction.

(b) The elements of port semantics visualized

(c) Actual presentation of connected and unconnected ports

Figure 2.1: Port and Signal Flow Visualization

outside world into the circuit, it may be considered an *input*. When viewed from the inside of the module however, an input appears to emit data. *Outputs* behave exactly the other way around. This matter is solved by implementing module input and output interfaces conceptually comparable to components, with an actual module port as child element. When talking about ports, usually the context makes clear, which kind of port is referred to or it will be explicitly stated.

2.1.6 Limitations

With a wide range of techniques for visualization available and many potential uses for the models that result from the visual design of circuits, it is also important to decide upon limitations of the applications implementation.

Usability Restrictions Features like zooming, rotation of schematic elements, cursor snapping to the dot grid and active-edge scrolling, were considered, but rejected as their estimated implementation effort did not meet with the projected gains in usability, especially with respect to the time limit imposed by the assignment.[7]

Functional Constraints Even though a feasible extension of the model at hand, the implementation of state was not pursued. As a result, it is not possible to simulate any behaviour, and the use of state-bearing components like *flipflops* is not allowed.

[7] The author is also aware that the tool often implies in the visualization that the data flow is oriented from left to right, which is not agnostic to the users culture. Later versions of the tool are intended to drop this restriction.

2.2 Implemented Features

2.2.1 Basic Interaction

In the following, a short overview over the basic functions of the application shall be given.

Project and Document management All works created by the user are organized on two hierarchy levels:

Documents are representations of exactly one design each. Their most notable trait is the association with a schematic visualization, which can be edited by the user. Documents do not exist on their own but rely on the existence of a project to be contained in. Any circuit described by a document carries the documents name and is referred to as a *module*.

Projects form a collection of documents and an associated component hierarchy. They can be saved to and loaded from a persistent storage medium, where the project itself forms a folder with all contained documents as separate JSON files.

While it is possible to discard unsaved projects, it is at the time of writing not implemented to delete documents via the UI. This effect can however be achieved by removing the corresponding JSON file from the projects folder and re-loading the project.[8]

The Component Hierarchy As described in section 2.1.3, a hierarchy is used for the organization of component types. These are available as *descriptor files*, which will be detailed in section 2.2.2. Also, categories are available, to allow arranging component types hierarchically as the user desires. It is possible to load provided descriptor files. The contained component descriptors will be sorted as child items of the currently selected category or the implicitly available root item, if no category is selected. By expanding entries of component types, information about contained elements like ports, configuration bits and functional behaviour description can be accessed.

Creating Schematics After the creation of a document, component types can be dragged from the component hierarchy view to create an instance of this exact type in the schematic. The newly spawned instance will automatically be given a valid and unique name. Each document tab offers buttons to add in- and output *module ports* that connect the module described by the document to the outside world. All placed component instances and module ports can be connected by wires to form a complete circuit design.

2.2.2 User-Defined Components

As discussed in section 2.1.3, it was decided to allow users to create their own component types. Writing a JSON file that represents a certain component turned out to be quite feasible, especially, when already provided with a template which only needs to be filled in.

[8] Due to time restrictions, there is a set of similar usability functions that were not present at the time of writing. Since most of them could be emulated by simply editing a JSON file, they were considered low priority.

Listing 2.1: A minimal component descriptor file

```
{
    "name":"lut2cnf",
    "ports":
    [
        {"name":"x0",   "direction":"in"},
        {"name":"x1",   "direction":"in"},
        {"name":"y",    "direction":"out"}
    ],
    "configBits":
    [
        {"name":"c",    "size":4}
    ],
    "functions":
    [
        "[ x0,  x1, !c_0,  y]",
        "[ x0,  x1,  c_0, !y]",
        "[ x0, !x1, !c_1,  y]",
        "[ x0, !x1,  c_1, !y]",
        "[!x0,  x1, !c_2,  y]",
        "[!x0,  x1,  c_2, !y]",
        "[!x0, !x1, !c_3,  y]",
        "[!x0, !x1,  c_3, !y]"
    ]
}
```

A Minimal Example To illustrate the process of creating a custom component type, a rather minimal example is shown in listing 2.1. It describes a 2-LUT with the input ports $x0$, $x1$, the output port y and four configuration bits $c_0 \cdots c_3$. The latter are created in line 11 by declaring the existence of a group of configuration bits called c, having four members. When loading the descriptor file, *q2d* will expand this notation into separate configuration bits. It was deemed useful to use this style of notation to be able to cope with larger amounts of configuration bits, contrary to declaring each bit explicitly. Given the case that certain bits carry a specific semantic meaning, they still can be put into groups appropriately. Note that the CBD is specified as a set of clauses. There are other ways of specifying the pCBDs of a component, which will be discussed in section 2.2.4.

2.2.3 Generation of Circuit Symbols

While testing the application, it has been found, that manually creating component symbols is quite time-consuming and users prefer achieving fast results over customizing their created component types. To speed up the workflow and relief the user of unnecessary side-tasks, it was found desirable to automatically create a default component symbol for each component type.

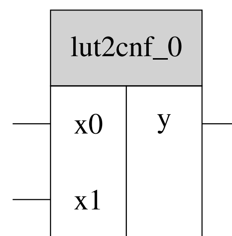


Figure 2.2: An automatically generated component symbol
Based on listing 2.1

Description of Automatically Generated Circuit Symbols Important elements for generating a circuit symbol are the component's ports, type and instance name. The latter is generated from the type name upon instantiation by appending an underscore and an incremental zero-based index.

Figure 2.2 shows the subdivision of the symbol in three main areas. On the top, the instance name is centred, while the left side contains all input ports, and output ports are positioned to the right. The position of the port symbols is calculated during the creation of the circuit symbol and added to the component descriptor. It is not required to specify port positions in the descriptor file, as long as no custom symbol is used.

User-Specified Circuit Symbols Sometimes, it feels more convenient to the user to define their own symbol for certain component types, either because they are accustomed to them or because they want to convey additional information by visual means. This can be achieved by modifying the descriptor file. It is only required:

- to add the property `symbolFile`, which specifies the location of the scalable vector graphics (SVG)-file containing the circuit symbol relative to the descriptor files location, and
- to extend the port descriptors by the positions, where to place the port symbols in the schematic. They are specified as a cartesian coordinates relative to the origin of the specified symbol file

An example can be found in listing 2.2. Note that the unaffected sections `name`, `configBits` and `functions` have been shortened to focus on the relevant changes.

2.2.4 Methods for Specifying Functional Behaviour

While working with *q2d*, it may become necessary to specify formulae. Two common examples are the pCBDs contained in component descriptors and the specification of the target behaviour. Both cases shall be examined further in the paragraphs below.

General Considerations Any formulae are specified in propositional logic. From a user's perspective, all behaviour is input as a list of strings. The elements of this list are treated as if they were conjuncted. Depending on the context, the set of valid variables may vary.

For the convenience of users, different notations for logical operators are allowed. It is possible to either use symbols or mnemonics.[9] They are listed in table 2.1.

[9] The recognition of mnemonics is case-sensitive. This might change in the future.

Listing 2.2: A descriptor file with a custom circuit symbol

```
{
    "name":"...",
    "symbolFile":"symbols/mySchematicSymbol.svg",
    "ports":
     [
        {   "name":"input0", "direction":"in",
            "pos":{"x":0, "y":30}
        },
        {   "name":"input1", "direction":"in",
            "pos":{"x":0, "y":10}
        },
        {   "name":"output", "direction":"out",
            "pos":{"x":50, "y":20}
        }
    ],
    "configBits":[...],
    "functions":[...]
}
```

Operator	Type	Symbols	Mnemonic
Negation	prefix	~ *or* !	`not`
Conjunction	infix	& *or* *	`and`
Disjunction	infix	| *or* +	`or`
Negated Conjunction	infix	*(none)*	`nand`
Negated Disjunction	infix	*(none)*	`nor`
Exclusive Disjunction	infix	^	`xor`
Negated Exclusive Disjunction	infix	*(none)*	`xnor`

Table 2.1: Overview of Logical Operator Notations in *q2d*

Notation Styles It is possible to use a clause notation like

$$\begin{aligned} &[x, !y, z] \\ &[x, y, !z] \end{aligned} \tag{2.1}$$

In this case, square brackets are used to delimit the clause, while commas separate the terms from each other. Unlike strict CNF, a clause may not only contain literals but whole boolean expressions.

