FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Hardware Description with Timing Requirements

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Stefan Resch
Matrikelnummer 0425306

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Steininger
Mitwirkung: Univ.Ass. Dipl.-Ing. Dr.techn. Matthias Függer

Wien, 22. November 2011

(Unterschrift Verfasser) (Unterschrift Betreuer)

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

Erklärung zur Verfassung der Arbeit

Stefan Resch
Koberweingasse 2/37
1150 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 22. November 2011 ______________________________
(Unterschrift Verfasser)

To my family.

Kurzfassung

Hardware Designs die für sicherheitskritische Anwendungen entwickelt werden müssen unter allen Umständen korrekt funktionieren. Eine Möglichkeit dies sicherzustellen ist durch (i) Beschreiben der Spezifikation in einem formalen Framework und (ii) Beweisen des korrekten Verhaltens der Designs auf der Abstraktionsebene dieses Frameworks. Bei diesem Ansatz bleibt immer noch eine Beweislücke zwischen der Darstellung des Designs im Framework und dessen Codierung in einer für die Fertigung verwendeten Hardware Beschreibungssprache.

In dieser Arbeit wird die neue Hardware Beschreibungssprache Dhdl vorgestellt. Dhdl erweitert das Formal Modeling and Analysis Framework von Függer [9]. Dadurch wird die Beweislücke zwischen der formalen Analyse und der Hardware Implementierung geschlossen. Mit dem Dhdl spezifischen, semantischen Zeitmodell können auch Simulationsergebnisse von Dhdl in das Formale Framework rückgeführt werden. Die Möglichkeit Zusicherungen über das Zeitverhalten direkt im Dhdl Design selbst zu spezifizieren ist ein großer Vorteil gegenüber aktuell verwendeten Beschreibungssprachen wie VHDL. Um die breite Anwendbarkeit von Dhdl zu zeigen wurden zwei Designs implementiert und deren Zeitverhalten analysiert: ein synchrones, am Beispiel einer einfachen Kaffeemaschine, und ein asynchrones, am Beispiel des DARTS Pulse Generation Algorithmus [21].

Abstract

Hardware designs intended for use in high reliability missions are required to behave correctly under all circumstances. One way of establishing their validity is by (i) stating a specification in terms of a formal framework and (ii) proving them correct within the framework. However, a proof gap between the formally stated design and the actual specification in terms of a hardware description language remains.

In this thesis, a novel hardware description language, called Dhdl, is introduced. Dhdl is tailored to the formal modeling and analysis framework defined by Függer in [9]. Thereby, it closes the gap between formal proof methods and hardware implementation. With its semantically defined timing model it is even possible to translate traces back to the modeling and analysis framework. The ability to specify timing properties within the design itself is a major advantage of Dhdl to currently popular hardware description languages like VHDL. To demonstrate the wide applicability of Dhdl, both a synchronous (a simple coffee machine) and an asynchronous hardware design, namely the pulse generation algorithm of DARTS [21], have been implemented and their timing constraints have been analyzed.

Acknowledgments

I want to thank Matthias Függer and Andreas Steininger for giving me the opportunity to choose this topic, and for their helpful feedback. I would also like to say thank you to Michael Zehender for introducing me to the method of test driven development and his feedback on the usability of Dhdl. Special thanks to Cristina Niculescu, my friends and my family who supported me during my studies.

Contents

CHAPTER 1

Introduction

In this thesis, the discrepancy between the definitions of systems within a formal framework and their description within a machine readable code is reduced. This is achieved by introducing a novel programming language for hardware description.

The core issue are hardware designs intended for use in high reliability missions that are required to behave correctly under all circumstances. One way of establishing their validity is by stating a specification in terms of a formal framework and proving them correct within the framework. However, a proof gap between the formally stated design and the actual specification in terms of a hardware description language remains. The problem here consists of different forms and levels of abstraction and whether they express the same system.

Abstraction is the key concept for describing complex processes, systems, et cetera. With the right level of abstraction it is possible to efficiently describe large systems in a way that is still feasible for a human mind. With each layer of abstraction, detailed information is lost in order to be able to focus on the core issues. It is crucial for these models to contain all the necessary information needed within and above the current layer of abstraction.

This leads us to the comparison of things, which can be a trivial task, e.g. no one will argue about 3 not being equal to 3. But showing equality of more complex systems might not be as simple. This gets even more complicated when using different languages. The best example here is the transformation of an English written specification to a program. Also, when using abstract models to prove properties of specific algorithms and implementing them later in code is a manual and quite error prone task. In the best case it would be possible to let a reliable tool generate verified code. To at least make this task quite simple for humans it would be of great advantage if the same language could be used. If this is not available due to the lack of expressiveness of the formal models which is necessary to ensure implementability, it is at least a great help if the style of description and the important keywords are the same in the specification and design entry.

1.1 Motivation for a New Hardware Programming Language

In [9], Függer defines a formal analysis and modeling framework for hardware, whose key feature is the ability to argue with timing properties about the correctness of circuits. As powerful as this framework is for formal analysis, it is incapable of describing larger systems in an efficient human readable way. Also, no currently available language for hardware description can be mapped directly from this framework to hardware and so a new language was the best approach. Dhdl has its basis in this framework but extends it with powerful expressions and syntactic sugar.

1.2 About Time

In classical synchronous designs time itself is not modeled. Instead, the circuits depend on a clock which has a dependency on time. The first simulation in the design flow, the behavioral simulation, has no concept of the causality of signals and rather shows the transformation of the system states from one well defined point in time to another and by this does not capture any possible occurring intermediate states. When going further down the design flow, timing behavior gets more detailed, but the simulations become more expensive with respect to the calculations they need. To simulate systems without clock at the behavioral level one must be creative, when using common tools.

To overcome this problem, the time model used in this thesis will be called causal time and will be derived from the structure of the circuit.

1.3 Methodology

The new language is designed starting from the formal framework's elements and refined with concepts from modern programming languages and general hardware design. To show that the language is powerful enough for implementing real-life systems, two reference designs are implemented. These designs are simulated in Dhdl and tested on a FPGA board for functionality. Also, one small reference module is implemented to analyze all aspects of Dhdl in detail. For this module, the correctness of the generated VHDL code is checked by means of simulation. The output is then compared with the Dhdl simulation. The results of the timing properties for this module are also checked manually. Furthermore, a formal semantic for a sufficiently large subset of the language is defined. This is used for better understanding of the trace generation and to test the behavior of the Dhdl simulator. The traces generated by the semantic interpretation are shown to be compatible to traces within the formal framework. With the reference module, the semantic interpretation of a specific simulation and design description is demonstrated.

That the method of semantic interpretation can be applied to a simulation and design description and generate a valid trace is also demonstrated with the reference module.

1.4 Related work

Starting with solutions for timing constraints only, the tools from Altera [1] and Xilinx [24] offer the possibility to specify assertions about delays on specific paths. These are absolute assertions and cannot be annotated in VHDL directly. This problem is overcome in [6], where VHDL is extended by timing constraints for the behavioral simulation with procedures. These constraints are translated to constraints for the synthesis tool. However, they still have a defined time range: if one wants to specify that one delay must be longer than another one, it has to be expressed with specific values.

Without explicitly stating timing constraints, in [22] the behavior of hardware is modeled with CCS and by this, naturally, a relative timing is defined. From this the logical behavior can be deduced, but it still is hidden in the specification. Thus, this method is different with respect to specifying timing constraints from all currently used languages. Also, it is not possible to check an absolute delay.

Only limited timing assertions are used in [20], where Haste programs are translated to asynchronous hardware and the timing needed for the handshake is tested and corrected. The program logic itself is not verified and there is no timing annotation within the Haste programs available. Also effects of glitches on the handshake signals are not taken into account.

Without considering signal delay properties at all, problems on a higher level are solved in [14]. In this work a model checker is used to verify the behavior of Rebeca programs which are then translated to hardware specifications. This approach to verify a specification and derive physical hardware from it is not as low level in the specification as our formal framework and its output is restricted to synchronous designs.

1.5 Remarks

The Dhdl version described in this thesis is Dhdl 0.9. I it is planned to develop it further and change small parts of the syntax.

Syntax in this thesis will be annotated as defined in [3, p. 28]. Please note that optional whitespace characters, that can usually be placed anywhere within the languages, will be ignored in the syntax description for the sake of simplicity. Only required newline characters are annotated.

In the main part of this thesis only the ideas and concepts of Dhdl will be described, as well as the formal semantics. The detailed Dhdl language specification can be found in Appendix A. For better readability, it also contains information covered within the first chapters of this thesis.

All the concepts presented have been implemented in Antlr and Java.

1.6 Structure of the Thesis

In Chapter 2 the underlying time model of Dhdl is described in detail. The general concepts and ideas of Dhdl are discussed in Chapter 3 and its implementation in Chapter 4. In Chapter 5 the formal semantics for Dhdl is defined and the reference designs are explained in Chapter 6 followed by the conclusion in Chapter 7.

CHAPTER 2

Causal Time

The causal time model is used as time basis for the simulator and the formal semantics. With this model it is possible to derive the simulated timed behavior from the circuit structure by means of the syntax. In our model it is stated that when a signal a is connected to a signal b with the information flow oriented from a to b, the changes appearing on b happen exactly a tick later on signal a. Ticks are used as basic time unit and one tick is the smallest amount of time that can pass within this model. To calculate the values for the next tick, a simulator is said to take a step. It is important to remember that the traces generated in this way are example traces of the design which might or might not appear similar in the final system. However, it gives a feeling of the causal relations of the signals within the system and a defined deterministic behavior which is very valuable for testing the circuits. Also, synchronous designs can be tested in this way, but special care must be taken when modeling the clock period. Here, the clock period is the time within two triggering edges of the clock signal, regardless of whether they are rising ones, falling or both. In synchronous hardware the clock period must be wider than the longest delay path found within the combinational logic of any two registers. This property has also to be fulfilled when simulating synchronous designs with causal time.