The clauses of equation 2.1 could be re-written into a single expression:

$$(x \,|\, !y \,|\, z) \& (x \,|\, y \,|\, !z) \tag{2.2}$$

As another often convenient alternative one might specify a boolean equation:

$$a = (b \text{ or } c) \text{ and not } d \tag{2.3}$$

Any non-CNF style allows the usage of the equality operator and parenthesis to express precedence. The example in equation 2.3 may be transcribed as

$$a \Leftrightarrow (b \vee c) \wedge (\neg d) \tag{2.4}$$

It was shown in equation 1.6 how the equivalence can be resolved. The clause notation is mainly found in descriptor files. Each function represents one clause and all functions combined create a complete CNF specification due to the line conjunction behaviour mentioned before. For most humans though, it feels more intuitive to specify boolean formulae, which results in this version being frequently used for specifying target formulae.

CBDs in Descriptor Files When creating component descriptors, it is often necessary to express the behaviour of a component. For that purpose, the descriptor files contain a *functions*-section. Within these functions, there is only a limited set of valid variables:

ports represented by their respective names as provided in the *ports*-section, as shown in listing 2.1, lines 3 to 8.

configuration bits that are named by taking the respective bit group name, adding an underscore and a corresponding incremental 0-based index number.

Target Formulae for SAT-solving A target formula requires a module to be applied on. Any module interface name is a valid variable within the target function.[10]

[10] All releases of the application, up to the time of writing, accept the full identifier of any component port as a variable as well. This behaviour seems to have no benefits regarding productive use so far, but proved useful for debugging.

3 Implementation Details

In this section, implementation details of the application will be discussed. Since the actual source files contain several thousands line of code, as well as dozens of classes and methods, it is not viable to explain everything in detail. Therefore, several simplifications have been made in the following, to maintain the focus on the presented features.

- All classes and methods that belong to the tool's implementation always reside in the super-namespace `q2d`, even if not explicitly stated.
- Classes that originate from Qt are not explicitly marked as such. They still can be easily spotted since it is a convention within the framework that all class names are prefixed with an uppercase letter *Q*.
- The presentation has been reduced to focus on the main structure. For this reason, accessors, attributes and methods used for bookkeeping or as helpers have been omitted unless they are in some way important for explanatory reasons.
- Code elements, that were already considered deprecated at the time of writing, also have been left out.

Any description given here naturally can only reflect the application's state at a certain point in time. To obtain further information on implementation details, it is strongly recommended to check out the project repository from GitHub.[1] In addition to the documentation delivered with the source code, a wiki is available at this site to help answering questions the user might have.[2]

UML diagrams have been added to this document to aid understanding. They can be found in appendix B

3.1 Classes Involved in the Component Meta-Model

The *component meta-model* has been established in section 2.1.3. Its implementation can be found in the `q2d::metamodel`-namespace. Each part of the class hierarchy inherits from Qt's `QObject` and `QStandardtem`. This is mainly to make use of the model-view capabilities delivered by the framework and ease the representation of the hierarchy in the UI considerably.

[1] `github.com/fer-rum/q2d`

[2] Should the wiki not be able to provide a satisfying answer, an issue should be opened, which will cause the developers to expand the wiki to cover the posed question as well.

Overview over Elements Everything in the meta-model is considered an *element* of the model. Elements possess a *type*, a *name*, and may have another element as *parent*. The type may be one of those listed in the *element type* enumeration. Setting the name is achieved by using the `QStandardItem` constructor, while the parent relationship is inherited from `QObject`, narrowing down the type.

The treelike structure of the meta-model is created by the subclass of *hierarchy elements*. Additionally to their name, they also have a *hierarchy name*, which serves as unique identifier, since it includes the names of all parent elements in addition to the elements own name. The principle is similar to absolute path names used in operating systems (OSs).[3] Further, hierarchy elements are restricted to only accept *categories* as parents. The intention is, to avoid that component types contain other component types, or even themselves. A nesting of categories on the other side poses no problem.

Since *component types* are the focus of the meta-model, it is obvious, to have a subclass for *component elements*, from which component types can then be composed. Contrary to their superclass, these are restricted to component types as parent elements.

Categories To keep an overview over many component types, categories have been introduced. They may contain an arbitrary set of *component descriptors* or other sub-categories. In the UI representation, categories can be expanded and collapsed by the user to help maintaining an overview or finding certain elements.

Component Descriptors As motivated in section 2.1.3, an abstraction into component types was desired. This task is fulfilled by the `ComponentDescriptor`-class. It acts the as parent element for everything that is needed to describe a component type's behaviour and available ports. Further, information about the visual representation of a component are stored. Instances of this class are derived from the user definitions as discussed in section 2.2.2.

Port-, Function-, and Configuration Bit Group Descriptors When creating a component type, sub-descriptors are required. These are also provided by the descriptor files. Connection facilities are covered by *port descriptors*, specifying their direction and the position relative to the component symbols origin.

A pCBD, as established in sections 1.2.1 and 2.2.4, is represented by a *function descriptor*.[4] The fact that pCBDs are given as strings, is reflected in the rather simple constructor.

Additionally, component types may include *configuration bit groups*. These are given by a name and a number of members, and upon adding them to a `ComponentDescriptor`-instance, will be expanded into a set of separate *configuration bits*.

Configuration Bit Descriptors Contrary to the other parts of a component descriptor, these are not explicitly described in the respective file but are created by *q2d* to have representative elements for each configuration bit in the component. This becomes useful when generating the solution to the QBF problem.

[3] Later during the implementation, a comparable naming and identification system for component instances had been introduced. Since both systems are very alike it is possible, that they will be merged eventually. The `Identifiable` class is discussed in section 3.2.

[4] Careful readers might note that pCBDs are, strictly speaking, no functions at all. This became clear during the work on the assignment, the name stuck however.

3.2 The Document Entry Class and its Factory

Section 2.2.1 established the management of data in *documents*. Since each of these documents describes a circuit as a whole, it is obvious that a data structure must exist, that can represent the elements of which the circuit is constructed. This is achieved by introducing *document entries*. A UML-representation of the classes discussed in the following is given in figure B.2.

The Environment of Document Entries Any document entry needs to be associated with exactly one *document*, which in turn might contain an arbitrary number of entries. Amongst each other, entries that belong to the same document can form hierarchies. Like all classes in the application, that might be involved in such a parent-child relationship, the `DocumentEntry` class inherits from `QObject`. As an additional feature, this also provides access to Qts *signal-slot* syntax, reducing the effort of implementing the communication between class instances notably. Internally, the application is split into a *model*, holding the circuits structure and a *schematic* representation, used by the UI. A document entry acts as a controller to the two, subjecting them to changes depending on the operations issued by the user. It therefore is also called a *related entry*. Since depending on the underlying model- and schematic elements, different interactions are possible, each document entry possesses a type which indicates what kind of elements it represents.[5] This subdivision is discussed further in section 3.3. Document entries are aware of the overall *model* and *schematic* indirectly through their parent document, and offer accessors to them for convenience reasons.

The `Identifiable` Helper Class From the hierarchical structure of documents and their entries, the need to identify them by a name arose. For this reason, a helper class was developed, providing an abstraction for classes that require their instances to be recognized by a unique token. Upon the generation of such an identifiable object, it is only required to provide a *local name*, and an optional parent object. Should the latter not be given, the current object may serve as root of a new hierarchy. A so-called *full ID* consists of a local ID prepended with the name of each parent up to the hierarchies root.

To aid with conceptualization, it is helpful to think of a file system, where the local name is only the file or folder name perse and the full ID can be compared to an absolute path.

A useful result of this approach is the possibility to query any given document for an entries full ID and obtain the corresponding object, if there is one. Also, the `Identifiable` class creates a central point of validation for all IDs used throughout the application.