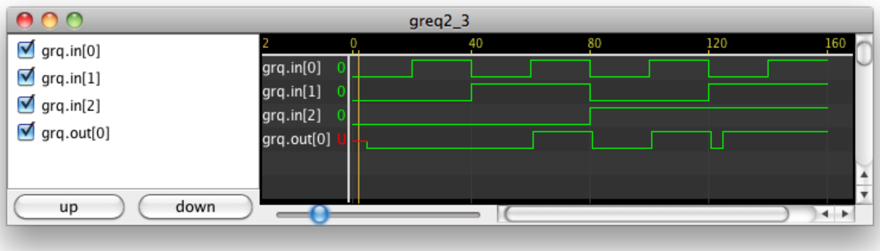

Figure 2.1: The trace of a `grequals` simulation.

Figure 2.1 shows a simulation trace of the `grequals` module, defined in the Dhdl library, with the configuration that it has true as output if two or more input signals are true. It is observable how this module changes the output with some delay when a new input is given. Even a glitch for the output is visible shortly after tick 120. Therefore, for writing tests it is necessary to keep the underlying time model in mind and to know when the circuit reaches the desired state. The longest path, by means of occurring delayed connections(<=), from any input signal to any output signal is here of great help. This value is not yet calculated by Dhdl.

2.1 Timing Behavior in the Functional Simulation

The functional simulation of hardware designs provides a timing behavior where each generic gate has a unit delay. At this level of the design flow, the hardware description code has already been translated to gate logic. However, the causal time model is directly derived from the syntax. Through this, the designer can predict the causal time behavior immediately from the code and specify simulation tests accordingly.

2.2 Compatibility with the Formal Analysis and Modeling Framework

The formal analysis and modeling framework uses continuous time and specifies the behavior of circuits by events happening at input ports and those eventually appearing within a certain time range later at the output ports. An appropriate hardware model using causal time can be shown to coincide with specifications stated in the formal framework when mapping the transitions in the causal time model to transitions in the continuous time model of the framework. More on this can be found in Chapter 5.

2.3 Timing Assertions for Generated Circuits

When connecting two signals, this naturally limits their relative timing behavior within generated circuits. Assuming that the signals a and b are connected with the information flow from a to b (b<=a), then a transition appearing on a must happen at the same time or later on b. Of course, the transition cannot be performed earlier. One could argue that there must be a certain delay at any case, but we will stick here to observable behavior. Also, when generating hardware, some signals might be removed or merged for optimization, which is favorable in most cases. Thus, these transitions at a and b could actually happen at the same time.

Through this the timing assertion for the generated hardware is that the delay between the value of the right side of the assignment and the resulting value of the signal on the left is greater or equal to zero. As already mentioned, the model of causal time satisfies this constraint with a fixed delay of one tick.

CHAPTER 3

Definition Concepts

This chapter describes the concepts and intentions of definitions in Dhdl. For a detailed syntax specification and useful examples, please refer to the Dhdl language specification in Appendix A. Some parts in the language specification might have already be defined and explained within this chapter, but are nonetheless included there for completeness.

Hardware, simulation and package definition are three different problem domains with the intention to design an integrated circuit. For each of these domains a separate language concept is presented, although they use the same core idea and similar syntax.

3.1 Hardware Definition

Todays most commonly used hardware description languages, like Verilog or SystemC, use an imperative approach for defining systems. Their concept of simulation is important for their language design, just take a look at the construct of sensitivity lists in VHDL. Dhdl has the model of causal time (see Chapter 2) to be able to take a different approach.

The basic element in Dhdl is a signal. These signals are kept within modules that have in- and output ports that are also signals. Each signal must be assigned exactly one expression. In the most simple case these are Boolean expressions. From the point of view of a module, all internal signals, output ports of the module and input ports of submodules are assigned within it. All other signals are assigned somewhere else. For the simulator it is now possible to use these expressions and the current values of the signals to determine all values after one tick time has passed. In this sense, Dhdl is similar to a functional programming language. This part of Dhdl is also possible to be modeled by the formal semantic defined in Chapter 5. Theoretically, these constructs are sufficient to model all possible circuits, but it might be more than inconvenient. The solution to this can be found in Section 3.1.4, where many more features of Dhdl are described that make designing hardware easier. And in Section 3.1.2 the available expressions will be explained in detail.

3.1.1 Types

All languages in Dhdl are strongly typed, which is an immediate result from the problem domain. There are five primitive types, $\mathcal{W}$, $\mathcal{B}$, $\mathcal{N}$, $\mathcal{I}$ and $\mathcal{D}$. $\mathcal{W}$ ranges over binary values and "undefined". $\mathcal{B}$ is for binary values, $\mathcal{N}$ for natural numbers including zero and $\mathcal{I}$ for integer values. Values in $\mathcal{D}$ define a range over decimal numbers. The order is $\mathcal{B} \subseteq \mathcal{W}$ and $\mathcal{B} \subseteq \mathcal{N} \subseteq \mathcal{I} \subset \mathcal{D}$. Only expressions of the wire type $\mathcal{W}$ can be later generated into hardware. The only two representatives of $\mathcal{W}$ are `Signal` and `Register`. $\mathcal{D}$ is used for checking timings and all other types are for generating hardware structure.

In the following, the term variable will be used to refer to a primitive type as well as a module.

3.1.2 Expressions

The key construct for describing logic circuits in Dhdl is the guarded expression. A guarded expression consists of subexpressions, where each has its own guarding expression and a resulting logic expression. In the most trivial case the guarding expression is empty and thus true. It can also be a list of expressions, where, if and only if all expressions evaluate to true, so does the guarding expression. The result of a guarded expression is the result of the first logic expression for which its guarding expression evaluate to true. Listing 3.1 shows how a simple $c := a \wedge b$ can be encoded in that way, provided that the part with "d <= true" is ignored. The first subexpression sets c to true if the guarding expression is true, and through this a and b are true. The second subexpression sets c to false in all other cases, since its guarding expression is empty.

Listing 3.1: Guarded and-gate expression.

```
d <= false                    // default value of d
c <=
    a == true, b == true      // first guarding expression
        -> d <= true, true    // side effect d set to true, c assigned true
        -> false              // "else" path, since the second
                              // guarding expression is empty
```

Another feature is that guarded expressions can also have side effects. When a guarding expression evaluates to true also the value of other signals can be set, provided that those signals have a general expression for determining their value otherwise. If that would not be the case, latches could easily be created by accident. Cross referencing guarded expressions is not recommended but it is currently not checked during translation. Again Listing 3.1 provides an example, this time for an expression defining $d := a \wedge b$ using side effects within the expression for c. More sophisticated examples for the use of guarded expressions can be found in the appendix in Section A.9.

3.1.3 Delay Annotation

The proofs in the formal analysis and verification framework require modules to satisfy certain time constraints. The key feature of Dhdl is to annotate these constraints directly within the definition of the modules. Through this, it is possible to check a variety of timing properties of the signals in the module and submodule. These expressions often have to use signals from within submodules. These expressions are the only ones that can also depend on structures of other modules and by that reduce the flexibility of the design, since there might occur problems when refactoring the used submodules. However, with each translation of the code, also the existence of the referred signals is checked and thus it should be possible to adapt these expressions quite fast during development.

Listing 3.2 gives two delay annotation examples. The "require_module_range" delay property states that the timing from the in to the out port of mod must be in between one and two nanoseconds. The second delay property, "require_slow_chain", states that the sum of the delays from the in to the out port of $mod1$, as well as the in to the out port of $mod2$, has to be greater than 10 nanoseconds.

Listing 3.2: Two delay annotation examples.

```
delay require_module_range := 1ns < (mod.in, mod.out) < 2ns
delay require_slow_chain := 10ns < (mod1.in, mod1.out) + (mod2.in, mod2.out)
```

3.1.4 Extended Description Concepts

With good programming languages it is possible to define solutions for common problems of their domain in a readable and comprehensive style. The concepts introduced in this section are intended to achieve exactly this for Dhdl.

Synchronous Designs

To design synchronous systems in a more comfortable way, Dhdl supports a special signal, namely register. A register in Dhdl is a signal that changes its value to zero when the reset signal is one, to the value of expression when there is a rising edge at the clock signal and keeps its previous value otherwise. The signals reset and clock must exist in the module where the register is defined and they are automatically connected.[1] When using port forwarding (see 3.1.4) for clock and reset interconnect, designs can be kept simple with respect to annotation of these networks.

Using a register in combination with a multiline expression yields a compact annotation of a state machine.

Interconnection of Ports

For convenience it is possible to just use a submodule's reference name to access certain ports. When assigning an expression that way it is connected to the input port "in" and the usage within an expression accesses the output port "out". Aside from the explicit annotation of submodules, it is also possible to embed a module within an expression. For this the module must be designed with the "out" output port and an input port parameter list. A simple example is given in Listing 3.3, where in line 8 the input port $inv.in$ is assigned a and in line 9 the output port $inv.out$ is assigned to b.

[1]The limitation to a high active reset and the clock trigger at the rising edge will be removed in later versions.

Listing 3.3: Using the submodule's reference name for port access.

```
module invert {
        in Signal in
        out Signal out <= not in
}

module result {
        in Signal a
        invert inv <= a
        out Signal b <= inv
}
```

Input ports marked with the keyword forward will be automatically connected to all not explicitly connected input ports of submodules with the same name. In Listing 3.4, the keyword forward is used in line 2 and through this, the signal $clock$ is automatically assigned to the signal $counter.clock$.