The Document Entry Factory Instantiating document entries requires a lot of secondary tasks, such as ID validation, construction of model- and schematic elements, bookkeeping. To ease the effort and reduce the likelihood of errors, a factory class has been introduced, which takes care of all the matters that are associated with instantiating a `DocumentEntry`. For this purpose, a set of functions is offered to create entries for each possible type. Since ports are always sub-entries to some component or module interface[6] , they are not instanced independently and therefore the corresponding helper function was omitted here. The default value for any of the IDs is invalid, triggering the factory to generate a valid

[5] Subclassing `DocumentEntry` also would have been a viable option. The projected gains however did not accommodate for the required efforts during the initial design. As always, this might change in the future, should the need arise.

[6] Module interfaces have been introduced in section 2.1.5.

one as a replacement. Since the user usually does not want to bother with naming everything in advance, this behaviour was deemed desirable.

3.3 Model and View

A lesson learned from the analysis of Qucs in the early stage of development was, that the separation of the internally kept model from the UI representation is imperative to maintainability and extendibility of an application. For this reason, *q2d* keeps the structure of the designed circuit independent of the visualization as a schematic.

To allow changes on the one side to be reflected on the other, document entries have been introduced in section 3.2.

Since Qt introduces a model-view relationship within its UI classes as well, in the following the term *model* will always refer to the circuit representation, if not explicitly stated otherwise.

Due to the amount of classes involved in the discussed area and their fine distinctions, in the following, a class-centred description will be given instead of a full-text presentation. This also is intended to help the reader find specific information more quickly.

A UML representation of the described model is given in figure B.3.

3.3.1 The Model Element Hierarchy

Models belong exclusively to one document each and complementary, each document has exactly one model. Both objects in this relationship are aware of each other. Viewed hierarchically, the model is subordinate to the document. This is reflected by employing `QObjects` parent property. It also allows to take advantage of the behaviour that deleting the parent object will trigger the deletion of all children as well, thus avoiding memory leaks. Models consist of a set of *model elements* that may be added via specialized methods.

Model Elements are first mentioned in section 3.2, where each `DocumentEntry` instance holds one model element, which in turn considers the entry *related*. Such model element has exactly one model as a hierarchical parent, which is just the model that belongs to the document the related document entry is part of. For convenience reasons, model elements offer access to their related entries ID accessors.

It is required, for solving the QBF problem, that each model element can yield information about the contained *configuration*, *input* and *node* variables as well as any pCBDs[7]. As a default implementation, the returned lists are empty, but sub-classes will override this behaviour, if required. Lastly, a *property map* may be requested, that contains information about the internal state of the model element. This is used for debug purposes as well as for the generation of tool tips by schematic representations.

Interfacing Model Elements form a specialized sub-class which offer the additional capability of managing ports. When queried for node variables, they query each of their ports and yield a collection of the results.

[7] Again, for historical reasons, the getter is called `functions()`, even though pCBDs are no functions at all.

Components are created by the use of a `ComponentDescriptor`, which was introduced in section 3.1. Multiple components may share the same descriptor, effectively making them instances of the same type of component. In addition to the inherited node variable getter, they can also provide configuration variables and CBDs as defined by their descriptor.

Module Interfaces are `InterfacingME`s, that have exactly one port and a direction determining the orientation of the data flow as viewed from the *outside* of the module. As a result, the directions given in the module interface and its port are opposites of each other.

Nodes are any elements that are driven by one signal source and may distribute this signal unaltered.[8] Currently, the only actual implementation of nodes are ports used for connecting interfacing model elements with each other. Technically, conductors should be considered nodes as well but, at the time of writing, no need has arisen to implement it that way.[9]

Ports additionally possess a *direction* that specifies whether data passing the port flows into the interfaced element or out of it. They further provide a custom property map, containing additional information about their connection state.

Component ports facilitate the data transfer into and out of components. If queried, they return a node variable, representing themselves.

Module ports are used exclusively by module interfaces. If the instance's `direction` is `OUT`, the associated module interface has an `IN` direction. Therefore, the port is to be represented by an input variable. Otherwise, it will yield an appropriate node variable.

Conductors are intended to connect interfacing elements with each other. Precisely speaking, they transfer data from one node to another. For this reason, they distinguish a sender and a receiver node. While from the senders view, the conductor acts as a driven element, for the receiver, it becomes a driver. While each conductor in the implementation only connects one sender to one receiver, many other tools instead abstract at the level of whole wiring nets.[10] Each conductor yields its sender and receiver as node variables and a σ-term as defined in equation 1.5.

3.3.2 The Schematics Element Hierarchy

For implementing the UI, heavy use of facilities offered by Qt has been made. This especially refers to the graphics framework, which already provides most of the base classes necessary to implement graphical representations and define user interaction. Due to the size of this framework, it cannot be explained here in detail. The reader might want to consult the Qt reference in parallel for detailed information on these classes and their interaction. When classes are derived from the framework, they often need to override

[8] Since in some contexts, the terms *driver* and *driven* may be used with different semantics, it is in the following also feasible to read *data source* and *data sink* instead, respectively.

[9] The underlying reason is, that conductors only represent a one-to-one connection, which is fairly trivial. Should these simple conductors be replaced by wire nets as done in other tools, such a net could be validly considered a node and be consequently implemented as such.

[10] It was discussed following these examples but no such changes were made up to the time of writing since it was found not realistic to implement the required changes within the given time frame. Also, the nomenclature of *nodes* would have to change for consistency reasons.

specific functions to achieve custom behaviour. These overrides have been omitted in figure B.4 to avoid cluttering the diagram.

Schematic is a specialization of a `QGraphicsScene`. It acts as container for elements to be displayed to the user. Each schematic belongs to exactly one document and visually represents the documents model. In turn, each document holds exactly one schematic, while its entries offer a convenience getter to the latter.

Schematic elements are specializations of `QGraphicObject`. A hierarchical connection between the schematic and its elements is constructed via the *scene-item relationship*, already provided by Qt's graphics framework. A schematic element might have an arbitrary number of sub-items, called *actuals*. These act as building blocks for the graphics displayed to the user. This hierarchical approach ensures that modifications like moving an element on the schematic view, will be applied to the actuals as well, while offering also flexibility to modify only selected sub-items. Further, the `scene()` getter has been overridden, to return the schematic, instead of a generic `QGraphicsScene`. A schematic element may provide `additionalJson()` and a `specificType()` that are used for serialization purposes.

Parent Schematic Elements are a specialization of schematic elements that has been introduced for type safety reasons. Again, the dependency between these elements and their hierarchical children already is given by means of the graphical framework.

Component Graphics Items represent components on the schematic. They are associated with the appropriate component descriptors, similar to their `Component` counterparts in the model as described in section 3.3.1. The hierarchy name of the descriptor will be used as the specific type in the JSON files to allow a re-creation of the component graphics item, even if the descriptor should change between saving and loading.

Module Interface Graphics Items visually represent the connections of the circuit to the outside world. The concept has been introduced in section 2.1.5. As stated there, module interfaces feature a `direction`, which specifies, if data flows into or out of the interfaced circuit.

Port Graphic Items are implementations of the concepts regarding port visualization presented in section 2.1.5. Unlike their counterparts in the model, no specializations had to be made since their representation is agnostic of the element they are interfacing. Since connecting ports with each other is done by dragging a connection from an output port and dropping onto an input port, a set of event handling functions had to be customized to allow this interaction. For the same purpose a distinct `wireDrawingMode` had to be introduced.

Wire Graphics Items visualize a data-transmitting connection from one port to another. Each wire has a data width of exactly one bit.[11] Wires will be routed automatically. For this purpose, they are internally split into `WireGraphicsLineItems`, which will be created in the process. The currently implemented routing algorithms are not very sophisticated yet, but already suffice for a huge amount of circuit designs.[12] If the start or end port is moved, the wire will be automatically

[11] Multiple bit wide connections or bus systems were considered very work intensive and were scheduled for a later release. They can alternatively be emulated by appropriate components, so users do not have to forgo this kind of feature completely.
[12] Especially referring to designs without loop-backs.

re-routed. When hovered by the mouse, all elements of a wire will highlight to help the user tracking its path. For data persistence, the wire graphics items offer a specialized additional JSON object that contains the full identifiers of the start and end port. Upon loading, the wire is routed again, so that any new algorithms available for this purpose are applied without user intervention.

3.4 The Quantor Interface

The computation of the QBF solution using Quantor as outlined in section 2.1.2 is very straight-forward. Quantor itself is written in *C*, so bridging classes to *C++* were created. Further, it is requited to build the contexts and transform them to CNF. At this point, the QBF solver can take over. Upon completion, it returns a structure representing the result, which again will be wrapped into a *C++* class.

The Solving Process The whole QBF solving process is controlled by the `QuantorInterface`. It inherits `QObject` since this enables it to use Qt's *signal and slot system*. This method of program-internal communication will trigger the solving process via the `slot_solveProblem()`.[13]

If posed with a problem, the target document will be queried for the model, from which the contexts can be build. Once available, these are fed into the `Quantorizer`, which in turn transforms them in a form that Quantor will accept as an input and trigger the actual evaluation of the QBF.

Upon completion, the solver will deliver a verdict, which might be either *satisfiable*, *not satisfiable*, *out of time* or *out of memory*. In the first case, it will also return a set of integer numbers, representing the solution found. The result object is interpreted by the `QuantorInterface` which will then present the verdict as a string and, if applicable, a mapping from the full IDs of configuration bits to their boolean value assignments that solve the problem. The found answer is propagated by emitting the `signal_hasSolution()`

Variable Representation in Quantor Internally, Quantor represents each variable by an integer number greater than zero. When using literals in clauses, the number of the variable is employed unaltered to represent a positive literal or inverted to represent a negative literal.

Context Creation in Detail It is a responsibility of the context creation process to provide a consistent mapping of the full IDs of variables yielded by model elements to their numerical representations to be used by Quantor. One context will be created for each interfacing model element, implementing the theoretical approach of equation 1.4. Additionally, there is a *global context*, in accordance with equation 1.12. Unlike the theoretical approach, the actual implementation does not merge all sets of variables and terms but rather keeps all model element contexts as children of the global context. This allows the re-use of already created `QIContext` instances. The resulting lookup overhead due to traversing an additional hierarchy level for variables was found to have a negligible impact on the time of problem solving. A further motivation for this approach was the specification of CBDs in descriptor files. These use only local names since it is not known, before instantiating a component, what its name will be and, in turn, what the full IDs of the used variables will be. Re-naming them appropriately in each of the pCBDs after component instantiation did not appear to offer any considerable benefits for the effort. For

[13] A lot of other classes discussed earlier use this mechanism for communication as well. It has not been explicitly mentioned so far, to keep things straightforward.

the proper assignment of variables to their numeric counterparts, a context must be aware of the range of values it can use for the mapping. Such a value is also called an *index*. Since contexts are not created in parallel but one after another, this can be easily achieved by passing the lowest index to be used when instantiating a `QIContext` instance. It will then collect all variables yielded by the abstracted model element. Upon encountering a new variable, a lookup will be made to see if it has already occurred in another context. This might for example be the case for node variables which occur in pCBDs as well as σ-formulae. Should this be the case, the variable's already known index will be adopted. Otherwise, the current index will be incremented and mapped to this variable. Once all variables are processed this way, the current index is the highest index used by this context and will be passed on to the next context to be used as its lower index bound. In the next step, all connections within the model elements are evaluated and σ^* as defined in equation 1.8 is introduced to the problem description. Lastly, the context also collects all pCBDs contained in the model element, or, in the case of the global context, the target formula.

The Quantorizer To transform the information gathered during context creation into a form which is suitable as input for Quantor, the `Quantorizer` will parse all CBDs as well as the target formula.[14] If any given boolean term cannot be parsed properly due to input errors or corrupted descriptor files, a `ParseException` will be thrown and the solving process will be aborted, reporting an error.[15] Otherwise, the variables contained in the term or function will be looked up and their indices used to create clauses, to be added to Quantor's description of the problem. Once done, the solver can proceed to work on the problem. Its result will be returned to the `QuantorInterface`.

Interpreting the Result The result is emitted in two parts by the `Quantorizer`. A `Result`-object is returned from the call to `solve()`, and the vector given as a parameter will be filled with the literal assignments of the solution. While the returned object can be cast to a string to obtain a textual representation, the vectors content has to be resolved back to full IDs. The sign of an element of the vector expresses if this certain variable should be configured to be either true or false. In doing so, a human-readable interpretation of the solution can be obtained and will be presented to the user in tabular form.

[14] The *quantorizer* essentially is a parser, created by the parser generator *wisent*[Preu04].

[15] Depending on the version of *q2d* used, error reporting might be facilitated by printing to `stderr`. UI support for error messages was introduced relatively late in the development process. It can be useful to launch the application from the console to not miss any messages.

4 An Example Workflow

4.1 The Task

To illustrate how *q2d* can be used productively, consider the following workflow example. For this purpose, a CLB as it is used in XILINX VIRTEX 5 field-programmable gate-arrays (FPGAs) is to be modelled. A circuit diagram of the part in question is given in figure 4.1. More information regarding the actual architecture can be found in the official manual.[1][Xili12] Further, the configuration required to realize a *full adder* with this model shall be computed.

How the Full Adder Will Work When looking at the architecture, it is easy to see that it already provides a carry chain, which shall be used. The `Dmux` output will be used for the sum, which requires the `O6` port of the lookup table (LUT) to emit the propagate bit. It is notable, that the `D` output is directly attached to the respective wire, so it could be used as a proxy for the propagate bit. If the adder does not propagate, `Cout` will assume the value of the generate bit, which in turn has to be emitted by the port `O5`. Due to the LUT's outputs having to assume different values, it is necessary to pin the input `D6` to *true*, as can be inferred from the description given in section 4.2.

4.2 A Component Descriptor for Xilinx' LUT6-2

While multiplexers are not very complicated and very obvious to describe via a formula, the LUT deserves a closer look. A special variant, featuring six input ports, but only two output ports is employed here. Research of the actual components internals yields that it is composited from two 5-input LUTs. The sixth input determines whether both output ports will emit the same signal as determined by the lower LUT or different signals, originating from both LUTs independently. This is depicted in figure 4.2, which also can be found in most of the official design guidelines[2][Xili13]. Since no appropriate component descriptor for this type of LUT is commonly available yet, it has to be created from scratch. As outlined in section 2.2.2, this can be done even with a simple text editor. The resulting descriptor file is to be found in listing 4.1. For overview reasons, the `functions` section has been shortened. Since the clauses used as pCBDs are very repetitive and systematic, users with some programming skills will most likely write a simple script to generate these lines. It can also be noted that there are different possible ways of describing the components behaviour correctly.

---

[1] Available at www.xilinx.com/support/documentation/user_guides/ug190.pdf

[2] For example at www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/virtex5_hdl.pdf

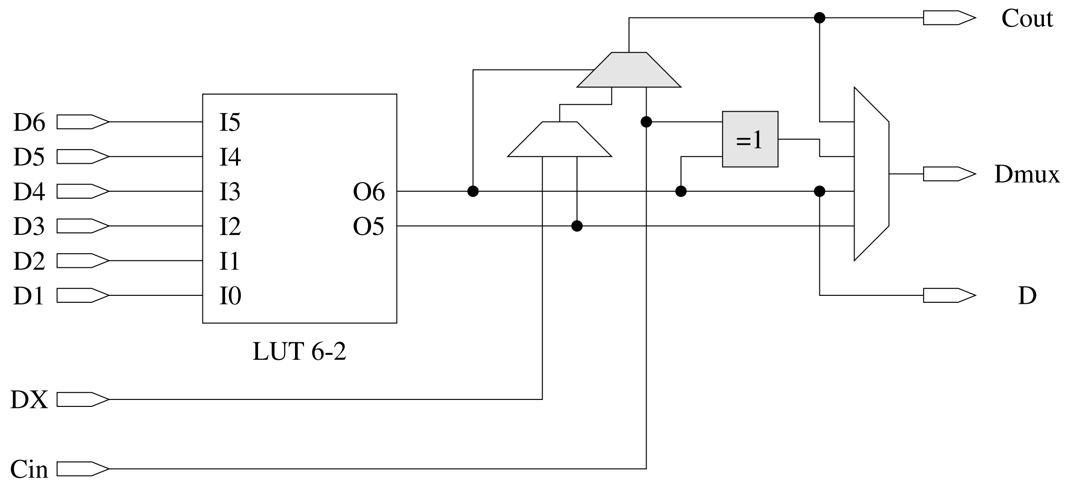


Figure 4.1: Circuit Diagram of a CLB Detail
White components are configurable.