Listing 3.4: Using the keyword forward for automatic interconnect.

```
1   module  {
2           forward in Signal clock
3           Counter counter
4           out Signal out[4] <= counter
5   }
6
7   module Counter {
8           in Signal clock
9           Register out <= increment(out)
10  }
```

Interfaces

Interfaces in Dhdl describe only the available ports of a module. As in other languages, like Java, interfaces are used to represent modules as black boxes and assure port compatibility within different implementations. Each module can implement any number of interfaces.

Parameters

Parameters are used to make modules and interfaces generic and to allow modules to be embedded directly into logic expressions. For easy distinction there are two types of parameters, namely generic parameters and signal parameters. They are differentiated by syntactical annotation and supported parameter types to force a clean separation.

Wiring Code

In hardware designs, problems are often solved by using circuit cells of the same structure and wiring them in a specific way. This can be done in Dhdl with wiring code. This wiring code is executed in a sequential manner and yields a module with the basic assigned signal structure for simulation and code generation. Within the code it is possible to define and assign signals, add delay constraints and also if-structures, as well as for-each-loops are available.

Intentionally, only loops of the for-each type are supported in order to encourage programmers to think about how to treat each of their used submodules systematically. It is also sufficient, since all signals must be assigned exactly once. The for-each loop construct of wiring code takes a variable and an identifier in square brackets as shown in Listing 3.5. The identifier will then be used as iteration variable within the code body of the loop, holding the current index value. In this example all input ports in of the submodules $cell$ will be connected to the specific input ports in of the whole module. This code, by definition, guarantees that all input ports in of the submodules are connected, but not that all input ports in of the current module are used.

Listing 3.5: For-each loop example.

```
foreach cell[i] {
        cell[i].in <= in[i]
}
```

Assertions

A common method to find flaws when writing software is to use assertions. Assertions in the hardware description of Dhdl have the scope of a module and are checked before the wiring code is executed. The main goal of these assertions is to check whether the generic parameters have valid values and the ports arrays are within the supported ranges.

Packages

To keep the designs well structured the concept of packages is used. Similar to packages in Java, packages in Dhdl are folders within the file structure and their modules can be loaded with import statements. Currently, no wildcard characters are supported.

Macros

Macros are accessed and imported like regular modules, but can yield specific simulation objects or code. Macros are defined in Java code and by this are able to extend the Dhdl language with numerous functions.

Replacement Terms

For each package it is possible to define identifiers that stand for expressions. These identifiers will then be replaced within all other expressions with the defined one. This is similar to C-macros and can be used for a designer's own constants.

3.2 Simulation Definition

Simulating hardware designs at an early stage of development is necessary to write working code. With the model of causal time in Dhdl it is even possible to have some sort of time dependence in the first possible simulation level. This is contrary to current available behavioral simulations of languages such as VHDL.

A simulation run written in Dhdl has the intention to show that a specific functionality of the design is correctly implemented.[2] For small modules, single simulation runs might also cover their complete specification. Different to the hardware definition, the simulation language is imperative. Mostly timed behavior is tested which is done by giving a specific input and waiting for a corresponding output. Thus, an imperative execution of the specification best reflects the problem domain.

The simulation principle of the Dhdl simulator is time based. The causal time model defines the minimum time difference with one tick causal time. When incrementing the causal time by one tick in a simulation run, all new signal state values are calculated based on the old signal state values. Through this method, timing loops are avoided.

Usually, there are several simulation runs executed for testing. If one of them fails the other ones are still run afterwards, since, for finding errors, it can also be useful to know exactly which simulation runs fail and pass. In a simulation file, several simulation runs can be specified.

3.2.1 Simulation Run Procedure

Each simulation run controls its own simulation environment that models the modules' behavior, controls the time and also the signal state recorders. Each simulation environment needs a specific setup to function properly. Due to this, a simulation run consists of three phases, namely setup, register and run. It starts with a simulation environment at tick zero, that contains no modules and by this also no signal state recorders. During the setup phase variables might be declared, etc. but most importantly the modules that are tested are added to the simulation environment. This phase is ended by sending the start command to the simulation environment. Then the register phase follows, in which signals are added to the listening list and through this, signal state recorders are implicitly added to the environment. To have complete traces, this is recommended to be done before the simulation environment has advanced time by taking a simulation step. However, this phase can be left out when no traces are needed. Finally, during the run phase inputs are manipulated, output values are verified and the simulation time is advanced.

3.2.2 Simulation Environment

The simulation environment controls the calculation of the signal values, the advancing of time and the recording of signal states. When the simulation time should advance by one tick time, a step command has to be issued. As the simulation environment takes a step, time is advanced by one tick and the new signal states are calculated purely based on the previous states. The simulation environment's signal state recorders sample only the old signal states, since the current states can still be changed by the further executed simulation code.

[2]This should be analogous to the test driven development philosophy, but there is no limitation in any way preventing a different use.

3.2.3 Results

To analyze the output of test simulations it is best to use assertions. When an assertion expression fails, the specific simulation run is aborted and either the line wise position of the assertion and its resulting value or a customized error message are printed. If no error occurs, the simulation runs are usually executed without any output. This of course is useful for regression testing, but can be tedious for development and further analysis.

For that purpose the simulation interpreter in Dhdl is also capable of printing the recorded signal values in a semicolon separated style, which can be further processed by other programs. There is also an internal waveform viewer available.

3.2.4 Language Features

For convenient simulation run description, some features also common to other imperative languages are available. In this section are the detailed descriptions for statements that differ to some extent from the general concepts.

Assigning Values

Contrary to the hardware language, when assigning values to signal arrays or Boolean variable arrays it is possible to use values within $\mathcal{N}$. This value is then automatically converted to a Binary number representation and stored right aligned in the Boolean array. Leading zeros are added. If the destination array is too small, an error is printed and the current simulation run aborted.

Assertions

An assertion can contain a whole list of comma separated expressions. For an assertion to evaluate to true, all subexpressions have to evaluate to true, otherwise its result is false. The last expression can be followed by a comma and a print command for a specific error output which will be executed in case the assertion fails. If no print is specified, simply the position and value of the failed assertion is printed. When an assertion evaluation fails, the current simulation run is aborted.

Loops

The simulation language supports all kinds of loops (while, do-while, do-until, for, for-each), where for and for-each have a Dhdl specific meaning. The for-each loop is already described in Section 3.1.4 and can be used in the simulation code just the same way. The for loop has a variable definition over a range in its head and executes the body accordingly. Listing 3.6 shows an example where the body of the loop is executed 10 times and variable i is iterating from 1 to 10. With the keyword `downto` it is possible to reverse the direction.

Listing 3.6: For loop example.

```
for i := 1 to 10 {
        cell[i].in <= in[i]
}
```

3.3 Package Definition

Neither the hardware definition, nor the simulation language provide a convenient way of describing how a final module is connected to pins of a chip and which target technology to use. Therefore, a configuration file consisting of assignments is used for the package definition.

CHAPTER 4

Dhdl Implementation and Interaction

As the previous chapter defined all the concepts for the languages, this one is devoted to the Dhdl translation and simulation programs' implementation and its interaction concept and implementation with focus on the development flow used to derive the net lists and the resulting hardware from the design and simulation specifications in Dhdl.

Dhdl has been developed according to the guidelines of test driven development as specified in [2].

4.1 Technology Decisions

The Dhdl program is written in Java 6 [16]. Java has the advantage of built-in garbage collection, support for Windows and Linux and provides rich libraries. Furthermore, there exist tools for automated testing, on the fly code compilation and lexer and parser generation. For testing, JUnit 4.8.2 [13] and Easymock 3.0 [23] are used, as they are widely spread within the test driven development community. Antlr 3.2 [19] was chosen to generate lexers, parsers and tree walkers, as it provides a comfortable language for definition (see [17] and [18]) and is still actively developed. With Javassist 3.13.GA[4] Java code is compiled on the fly. As external synthesis tool, Quartus II 9.1sp1 Web Edition from Altera [5] was chosen, since it fits to the Altera DE2 Development and Education Board, on which the generated hardware was tested.

4.2 The Dhdl Development Flow

The Dhdl development flow is a result of the used tools and concepts defined prior. All interaction with the Dhdl program happens along this development. During the development of a Dhdl design, the complete flow chart given in Figure 4.1 is covered. All rounded rectangles represent tasks performed by a program, the circles yes/no feedback, and the other rectangles are files. The input for the Dhdl program are hardware, simulation and package description when all possible tasks should be performed. With the help of an external synthesis tool a netlist is generated

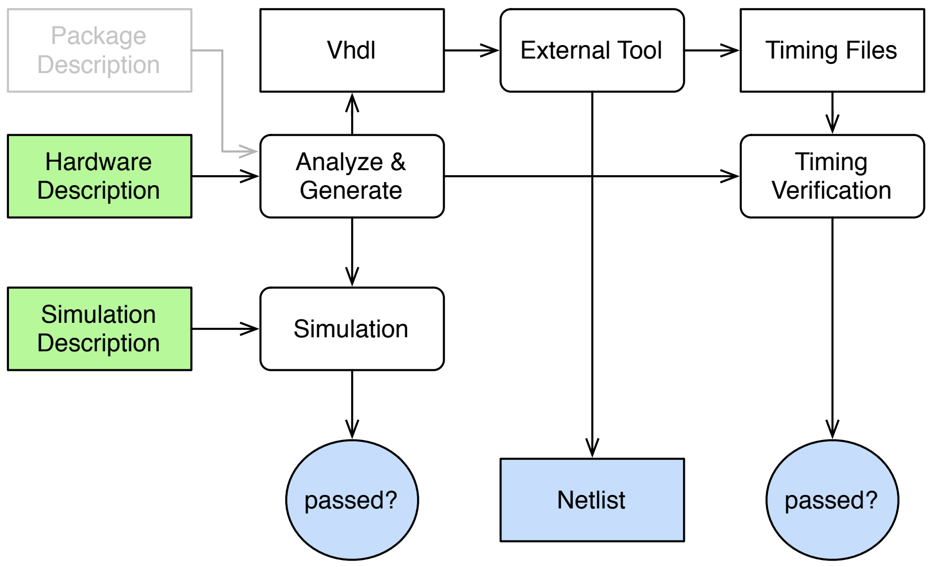

Figure 4.1: The development flow of a hardware design from specification to timing verification.

and Dhdl reports whether the simulations ran correctly and the timing requirements are met. For simulation runs, the hardware and simulation description are enough to yield the result, whereas for the timing verification also the package description is needed since the timing information of the full netlist is processed for accurate results. In the following the different tasks of the flow chart are explained in detail.

4.2.1 Analyze

Each design contains a single module as top level entity. Module identifiers are either directly passed to the Dhdl program or are retrieved from a package or simulation description. During the analysis of the hardware description, each module is parsed and stored to an internal data structure. Then the compliance with its parent modules and implemented interfaces is checked. Also, the assertions are evaluated. The wiring code is then executed to obtain the full module. All expressions are optimized and type checked and it is assured that all output signals of each module are assigned expressions.

4.2.2 Generate

Depending on the desired result either VHDL code is generated from the internal module representation or Java code. The Java code specifies classes extending a module simulation class. These classes are compiled on the fly and used for simulation. If a package file is given, a tcl-script for the external tool can be generated.

4.2.3 Simulation

For executing a simulation run the simulation run description is parsed and interpreted sequentially. Every time a module is added to the simulation environment it is analyzed and when starting the simulator all simulation objects are generated. When all commands of a simulation have been successfully executed, the simulation has passed.

4.2.4 External Tool

If the external tool for synthesis and timing analysis is started by the Dhdl program, its execution is controlled with the tcl-script generated from the package description. After the external tool has terminated, Dhdl can start the timing verification.

4.2.5 Timing Verification

For timing verification, the timing files are parsed and their signal network with timing information is constructed. Then all the delay assertions of the modules are evaluated. To evaluate a delay between signals, these signals are looked up in the network and then the minimum and maximum delay value of the paths in between them is determined with a depth first search algorithm.

4.3 Specific Issues

Most parts of the program are written in a straightforward fashion as it should be. This section is devoted to the parts where this method was not sufficient enough and it is worth to take a closer look at the solution. Also, in instances where a different approach seemed promising.

4.3.1 Network Representation

The signal delay extraction is the core feature for timing verification. As external synthesis and time analysis tool, Quartus is used and due to this some parts of building up the timed network depend specifically on the output formats of Quartus. The data structure for the internal network representation is a map from strings to lists of connections, where a connection contains the connected signals, the delay between them and the hierarchy level within the module structure. The index string is the origin signal of the connection. Another map called entry map consisting of the same data structure, but with zero delays within the connections, is used to connect the input and output ports of modules to the network and by this translates the names of the Dhdl naming system to the ones used by the Quartus output.

Parsing the Timed Network

Quartus generates two files for timing analysis, i.e. the VHDL file containing the network for the final design using the suffix vho and a standard delay format file (SDF file, see [12]) with the delays using the suffix sdo. Since Quartus names the signals with their full path according to the

VHDL component structure, it is possible to deduct the according modules and signals within Dhdl.

First, the network is derived by parsing the vho file. The containing cells with port mapping are extracted and the cell names are translated to the Dhdl names and put into the network with the according module level and zero delays. The translated cell name with an added space and either “in” or “out” and a list of the full input and output names and zero delays is then added to the entry map described earlier in “Network Representation”.

Then the delay file is parsed and for each specified delay the associated connection is found within the network map and its delay is set. The delays used in Dhdl only contain the minimum and maximum value and so during the extraction of delays all but those two are ignored.

The VHDL lexer and parser only implement the VHDL subset used within the generated files. For parsing the SDF file, a full lexer complying with the standard specified in [12] has been implemented, but again the parser recognizes only the constructs used by Quartus. In both cases, a full implementation according to the standards would have increased the programs’ size, complexity and vulnerability to bugs without any payoff.

Searching the Delays

A simple DFS algorithm is used to determine the delay in between signals. From the entry map, described earlier in “Network Representation”, the list of signal input ports is taken for the starting signal and the list of output ports for the destination signal. Then the DFS algorithm is run for each signal from the starting list. A signal is not further analyzed when the current signal is within the destination list or the level of the current signal is lower than the module containing the delay assertion. In the first case, a zero delay is returned and in the second, that no path was found. This is valid, since delay assertions can only refer to signals of their module and its submodules. This property is deducted naturally from the concept of a module enclosing complexity and providing an interface. Also through the language definition, by having no operator to refer to a higher module in the architecture or providing a special reference name for the enclosing module. When all follow up signals have been analyzed the merge of all further signal delays is returned. However, all connections to signals within the assertions’ module and its submodules have to be analyzed, since the minimum and maximum delay can occur within completely different paths.

4.3.2 Hardware Description Parsing

The parsing of hardware description files happens in the following order. First, all files are syntactically parsed and an abstract syntax tree is constructed. Each file represents its own scope. From these trees the information is passed to the objects representing replacement terms (described in Section 3.1.4), modules and interfaces. Then, the parameters of the modules and interfaces are interpreted and the port’s expressions are optimized in a first step. Then, the modules and interfaces are type checked and the assertions are evaluated. After this has been done for all modules and interfaces, the internal signals of the modules are optimized and all assigned expressions are type checked and further optimized. Finally, the wiring code is interpreted, the delay assertions are optimized and it is checked whether all expressions that are needed are assigned.

4.3.3 Expression Optimization

Expressions in Dhdl are optimized from the leaves to the roots where every subexpression is checked whether it is replaceable by a single node or a smaller expression tree and, if so, the replacement is performed. Through this constants are replaced by numbers, numbers are merged, double negation is removed, Boolean expressions where one parameter is a constant are compressed and clauses and unreachable parts of guarded expressions removed. Assertions in modules are only evaluated with this expression optimization.

4.3.4 Expression Parsing with Operator Precedence

The recommended method for parsing expressions in Antlr with operators of different precedence order is by using Antlr's backtracking feature to find a parse tree matching the expression grammar (see [17, pages 266 to 268]). This works only for expressions allowing a small set of possible operators within them.

However, Dhdl has and needs a rich set of operators, not only to construct guarded expressions. To overcome this problem, expressions are parsed in two steps. First, each expression is parsed and transformed into a list of node elements, where each node represents exactly one syntax element. The nodes are assigned values according to their precedence level: 0 for a final element, 1 for all unary operators, 2 for multiply, modulo and divide, etc. The recognition of unary and binary minus is done by Antlr according to the specified expression grammar. Then, these nodes are sequentially processed and formed into a tree that conforms to the specified operator precedence. This tree is constructed starting from a sentinel node with the highest precedence level as root. A pointer to the last inserted node is kept and initially points to the sentinel. Then, the expression list is proceeded sequentially. Final elements and unary operators are simply added to the pointer as child nodes. Otherwise, the pointer moves upwards in the tree as long as the inserted node has a higher precedence value than the pointer. Then, the last child of the pointer is added to the new node as a child and the pointer swaps its last child with the new node. Afterwards, the sentinel root node of the tree is found by searching upwards from the pointer and this tree is then used as AST for the expression. When evaluating an expression tree that is generated with this algorithm, operators on the same precedence level are evaluated from right to left. Figure 4.2 shows an example of a tree construction from an expression according to this algorithm.

4.3.5 Simulator Generation

The Dhdl simulation environment in Java consists of a simulator class, a class for the signal state recorders, a class for the simulated signals and one for the registers, as well as one class for each used parameter setting for a module. These modules are all derived from one module simulation class. This high amount of classes is necessary since the representation of the modules is translated to Java code and compiled on the fly. The reason for this is speed. Clearly, a compiled

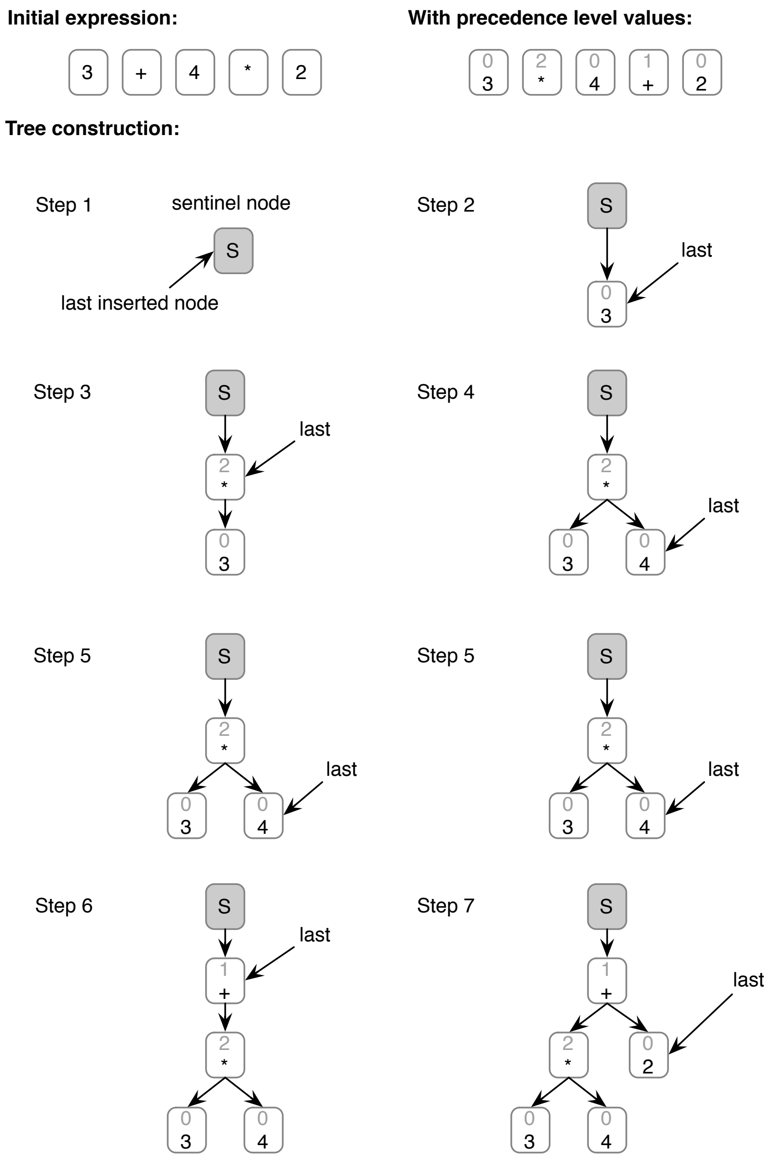

Figure 4.2: The construction of an expression tree with precedence levels.