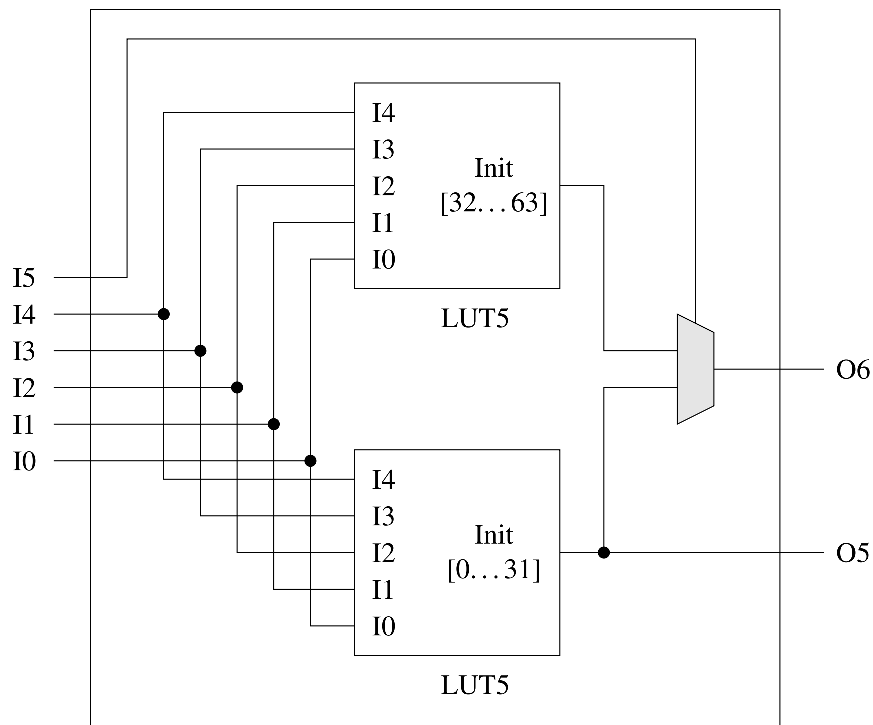


Figure 4.2: Circuit Diagram of Xilinx' LUT6-2

Listing 4.1: The Component Descriptor for Xilinx' LUT6_2

```
{
 "name":"xi-lut6-2",
 "ports":
   [
    {"name":"I0", "direction":"in"},
    {"name":"I1", "direction":"in"},
    {"name":"I2", "direction":"in"},
    {"name":"I3", "direction":"in"},
    {"name":"I4", "direction":"in"},
    {"name":"I5", "direction":"in"},
    {"name":"O5", "direction":"out"},
    {"name":"O6", "direction":"out"}
   ],
"configBits":
   [
     {"name":"Init", "size":64}
   ],
"functions":
   [
     "[!I4, !I3, !I2, !I1, !I0, Init_31, !O5]",
     "[!I4, !I3, !I2, !I1, !I0,!Init_31,  O5]",
     "...",
     "[ I4,  I3,  I2,  I1,  I0, Init_0 , !O5]",
     "[ I4,  I3,  I2,  I1,  I0,!Init_0 ,  O5]",

     "[ I5,  O5, !O6]",
     "[ I5, !O5,  O6]",

     "[!I5, !I4, !I3, !I2, !I1, !I0, Init_63, !O6]",
     "[!I5, !I4, !I3, !I2, !I1, !I0,!Init_63,  O6]",
     "[...]",
     "[!I5,  I4,  I3,  I2,  I1,  I0, Init_32, !O6]",
     "[!I5,  I4,  I3,  I2,  I1,  I0,!Init_32 , O6]"
   ]
}
```

4.3 Designing the Model

With all preparations made, it is simple to create the circuit within *q2d*. In the following, a brief walk-through will be given. It will be assumed that a new project had been created and the current document is empty.

Loading the Required Component Descriptors Users, who start with a fresh copy of the application will most likely not yet have a set of components to choose from. They may either create the required component descriptors by themselves[3] or obtain such descriptors from separate sources[4]. The actual import can be done for each descriptor via the menu Component Hierarchy > Add Component Type, which will then prompt for the descriptor file. A component library feature to ease this task has been drafted at the time of writing but is still rudimentary.

Placing and Connecting Components A Component will be placed by dragging the component descriptor from the component hierarchy onto the schematic. The schematic symbols can be dragged to another position at any time. Module interfaces can be created by clicking the appropriate button in the upper right corner of the document's tab and entering a unique name. Wire connections are created by dragging from the driving port and dropping at the driven port.

Pinning an Input to a Fixed Value As noted in section 4.1, it is required to make sure, that the input `D6` is always true. This can be achieved by connecting it to a component that always returns *true*. Doing this directly in the target formulae is possible but a preferable alternative would be to create a component, which only has one output port `out` and for example

```
[out]
```

as CBD. Connecting an instance of this component type to `D6` will yield the desired results. This approach could be expanded by using a configurable multiplexer (CMux), which is fed by an *always-true* component, an *always-false* component and the actual module input[5]. Even more degrees of freedom can be obtained by connecting all module inputs to such a CMux. Additionally, this allows to evaluate whether the signal in question needs to be pinned to a certain value or not.

4.4 Computing the Desired Configuration

Applying the Target Formulae The target formulae can be specified in a text input area near the upper border of the document's tab. Assuming that the module interfaces have been named according to figure 4.1, a possible description of the desired full adder is

```
dmux = (d1 ^ d2) ^ cin
cout = (d1 & d2) | ((d1 ^ d2) & cin)
```

[3] Which is a great opportunity to get familiar with the component creation process.
[4] A collection of component descriptors can be found at `github.com/fer-rum/q2d-components`.
[5] If desired, all this functionality could also be combined into one component.

Here, the inputs `D1` and `D2` are used for the summands , `Dmux` for the sum and `cin` and `cout` for the carry chain. Clicking the Check SAT button will trigger the evaluation of the design with respect to the input target formulae.

Interpreting the Output A separate window will inform the user about the result of the solving process. At its top, the overall verdict will be printed in textual form[6]. Further a configuration solving the SAT problem is presented in tabular form. Each configuration variable will be listed by its full ID together with the boolean value it has to assume to satisfy the target formulae. If a certain configuration bit is not included in the solution, its representing variable was eliminated during the solving process. Consequently the value assumed by this bit is not relevant for the SAT problem and the bit can be assigned an arbitrary value.

Further Modifying the Document When saved, the document is written out as a plain text JSON file within the projects folder. This allowsits easy modification outside of *q2d*. By changing the documents file, users can achieve features not (yet) supported by the UI at the time of writing. Examples are re-wiring, renaming or deleting model elements.

[6] As mentioned before, earlier versions of the application did not report errors like malformed input to the UI and rather printed a console message. Should no window for the solution appear, it is most likely, that additional information will be available there.

5 Summary and Outlook

5.1 Achieved Results

In this writing, the tool *q2d*, its theoretical background and design were presented.

Goal The intention behind this work was to create an application that allows users to easily create a circuit design containing configurable components within a graphical UI along with a specification of an intended behaviour. The tool evaluates whether this specification can be fulfilled by a component configuration and, if this is the case, also provides a configuration that solves the posed problem. It was established that this requires the solving of a QBF problem and how the design can be described as such.

Development and Design Decisions Initially, it was intended to use Qucs as a code base and employ a pure SAT solver for evaluating the designed circuits. This approach turned out to be infeasible due to design issues and the state of Qucs' code at the time of writing. As a consequence, it was chosen to implement a new tool for the intended purpose. From the experience made with the earlier attempt, several major design decisions were inferred:

- The application was developed using Qt. Retrospectively, the time invested in getting familiar with this framework seemed to be well invested due to the amount of work saved when dealing with common implementation tasks.
- The QBF solver Quantor has been employed instead of a pure SAT solver. This eradicated the need for intermediate file formats and streamlined the circuit evaluation process considerably.
- Components are not hard-coded. They instead reside in separate descriptor files, which are easy to write, read and adapt by users.
- Component behaviour is described using boolean formulae or CNF. The same applies to the description of the intended functionality of the circuit. Allowing different variations of specifying operators allows users to use the notation they prefer.
- Automatically generated circuit symbols for components relief the users of providing such for each component type they design. As a result, custom components can be developed even quicker.

Functionality and Appearance Basic project and document management features are available and users are enabled to import and employ any component they have a descriptor file for. Several visualisation techniques have been employed to allow a faster orientation within the schematic design and improve the perception of relevant information. An emphasis has been put on the presentation of component ports. Additionally, *details on demand* are available for all circuit elements.

Implementation Details The core classes of the application and their interaction with each other have been described. The focus mostly lies on their purpose and concepts. For the `QuantorInterface` the inner workings have been further elaborated . It thereby was underlined how the theoretical approach was realized in the application.

A Workflow Example To demonstrate the usage of the tool, the schematic of a *Xilinx Virtex 5* CLB has been reproduced, implementing the required custom components in the process. It was shown that a *full adder* could be implemented with such a CLB, by computing a configuration that solved the posed problem.

5.2 Suggested Improvements

Being still in early development, *q2d* still requires work to become more comfortable for users and further ease the design and evaluation process. Component libraries and generator algorithms for certain component types further reduce the effort, the user needs to undertake to obtain a customized component type. UI improvements will further enhance the users ability to perceive and interpret the configuration computed by the tool. A lot of additional features can be imagined, extending the range of application for this tool such as

- the export of configured designs as HDL,
- the extension of the model to support stateful components, and
- adding interfaces for custom plug-ins.

In the end, users working with the tool will be the best source for improvement suggestions. It is their experiences, made while using the application, that should be made as rewarding as possible.

References

[Bern10] BERNHARD PREIM, R. D.: *Interaktive Systeme, Band 1*. (2010)

[BrJa08] BRINSON, M.; JAHN, S.: *Qucs: A GPL software package for circuit simulation, compact device modeling and circuit macromodeling from DC to RF and beyond*. http://www.mos-ak.org/eindhoven/papers/06_Qucs_MOS-AK_Eindhoven.pdf, April/2008

[Eng+02] ENGELSON, V.; FRITZSON, D.; FRITZSON, P.: *Automatic Generation of Graphical User Interface from C++ Data Structures.* In: PRIBEANU, C. (Hrsg.); VANDERDONCKT, J. (Hrsg.): *TAMODIA*, INFOREC Publishing House Bucharest, 2002. – ISBN 973–8360–01–3, S. 72–77

[Lin+07] LING, A. C.; SINGH, D. P.; BROWN, S. D.: *FPGA PLB Architecture Evaluation and Area Optimization Techniques Using Boolean Satisfiability.* In: IEEE Trans. on CAD of Integrated Circuits and Systems, Vol. 26 #7 (2007), S. 1196–1210

[Nar+06] NARIZZANO, M.; PULINA, L.; TACCHELLA, A.: *Report of the Third QBF Solvers Evaluation.* In: JSAT, Vol. 2 #1-4 (2006), S. 145–164

[Preu04] PREUSSER, T.: *The wisent Parser Generator*. Fakultät Informatik, Technische Universität Dresden, Forschungsbericht TUD-FI04-1, 2004. – ISSN 1430–211X

[SaBa05] SAMULOWITZ, H.; BACCHUS, F.: *Using SAT in QBF.* In: VAN BEEK, P. (Hrsg.): *CP* Band 3709, Springer, 2005 Lecture Notes in Computer Science. – ISBN 3–540–29238–1, S. 578–592

[Saf+06] SAFARPOUR, S.; VENERIS, A.; BAECKLER, G.; YUAN, R.: *Efficient SAT-based Boolean Matching for FPGA Technology Mapping*. In: *Proceedings of the 43rd Annual Design Automation Conference*. New York, NY, USA: ACM, 2006 DAC ’06. – ISBN 1–59593–381–6, S. 466–471

[Sch+10] SCHLEGEL, T.; RASCHKE, M.; KNITTIG, M.; DRIDIGER, S.; TARAS, C.: *Evaluation of Current User Interface Generator Frameworks for Graphical Interactive Systems*. In: *Proceedings of IADIS International Conference Interfaces and Human Computer Interaction*, 2010

[Tor+14] TORRI, G.; BRINSON, M.; SCHREUDER, F.; ROUCARIES, B.; NOVAK, C.; CROZIER, R.: *Building a second generation Qucs GPL circuit simulator: package structure, simulation features and compact device modelling capabilities*. http://www.mos-ak.org/london_2014/presentations/09_Mike_Brinson_MOS-AK_London_2014.pdf, März/2014

[Xili12] XILINX: *Virtex-5 FPGA User Guide*, März/2012

[Xili13] XILINX: *Virtex-5 Libraries Guide for HDL Designs*, Oktober/2013

Appendix A

Acronyms and Glossary

CBD Component Behaviour Descriptor 2, 14, 17, 23, 25, 26, 30, *Glossary:* Component Behaviour Descriptor

CLB configurable logic block 1, 5, 10, 27, 34

CMux configurable multiplexer 30

CNF clause normal form 3–5, 8, 16, 17, 25, 33

Component Behaviour Descriptor The set of all pCBDs referring to the same component context forms the holistic description of the components behaviour. A formal definition can be found in section 1.2.1. 2

DIN *Deutsche Industrienorm*, German industrial standard. 11

FPGA field-programmable gate-array 27

GitHub A web-based provider for *git*-repositories. The site features convenient tools for many standard tasks that come with the management of projects and repositories. The official website can be found at `www.github.com`. 19

HDL hardware design language 9, 10, 34

JSON JavaScript object notation 8, 9, 13, 24, 25, 31

LUT lookup table 14, 27

OS operating system 20

partial Component Behaviour Descriptor A boolean formula, containing a subset of the node and configuration variables of a component context, is called a *partial* CBD. All pCBDs of a component form the CBD. See also section 1.2.1 and the entry for Component Behaviour Descriptor. 2

pCBD partial Component Behaviour Descriptor 2, 14, 15, 20, 22, 25–27, *Glossary:* partial Component Behaviour Descriptor

QBF quantified boolean formula 1, 2, 4, 8, 10, 20, 22, 25, 33

Qt A framework based on the *C++* programming language. It extends the language features of *C++* by the use of a meta-compiler and and thereby enables amongst other things the use of introspection and signal-slot concepts. Common OSs like *Linux*, *Windows*, *OS X* or *Android* are well supported. Furthermore, libraries are offered for a lot of standard tasks like the abstraction of graphical UIs, JSON-parsing, device interaction, unit testing. . . integrated development environments (IDEs) specialized for the development with *Qt* exist and provide support for the visual design of UIs and internationalisation of developed applications while also integrating user-defined external tools and workflows. The official website can be found at `qt-project.org`. 5, 8, 19, 21–25, 33

Quantor An open-source solver for QBF problems. It was developed by the Johannes-Kepler University of Linz and is written in *C*. Internally, the SAT solver *picosat* is employed. Both tools can be found at `fmv.jku.at/quantor` and `fmv.jku.at/picosat` respectively. 5, 8, 10, 25, 26, 33

Qucs The *Quite universal circuit simulator* is an open source tool for the visual design and simulation of analogous circuits. Recent development has expanded it to offer basic design and evaluation features for digital components as well. An exhaustive description can be found at [BrJa08] in combination with [Tor+14]. The official website can be found at `qucs.sourceforge.net` 7, 9, 22, 33

SAT satisfiability 4, 5, 8, 31, 33

SVG scalable vector graphics 15

UI user interface 1, 5, 8, 9, 13, 19–23, 26, 31, 33, 34

UML unified markup language 9, 11, 19, 21, 22

XML extended markup language 8

Appendix B

UML Diagrams

The following UML diagrams are referenced and explained in chapter 3. It is recommended to read it beforehand or in parallel to inspecting the diagrams to facilitate understanding the topic.