Java class is faster during execution than an interpreter written in Java and the most frequent task during simulation is the calculation of signal values as time advances. Thus, optimizing this step first is a promising approach to generating a fast simulator. In this context, the terms module, signal and register now refer to their simulation classes.

The calculation of the new signal values after a tick has passed is implemented as follows. The simulator controls this process with two functions, i.e. calcTick and switchTick. A signal or register, respectively, has the functions getState, setState and switchTick. Only after calling switchTick, a value that has been set with setState will be read with getState. Therefore, calcTick of the first module calls all submodules' calcTick methods and then calculates the new signal values and sets them with setState. Similarly, switchTick calls all switchTick methods of the submodules and then those of its own signals and registers.

The generated modules, as mentioned before, derive themselves from the module simulation class. This class also has the functions initLocal and calcTickLocal. Code for those two functions is generated from the module description and they are replaced on the fly. Also, new members are added for the submodules, and local signals and registers. The initLocal function contains all the code for initializing and adding the submodules and connecting signals. In calcTickLocal, the code for calculating the outcome of the module's assignment expressions is performed. Extending and compiling classes on the fly is implemented with the use of Javassist (see [4]). For fast calculation of values from the wire type $\mathcal{W}$, a class containing two dimensional arrays is used, just like the one for symbol checking in [18, page 214].

CHAPTER 5

Semantics

In this chapter, the deterministic execution model of the Dhdl simulator is specified with a formal semantics. The resulting configurations of the semantic interpretation are then mapped to executions of the formal modeling and analysis framework of [9].

The specified semantics cover the core features of Dhdl and the formal modeling and analysis framework. So, for the hardware definition it includes modules with ports, signals and submodules. The available expression will be of logical and guarded type, but with the constraint that objects with guarded expressions must be defined in a module after all others have been defined. Furthermore, it is necessary that within an expression for a signal, this very signal is not assigned an expression within a side effect.[1] Also, the syntax of the simulation code is limited to the basic commands.

The definitions and methods in this chapter closely relate to those used in [10], since it is a well known and widely accepted approach. In its core, the semantic interpretation models the execution of the simulation code. It reflects the execution of simulation code with a structural operational semantics in order to exhibit the behavior of the simulation interpreter.

5.1 Basic Definitions

In the semantic interpretation, the Dhdl definition files are viewed as describing a transition function from one state of the signals in a module to another state, after one tick of time in the causal time model has passed (see Chapter 2). The simulator code is the starting point for the semantics execution. The structural operational semantics defined for Dhdl operate on the following construct:

$$\langle (S, D, T), (s, e, h) \rangle \tag{5.1}$$

Where,

[1]This is also not recommended in general Dhdl.

- S is the string for simulation code,
- D the string for definition code,
- T the transition function as a mapped set,
- s the signal state as a mapped set,
- e the environment as a mapped set and
- h the history of the recored states as a mapped set.

A semantic interpretation starts at the initial configuration and rules are applied for as long as is possible. When no further rule can be applied, the semantic interpretation is terminated and its result is the last configuration.

In the following all types used in the construct above are explained in detail:

Strings Strings are, when necessary, annotated with single quotes and their concatenation with a plus sign, e.g. 'abc' $+$ 'd' $=$ 'abcd'. The empty string will be represented with ϵ and newlines with '\n', like in C. Sometimes, the single quotes are left out for better readability.

Within this semantic rules and algorithms, strings and their substrings can also be referred to as abstract syntax trees (see [15, page 70]). The referring name is then identical with a syntax definition name and thus the string has to map the syntax rule. An explicit model of these trees would cause the rules and algorithms to be more difficult to read without providing more insight.

Command Identification When referring to the first command within the code of S or D, this is written as the command followed by a colon and then a symbol for the remaining code. E.g. $D =$ Signal $a : D'$ means that D starts with the definition of member a from the type Signal. Since there are no semicolons used in Dhdl to determine the end of a command, the colon is used to represent it. This colon style can also be used to select another symbol within the code, e.g. $D =$ module $name$ {: D' :} : D'', where the closing brace is the one corresponding to the first opening bracket.

Mapped Sets Mapped sets are sets of tuples of the form $a \mapsto b$, where the left string a of the tuple maps to the right string b. These mapped sets are annotated with mapping arrows $\mapsto$ within the comma separated tuples, e.g. $M = \{a \mapsto b, c.e \mapsto d\}$. The left string of the tuple is also called symbol and the right one, element, for the purpose of clear distinction. The element can be accessed with the the symbol and dot operator, e.g. $M.a$. If the symbol contains dots, also brackets can be added. They are not required, but emphasize the symbol string, e.g. $M.(c.e) = M.c.e = M.'c.e'$. Only symbols fulfill the $\in$ property, e.g. $a \in M$ and $b \notin M$. Furthermore, the [ ] operator adds or “substitutes” symbols, e.g. $M' = M[e \mapsto a][a \mapsto f] = \{a \mapsto f, c.e \mapsto d, e \mapsto a\}$. When concatenated, these operators are executed in the order of appearance, starting with the left most one. For readability, it is possible to have more expressions set at the same time, e.g. $M[a \mapsto b][c \mapsto d] = M[a \mapsto b, c \mapsto d]$. If the accessed symbol is not part of the set, the expression is equal to ϵ, e.g. $\emptyset.b = \epsilon$.

5.2 Syntax

For semantic analysis of a language a full syntax definition is necessary. The full EBNF syntax of the supported Dhdl language subset is given in Listing 5.1.

Listing 5.1: Syntax definition expressions modeled by the formal semantics.

```
sig_expr ::=
        expr Newline |
        g_expr+

g_expr ::=
        clause_list '->' side_effects? expr Newline

w_expr ::=
        (clause_list '->' expr)+

c_list ::= (expr (',' expr)*)?

side_effects ::= (variable '<=' expr ',')+

expr :=
        un_op expr |
        expr bin_op expr |
        '(' expr ')' |
        value |
        variable

assert_expr :=
        un_op expr |
        expr bin_op expr |
        '(' expr ')' |
        value |
        sim_signal

un_op ::= 'not' | '!'

bin_op ::=
        'and' | 'nand' | 'or' | 'nor' | 'xor' |
        'equiv' | '==' | '!='

value ::= 'true' | 'false'

variable ::=
        Identifier ('.' Identifier)?

sub_port ::=
        Identifier '.' Identifier
```

```
sim_signal ::=
        Identifier ('.' Identifier)*

Identifier ::=
        [a-zA-Z] ('_' Identifier_character |
                Identifier_character)*

Identifier_character ::= [a-zA-Z0-9]

Newline ::=
        (('\r')? '\n')

module ::=
        'module' '{'
                module_element*
        '}'

module_element ::=
        'in' 'Signal' Identifier Newline |
        'out' 'Signal' Identifier '<=' sig_expr |
        'Signal' Identifier '<=' sig_expr |
        ':' sub_port '<=' sig_expr |
        Identifier Identifier Newline

simulation ::=
        (simulation_command Newline)*

simulation_command ::=
        'simulator' 'add' Identifier ':=' Identifier |
        'simulator' 'start' |
        'simulator' 'listen' sim_signal |
        'simulator' 'step'
        sim_signal ':=' expr
        'assert' assert_expr
```

Initially, passed strings to D have to match the $module$ rule and those for S have to match $simulation$.

5.3 $\mathcal{W}$ Semantics

$\mathcal{W}$ arithmetic is used for calculating values of signals by semantic interpretation of expression code. From the rules shown below, rule 1 is applicable for empty clause lists, which has the truth value true, just like an empty formula. Rules 2 to 4 just map the strings to the objects in $\mathcal{W}$. Rule 5 is for accessing the values of signals. Rules 6 to 12 model logic operations. Rule 13 is for clause lists and rules 14 and 15 for guarded expressions.