Qt classes At the time of writing, Qt version 5.4 was used in the implementation. The official documentation for these classes can be found at `doc.qt.io/qt-5/classes.html`

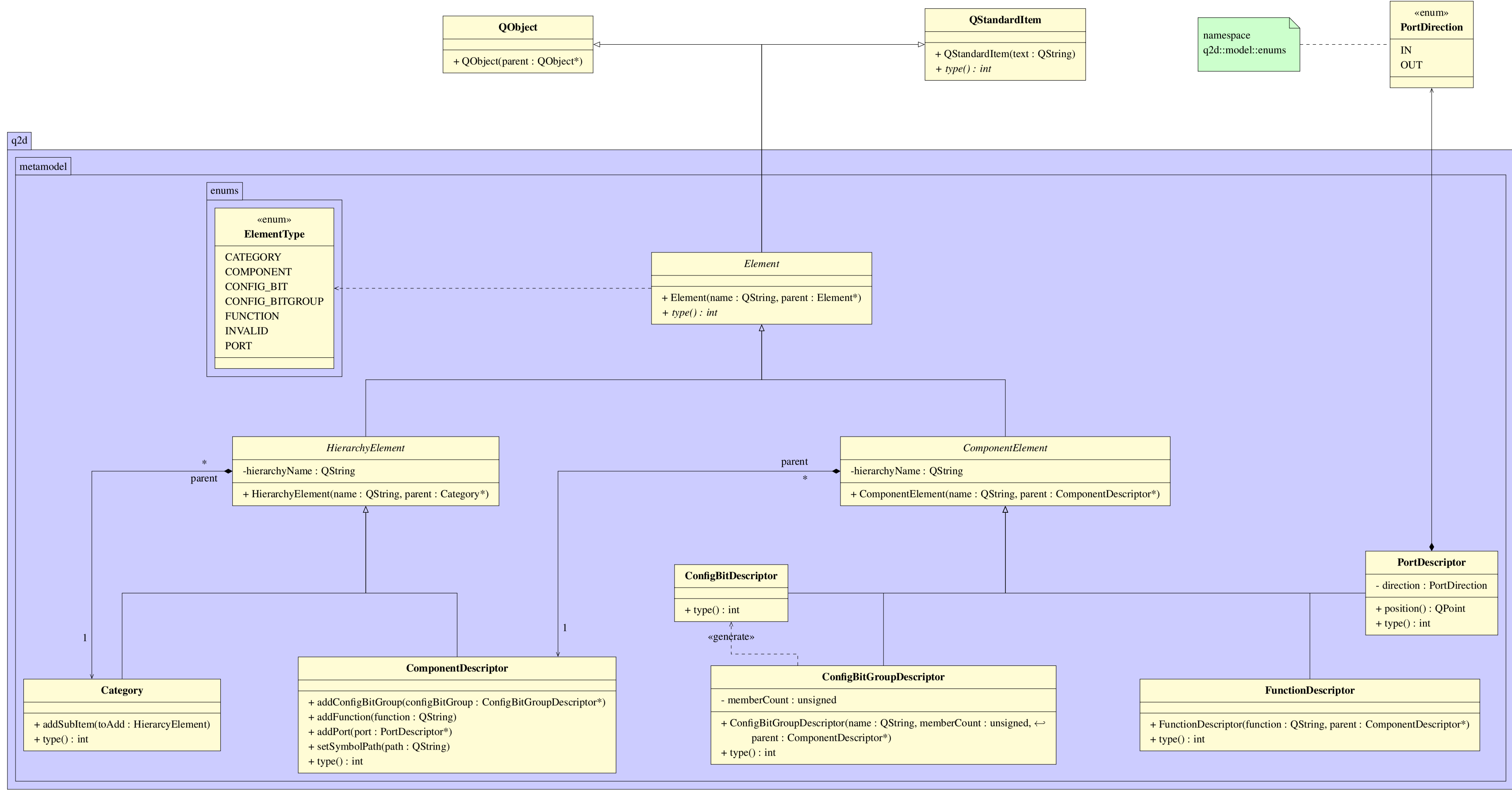


Figure B.1: UML-Diagram for the Component Meta-Model

Figure B.2: UML-Diagram for the Document Entry and its Factory

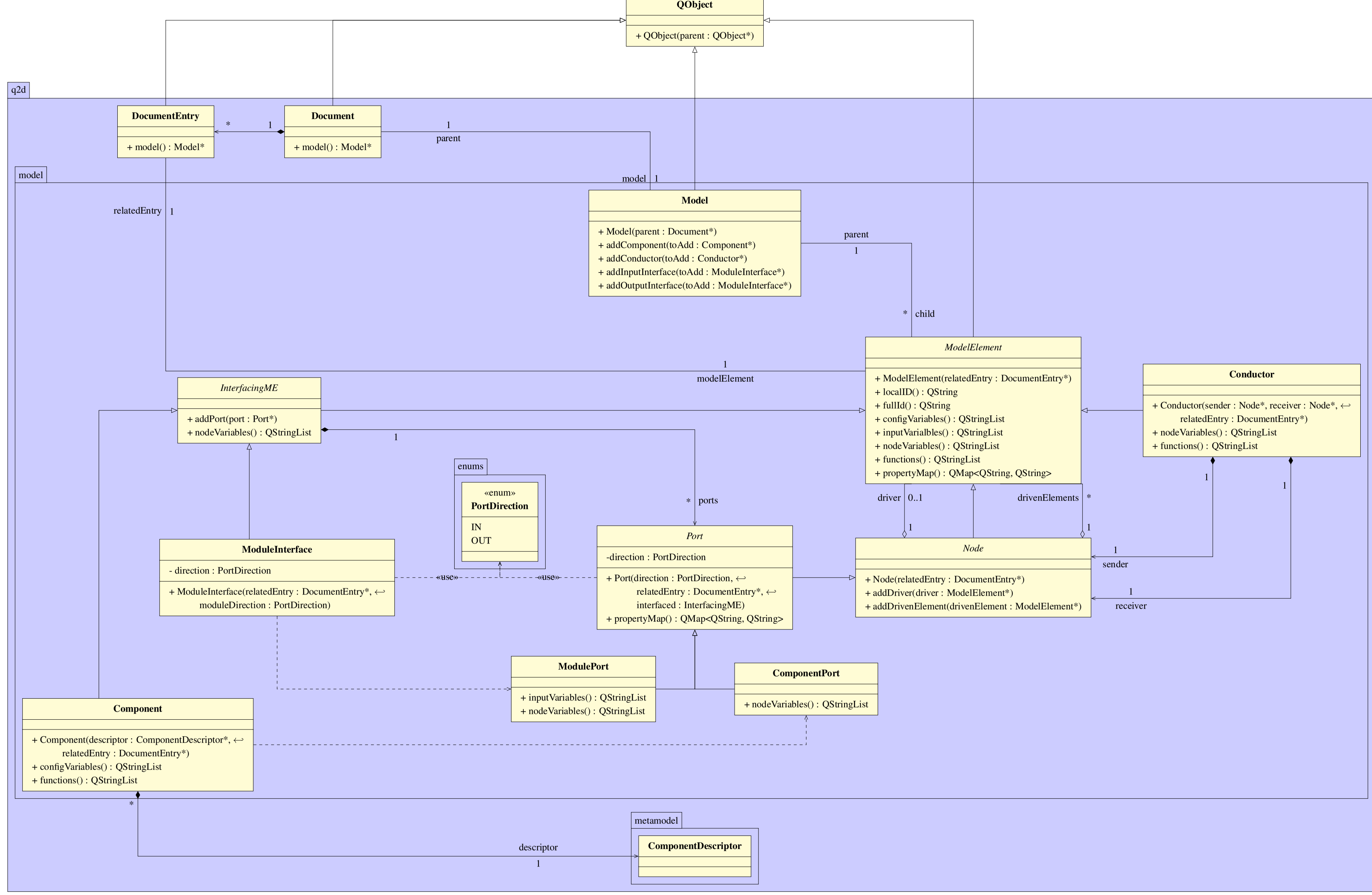


Figure B.3: UML-Diagram for the *q2d*-Model

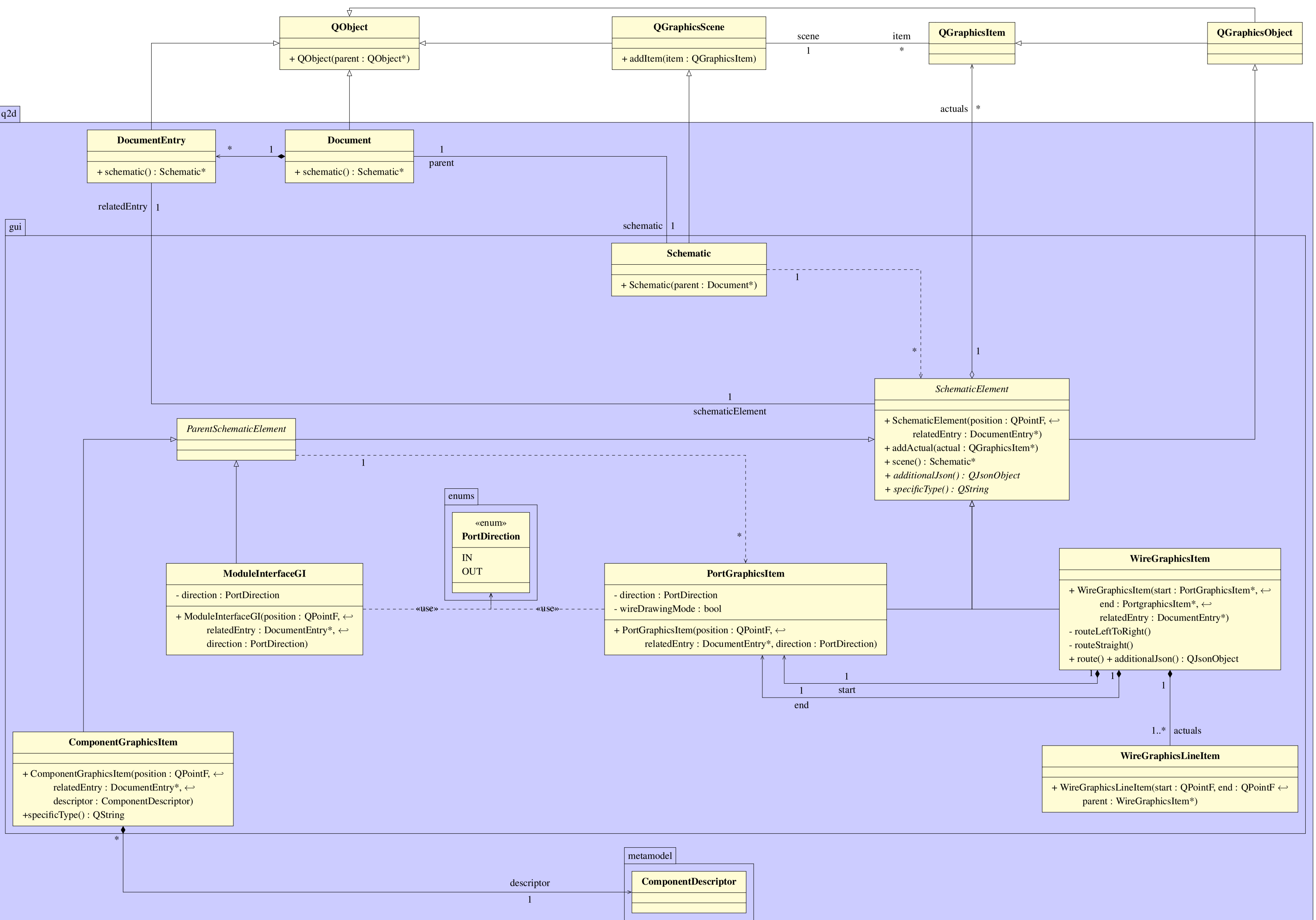


Figure B.4: UML-Diagram for the *q2d*-Schematic visualization

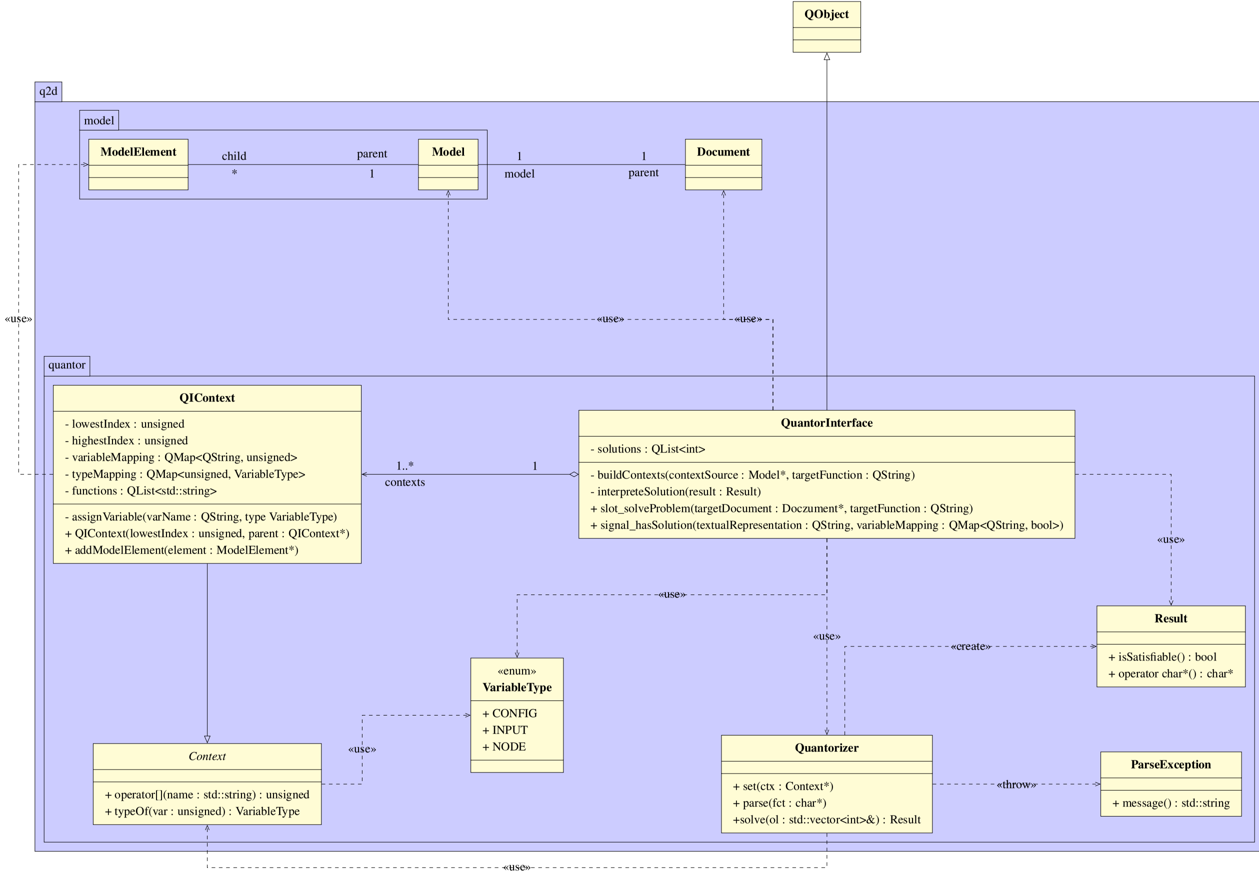


Figure B.5: UML-Diagram for the *q2d*-Quantor Interface