1. $\mathcal{W}[]s = \text{true}$

2. $\mathcal{W}[\text{true}]s = \text{true}$

3. $\mathcal{W}[\text{false}]s = \text{false}$

4. $\mathcal{W}[\text{undef}]s = \text{undef}$

5. $\mathcal{W}[sim\_signal]s = \mathcal{W}[s.sim\_signal]$

6. $\mathcal{W}[\text{not}\ a]s = \mathcal{W}[!\ a]s = \begin{cases} \text{true} & \text{if } \mathcal{W}[a]s = \text{false} \\ \text{false} & \text{if } \mathcal{W}[a]s = \text{true} \\ \text{undef} & \text{if } \mathcal{W}[a]s = \text{undef} \end{cases}$

7. $\mathcal{W}[a_1\ \text{and}\ a_2]s = \begin{cases} \text{true} & \text{if } \mathcal{W}[a_1]s = \mathcal{W}[a_2]s = \text{true} \\ \text{false} & \text{if } \mathcal{W}[a_1]s = \text{false} \vee \mathcal{W}[a_2]s = \text{false} \\ \text{undef} & \text{if otherwise} \end{cases}$

8. $\mathcal{W}[a_1\ \text{nand}\ a_2]s = \begin{cases} \text{true} & \text{if } \mathcal{W}[a_1]s = \text{false} \vee \mathcal{W}[a_2]s = \text{false} \\ \text{false} & \text{if } \mathcal{W}[a_1]s = \mathcal{W}[a_2]s = \text{true} \\ \text{undef} & \text{if otherwise} \end{cases}$

9. $\mathcal{W}[a_1\ \text{or}\ a_2]s = \begin{cases} \text{true} & \text{if } \mathcal{W}[a_1]s = \text{true} \vee \mathcal{W}[a_2]s = \text{true} \\ \text{false} & \text{if } \mathcal{W}[a_1]s = \mathcal{W}[a_2]s = \text{false} \\ \text{undef} & \text{if otherwise} \end{cases}$

10. $\mathcal{W}[a_1\ \text{nor}\ a_2]s = \begin{cases} \text{true} & \text{if } \mathcal{W}[a_1]s = \mathcal{W}[a_2]s = \text{false} \\ \text{false} & \text{if } \mathcal{W}[a_1]s = \text{true} \vee \mathcal{W}[a_2]s = \text{true} \\ \text{undef} & \text{if otherwise} \end{cases}$

11. $\mathcal{W}[a_1\ \text{xor}\ a_2]s =$

$$\mathcal{W}[a_1 \,!= a_2]s = \begin{cases} \text{true} & \text{if } \mathcal{W}[a_1]s \neq \mathcal{W}[a_2]s \wedge \mathcal{W}[a_1] \neq \text{undef} \wedge \mathcal{W}[a_1] \neq \text{undef} \\ \text{false} & \text{if } \mathcal{W}[a_1]s = \mathcal{W}[a_2]s \neq \text{undef} \\ \text{undef} & \text{if otherwise} \end{cases}$$

12. $\mathcal{W}[a_1\ \text{equiv}\ a_2]s =$

$$\mathcal{W}[a_1 == a_2]s = \begin{cases} \text{true} & \text{if } \mathcal{W}[a_1]s = \mathcal{W}[a_2]s \neq \text{undef} \\ \text{false} & \text{if } \mathcal{W}[a_1]s \neq \mathcal{W}[a_2]s \wedge \mathcal{W}[a_1]s \neq \text{undef} \wedge \mathcal{W}[a_1] \neq \text{undef} \\ \text{undef} & \text{if otherwise} \end{cases}$$

13. $\mathcal{W}[a_1, a_2]s = \begin{cases} \text{true} & \text{if } \mathcal{W}[a_1]s = \text{true} \wedge \mathcal{W}[a_2]s = \text{true} \\ \text{false} & \text{if } \mathcal{W}[a_1]s = \text{false} \vee \mathcal{W}[a_2]s = \text{false} \\ \text{undef} & \text{if otherwise} \end{cases}$

14. $\mathcal{W}[clause\_list \text{ -> } expr]s$

$$= \begin{cases} \text{true} & \text{if } \mathcal{W}[clause\_list]s = \text{true} \wedge \mathcal{W}[expr]s = \text{true} \\ \text{false} & \text{if } \mathcal{W}[clause\_list]s = \text{true} \wedge \mathcal{W}[expr]s = \text{false} \\ \text{undef} & \text{if otherwise} \end{cases}$$

15. $\mathcal{W}[clause\_list$ -> $expr\ w\_expr]s$

$$= \begin{cases} \text{true} & \text{if } \mathcal{W}[clause\_list]s = \text{true} \wedge \mathcal{W}[expr]s = \text{true} \\ \text{false} & \text{if } \mathcal{W}[clause\_list]s = \text{true} \wedge \mathcal{W}[expr]s = \text{false} \\ \text{true} & \text{if } \mathcal{W}[clause\_list]s = \text{false} \wedge cW[w\_expr]s = \text{true} \\ \text{false} & \text{if } \mathcal{W}[clause\_list]s = \text{false} \wedge cW[w\_expr]s = \text{false} \\ \text{undef} & \text{if otherwise} \end{cases}$$

5.4 Rules

Since the causal time model is the basis for hardware simulation in Dhdl, structural operational semantics are used for interpretation. These pronounce especially well the tick-wise transition from state to state as time advances. The semantic rules given here can be divided into two classes according to their functionality. System definition rules translate definition code D into transition code for the mapped set T. This transition code is a condensed version of the definition code and interpretable by $\mathcal{W}$ semantics. Simulation rules directly interpret the simulation code S. When a new module is added to the system, all system definition rules must be executed before the simulation code can be further interpreted. This is achieved by requiring that $D = \emptyset$ for each simulation rule. Only a small set of rules is defined that still supports the specified subset of Dhdl in order to keep the semantic interpretation simple.

Rules have one of the following two shapes:

1. $A \Rightarrow B$
2. $\dfrac{C \Rightarrow^* D}{A \Rightarrow B}$

With rules of the first shape, it is possible to directly reach the configuration B by applying the rule to A. For the second type, the configuration B can only be reached, if and only if the configuration D can be reached by applying one or more rules on the configuration C.

In the following context, $type$ and $name$ have the syntax of $Identifier$ and, although the cursive names are included within strings, they represent syntactical constructs. This is done to improve the readability of where a definition within the code ends and the next one begins.

5.4.1 System Definition Rules

A hardware design in Dhdl is characterized by its modules, signals and their expressions. These rules are used to handle the hardware definition code D.

Module Definition

[module]

$$\frac{\langle (S, D', T), (s, e, h)\rangle \Rightarrow^* \langle (S, \epsilon, T'), (s', e', h)\rangle}{\langle (S, \text{'module } name \text{ \{' : } D' \text{ : '\}'}, T), (s, e, h)\rangle \Rightarrow \langle (S, \epsilon, T'), (s', e, h)\rangle}$$

Interpreting a module definition. This rule forces the body of the module to be fully interpreted and all of its definitions transferred to the transition set.

[submodule]

$$\frac{\langle (S, e.\mathit{type}, T), (s, e', h)\rangle \Rightarrow^* \langle (S, \epsilon, T''), (s'', e'', h)\rangle}{\langle (S, \text{'}\mathit{type}\ \mathit{name}\text{'} : D, T), (s, e, h)\rangle \Rightarrow \langle (S, D, T''), (s'', e'', h)\rangle}$$

- precondition: $\mathit{type} \in e$
- $e' = e[e.\text{mod\_curr} \mapsto e.\text{mod\_curr} + \text{'.'} + \mathit{name}]$

Including a submodule within a module. The nested path for the submodule is kept in mod_curr of the environment set e. The module definition of the submodule is retrieved from e and fully interpreted. No check for modules with the same name is performed, but the sets of signals within the modules have to be disjoint. A direct check would increase the semantics' complexity without providing an adequate advantage.

Signal Definition

[input]

$$\langle (S, \text{'in Signal } \mathit{name}\text{'} : D, T), (s, e, h)\rangle \Rightarrow \langle (S, D, T), (s', e, h)\rangle$$

- precondition: $e.\text{mod\_curr} + \text{'.'} + \mathit{name} \notin s$
- $s' = s[e.\text{mod\_curr} + \text{'.'} + \mathit{name} \mapsto \text{undef}]$

Defining an input port. The input port must not be defined elsewhere and naturally, the defining module cannot set its expression.

[output]

$$\langle (S, \text{'out Signal } \mathit{name} \text{ <= } \mathit{sig\_expr}\text{'} : D, T), (s, e, h)\rangle \Rightarrow \langle (S, D, T'), (s', e, h)\rangle$$

- precondition: $e.\text{mod\_curr} + \text{'.'} + \mathit{name} \notin s$
- $s' = s[e.\text{mod\_curr} + \text{'.'} + \mathit{name} \mapsto \text{undef}]$
- $T' = \text{addexpression}(T, e.\text{mod\_curr}, \mathit{name}, \mathit{expr})$

Defining an output port. The output port must not be defined elsewhere and its expression must be assigned at definition.

[signal]

$$\langle (S, \text{'Signal } \mathit{name} \text{ <= } \mathit{sig\_expr}\text{'} : D, T), (s, e, h)\rangle \Rightarrow \langle (S, D, T'), (s', e, h)\rangle$$

- precondition: $e.\text{mod\_curr} + \text{'.'} + \mathit{name} \notin s$
- $s' = s[e.\text{mod\_curr} + \text{'.'} + \mathit{name} \mapsto \text{undef}]$
- $T' = \text{addexpression}(T, e.\text{mod\_curr}, \mathit{name}, \mathit{sig\_expr})$

Defining a signal. The signal must not be defined elsewhere and its expression must be assigned at definition. This definition might lead to a pitfall, as no visibility checks are modeled within the semantics to keep them simple. Thus, these signals could be accessed like output ports.

[wire]

$$\langle (S, \text{': } \mathit{sub\_port} \text{ <= } \mathit{sig\_expr}\text{'} : D, T), (s, e, h)\rangle \Rightarrow \langle (S, D, T'), (s, e, h)\rangle$$

- preconditions: $e.\text{mod\_curr} + \text{'.'} + \mathit{sub\_port} \in s$, $e.\text{mod\_curr} + \text{'.'} + \mathit{sub\_port} \notin T$
- $T' = \text{addexpression}(T, e.\text{mod\_curr}, \mathit{variable}, \mathit{sig\_expr})$

Assigning a submodule's input port. The port to be assigned must exist and it must not have been assigned so far. Furthermore, the syntax requires that it is a port of a submodule and must therefore be an input port, since all others have to be assigned on definition.

5.4.2 Simulation Rules

Simulation rules define the semantics for code within S.

Adding Modules

[add]

$$\langle (\text{'simulator add } \mathit{name} \text{ := } \mathit{type}\text{'} : S, \epsilon, T), (s, e, h)\rangle \Rightarrow \langle (S, e.\mathit{type}, T), (s, e', h)\rangle$$

- preconditions: $\mathit{type} \in e$, $e.\mathit{sim\_stat} = \text{init}$
- $e' = e[\text{mod\_curr} \mapsto \mathit{name}]$

Modules can only be added before the simulator is started.

Starting the Simulator

[start]

$$\langle(\text{'simulator start'} : S, \epsilon, T), (s, e, h)\rangle \Rightarrow \langle(S, \epsilon, T), (s, e', h)\rangle$$

- preconditions: $type \in e$, $e.sim\_stat = \text{init}$
- $e' = e[\text{sim\_stat} \mapsto \text{listen}]$

Simply prevents the adding of more modules.

Listen to a Signal

[listen]

$$\langle(\text{'simulator listen } sim\_signal : S, \epsilon, T), (s, e, h)\rangle \Rightarrow \langle(S, \epsilon, T), (s, e, h')\rangle$$

- preconditions: $e.sim\_stat = \text{listen}$, $sim\_signa \in s$, $sim\_signa \notin h$
- $h' = h[signal \mapsto \epsilon]$

Enlists a signal to be recorded in the state history.

Execute Simulator Steps

[step]

$$\langle(\text{'simulator step'} : S, \epsilon, T), (s, e, h)\rangle \Rightarrow \langle(S, \epsilon, T), (s', e', h')\rangle$$

- preconditions: $e.sim\_stat \in \{\text{listen}, \text{init}\}$
- $\forall signal \in s : s' = \mathcal{W}[T.signal]s$
- $h' = h(\forall signal \in s : [signal \mapsto h.signal + \text{':'} + s.signal])$
- $e' = e[\text{sim\_stat} \mapsto \text{run}]$

Computes the new state values with the $\mathcal{W}$ semantics and records the past states in the state history.

Set Value

[set]

$$\langle(\text{'}sim\_signal := sig\_expr\text{'} : S, \epsilon, T), (s, e, h)\rangle \Rightarrow \langle(S, \epsilon, T), (s', e, h)\rangle\rangle$$

- precondition: $sig\_expr \in s$
- $s' = s[name \mapsto \mathcal{W}[sig\_expr]s]$

Assigns the string to a signal.

Assert

[assert^t]

$$\langle(\text{'assert } assert\_expr\text{'} : S, \epsilon, T), (s, e, h)\rangle \Rightarrow \langle(S, \epsilon, T), (s, e, h)\rangle\rangle$$

- precondition: $\mathcal{W}[assert\_expr]s = \text{true}$

When the assertion holds, the semantic interpretation just continues.

[assert^f]

$$\langle(\text{'assert } assert\_expr\text{'} : S, \epsilon, T), (s, e, h)\rangle \Rightarrow \langle(\text{FAIL}, assert\_expr), (s, e, h)\rangle$$

- precondition: $\mathcal{W}[assert\_expr]s = \text{false}$

When the assertion fails, the semantic interpretation is terminated and a failure configuration is returned.

Terminate

[term]

$$\langle(\epsilon, \epsilon, T), (s, e, h)\rangle \Rightarrow \langle\text{TERM}, h'\rangle$$

- preconditions: $e.sim\_stat \in \{\text{listen}, \text{init}\}$
- $h' = h(\forall signal \in s : [signal \mapsto h.signal + \text{':'} + s.signal])$

When no further simulation commands can be executed, the semantic interpretation is terminated and the state history returned.

5.4.3 Initial Configuration

The initial configuration of a system $\langle(S_i, D_i, T_i), (s_i, e_i, h_i)\rangle$ is defined as follows:

- S_i is the simulation code that will be executed.
- $D_i = \epsilon$
- $T_i = \emptyset$
- $s_i = \emptyset$
- $e_i = \{\text{sim\_stat} \mapsto \text{init}, \text{mod\_curr} \mapsto \epsilon\} \cup rms$, where rms represents all the required module definitions in the form of $module\_name \mapsto module\_code$, where $module\_code$ matches the syntax of $module$. The intention of sim_stat is to determine the current status of the simulator phase.
- $h_i = \emptyset$

5.4.4 Termination

After applying the termination rule, a configuration of the type $\langle \text{TERM}, h \rangle$ is reached, no further rule can be applied and the simulation terminated correctly. The result of the simulation is the history of the recorded signals h. Should the simulation end in $\langle (\text{FAIL}, a), (s, e, h) \rangle$, the assertion expression a was not valid. If during the semantic interpretation a different configuration is reached, where no rule is applicable, then the code has a syntax error or a non existent symbol is accessed.

5.4.5 Functions

The algorithms of the functions used to transform part of the definition code to the transition set T are written in a style close to regular programming languages and use the := operator for assigning a new value to a variable. These functions make extensive use of the possibility to refer to substrings in the style of abstract syntax trees.

add_expression

This function is specified in Algorithm 1. It adds a new expression for a signal to the mapped set T.

Parameters:

- T_c is the transition map before applying the rule as a mapped set.
- mod_curr is the current module's path and name as a string.
- $name$ is the name of the signal to be assigned an expression as a string.
- $expression$ is the expression to be added to the signal as a string.

Variables:

- T' is the transition map with side effects as a mapped set.
- $name'$ is the full name of the signal to assign the expression as a string.
- $expression'$ is the expression, without side effects, to be added for the signal as a string.
- $expression''$ is the expression, with full signal names and without side effects, to be added as a string.

Returns: A mapped set with the added expression.

Algorithm 1 add_expression $(T_c, mod\_curr, name, expression)$

```
1: $T' =$ add_sideeffects$(T, mod\_curr, expression)$
2: $name' =$ expand_signalnames$(mod\_curr, name)$
3: $expression' =$ remove_sideeffects$(expression)$
4: $expression'' =$ expand_signalnames$(mod\_curr, expression)$
5: **return** $T'[name' \mapsto expression'']$
```

add_sideeffects

This function is specified in Algorithm 2. It adds the side effects to the signal expressions of the mapped set for the transition function.

Parameters:

- T_c is the transition map before applying the rule as a mapped set.
- mod_curr is the current module's path and name as a string.
- $expression$ is the expression where the side effects are extracted from as a string.

Variables:

- T_r is the transition map for returning.
- A is an intermediate set for assembling the side effects functions as a mapped set.
- gl is the current guarded line of the expression as a string.
- $name$ is the name of the individual signals to assign the side effects as a string.
- $name'$ it the full name of the individual signals to assign the side effects as a string.
- $cexpr$ is the expression to be added for the signal as a string.
- $expr'$ is the expression to be added for the signal with full signal names as a string.
- $clause\_list$ is the clauses within the guarded line as a string.
- $clauses'$ is the clauses within the guarded line with full signal names as a string.
- $signal$ is the signal name to have its expression extended as a string.

Returns: A mapped set with the added side effect expressions.

Algorithm 2 add_sideeffects $(T_c, mod\_curr, expression)$

```
1: $T_r$:=$T_c$
2: **if** $expression$ is not of $guarded\ expression$ type **then**
3:     **return** $T_r$
4: **end if**
5: **for each** $guarded\_line$ in $expression$ as $gl$ **do**
6:     **for each** $side\_effect$ in $expression$ as $name$ <= $cexpr$ **do**
7:         $name'$:= expand_signalnames($mod\_curr, name$)
8:         $expr'$:= expand_signalnames($mod\_curr, cexpr$)
9:         $clauses'$:= expand_signalnames($mod\_curr, clause\_list$ of $gl$)
10:        $A$:=$A[name' \mapsto A.name' + clauses'$+'->'+$expr' + Newline]$
11:    **end for**
12: **end for**
13: **for each** element in $A$ as $signal \mapsto expr$ **do**
14:    $T_r$:=$T_r[signal \mapsto expr + T.signal]$
15: **end for**
16: **return** $T_r$
```

remove_sideeffects

This function is specified in Algorithm 3. It removes side effects from an expression.

Parameters:

- $expression$ is the expression where the side effects are removed from as a string.

Variables:

- $rexpr$ is the expression to be returned as a string.

Returns: A string with the cleaned expressions.

Algorithm 3 remove_sideeffects $(expression)$

```
1: **if** $expression$ is not of $guarded\ expression$ type **then**
2:     **return** $expression$
3: **end if**
4: **for each** $guarded\_line$ in $expression$ as $gl$ **do**
5:     $rexpr$:=$rexpr + clause\_list$ of $gl$+'->'+$expr$ of $gl + Newline$
6: **end for**
7: **return** $rexpr$
```

expand_signalnames

This function is specified in Algorithm 4. It replaces all variable occurrences with their full reference name in T.

Parameters:

- mod_curr is the current module's path and name as a string.
- $construct$ is the construct to be expanded as a string.

Variables:

- $rconstruct$ is the construct to be returned as a string.

Returns: A string with full signal names.

Algorithm 4 expand_signalnames $(construct)$

```
1: $rconstruct$:=$construct$
2: for all structures $s$ in $rconstruct$ of type $variable$ do
3:     substitute $s$ with $mod\_curr$+'.'+$s$ within $rconstruct$
4: end for
5: return $rconstruct$
```

5.4.6 Integration to the Formal Framework

The formal analysis and modeling framework of [9] defines three different signal representations. Closest to the history representation of the signals in the semantics is the status representation of the framework. It is defined by the function $\tilde{S}(t)$:

$$\tilde{S} : R_0^+ \to \{0,1\},$$

where each point of realtime is associated with a Boolean value. In this work, the codomain of $\tilde{S}$ was extended: Traces in the history of the Dhdl semantic can also contain undef. One can deduce from the definition of $\mathcal{W}$ and the fact that the undef cannot be assigned due to the syntax definition that, as soon as a state in the semantic execution without undef is reached, only true and false values occur in the remaining execution. Let T_0 be the smallest tick number such that a state without undef occurs, starting the enumeration with tick 0. Let l be the signal name of the signal S and h the history of a semantic execution. Then, for all $t \in R_0^+$, $\tilde{S}(t) = hmap(t + T_0, h.l)$ with:

$$hmap(t, s_h) = \begin{cases} 0 & \text{if } 0 \leq t < 1 \wedge s_h \text{ starts with 'false:'} \\ 1 & \text{if } 0 \leq t < 1 \wedge s_h \text{ starts with 'true:'} \\ hmap(t-1, s_f) & \text{where } s_f = s_h \text{ without the first value and colon} \end{cases}$$

By starting at the time when there occurs no undef anymore, it is possible to transform the traces, as introduced in this work, to valid traces of the formal framework.

CHAPTER 6

Reference Designs

The reference designs, presented in this chapter, show the capabilities of Dhdl and are also used to test the correctness of the translation and timing analysis. Another goal of these designs is to show how the features of Dhdl harmonize with each other.

6.1 All Aspects of a Small Design

Aside from the functionality check, deeper insight into Dhdl is another motivation for implementing and analyzing all translations and simulation traces of a small reference design.

6.1.1 Reference Design Specification

The criteria for a suitable reference design are as follows: First, it has to show the core features of the language, which are guarded expressions, causal time and timing assertions. Second, it has to be within the syntactical description of the specified formal semantics. Lastly, it has to be feasible for manual analysis of the generated VHDL code, obtained timings, the semantic interpretation and simulation. The output files of the synthesis tools and the semantic interpretation traces tend to be large compared to the initial designs, thus we decided to specify a design as small as possible. This example is represented in Listing 6.1. Apart from line two and three, it is within the syntax used by the formal semantics as specified in Chapter 5. It contains a guarded expression, the signal $last$ can be used to demonstrate the functionality of the causal time model and a delay assertion is also included. It is an implementation of an and-gate with two output lines holding eventually the same value. With this simple behavior it is possible to check the correct results for Dhdl simulation, VHDL generation, VHDL simulation, timing verification, semantic interpretation and the behavior on a FPGA board.

Listing 6.1: The small reference design.

```
1 module And {
2         delay time_assertion := in0ToLast > 10ps
```

```
        delay in0ToLast := (in0 , last)

        in Signal in0
        in Signal in1

        out Signal out <=
                in0 == true, in1 == true
                        -> true
                        -> false

        out Signal last <= out
}
```

6.1.2 Dhdl Simulation

Figure 6.1 shows the simulation trace of the Dhdl simulation that can be found in more detail in Appendix B.1.1. Initially, the signals *out* and *last* have the value undef, which is specified by the formal semantics in Chapter 5. The signal *out* behaves according to the and-gate truth table, with one tick delay, as specified by the causal time model. Also, *last* has the value of *out* with one tick delay.

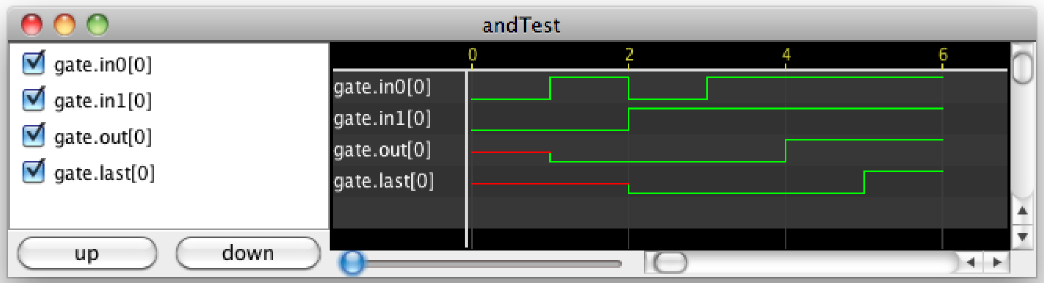

Figure 6.1: The trace of a the and-gate simulation.

6.1.3 Generated VHDL Code

From the Dhdl specification, the VHDL code in Listing 6.2 is generated according to the definitions in Appendix A.6.1. The ports from the original design can easily be identified, as well as the module itself. The processes "in_ports" and "out_ports" just map the ports to the internal signals, distinguishable by their suffixes "_port" and "_signal". The internal signals are necessary since, contrary to Dhdl, output ports can only be read, but not assigned in VHDL. The mapping of the input ports is performed for consistency and, through this, readability. The expressions assigned in Dhdl are translated to the process "calculate_signals". The expression for

$last$ is clearly equivalent to the assignment in Dhdl and the if-else structure for out reads like the guarded expression: If $in0$ and $in1$ are true, out is also true, otherwise out is false.

Listing 6.2: The VHDL code generated from the reference design.

```
-- generated by dhdl
-- 2011/07/19 18:37:14

library ieee;
use ieee.std_logic_1164.all;

entity And_0 is
        port (
                in0_port : in std_logic;
                in1_port : in std_logic;
                out_port : out std_logic;
                last_port : out std_logic
        );
end And_0;

architecture behavior of And_0 is

        signal in0_signal : std_logic;
        signal in1_signal : std_logic;
        signal out_signal : std_logic;
        signal last_signal : std_logic;

begin
        in_ports: process (in0_port, in1_port)
        begin
                in0_signal <= in0_port;
                in1_signal <= in1_port;
        end process;

        calculate_signals: process (in0_signal, in1_signal, out_signal,
            last_signal)
        begin
                last_signal <= out_signal;
                if (in0_signal = '1') and (in1_signal = '1') then
                        out_signal <= '1';
                else
                        out_signal <= '0';
                end if;
        end process;

        out_ports: process (out_signal, last_signal)
        begin
                out_port <= out_signal;
                last_port <= last_signal;
        end process;

end behavior;
```

6.1.4 VHDL Simulation

To keep the trace of the VHDL simulation similar to the one of the Dhdl code, the values of the input signals are assigned in the same order and one tick is resolved to a time of 1ns. The simulation, specified in Appendix B.1.2, is run for 6ns and the resulting waveform can be found in Figure 6.2. However, contrary to the Dhdl simulation, all values of the input ports instantly effect both output ports.

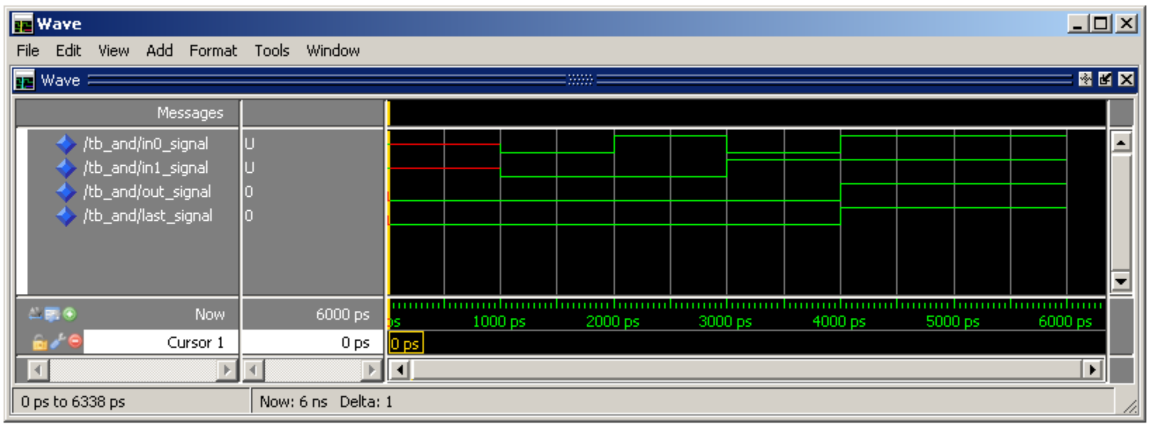

Figure 6.2: The trace of the and-gate VHDL testbench.

6.1.5 Timing Assertions

From the VHDL code the external tool generates timing files (see Appendix B.1.3). Listing 6.3 contains the output of the Dhdl timing analysis with inserted newlines for readability. The value of the $in0ToLast$ delay expression is the time it takes for a signal change at $in0$ to affect the output signal $last$. The timing assertion $time\_assertion$ then checks whether this delay is larger than 10ps. From the output we can tell that this assertion holds.

Listing 6.3: Dhdl timing analysis output of the and-gate.

```
Time Reader: parse time/And_vhd.sdo
Time Reader: parse time/And.vho
[line 4, char 19]: Delay in0ToLast := (in0[1], last[1]) on path "
  base component" is satisfied. [1.1303999999999998E-8,
  1.1303999999999998E-8]
dhdl/And.df[line 3, char 35]: Delay time_assertion := (in0ToLast >
  1.0E-11) on path "base component" is satisfied. [1.0E-11,
  1.1303999999999998E-8]
0 errors 0 warnings 0 assertion violations
```

From the timing files, the graph of Figure 6.3 can be derived by hand. In this graph, the direction of the edges is from top to bottom and the weights are either $0s$, or the given value. Following the lines, the resulting delay is 11304ps, which is almost the value returned by Dhdl found in Listing 6.3 at the end of line four. The difference comes from the floating point representation in Java. Since it is less than 10^{-23}s and so way beyond our measuring or designing precision, it should not be bothered. Interestingly, the delays between the and-gate output port and each of the output port pins on the FPGA are the same. Since these pins are right next to each other on the chip, the delay difference between them is too small to be modeled.

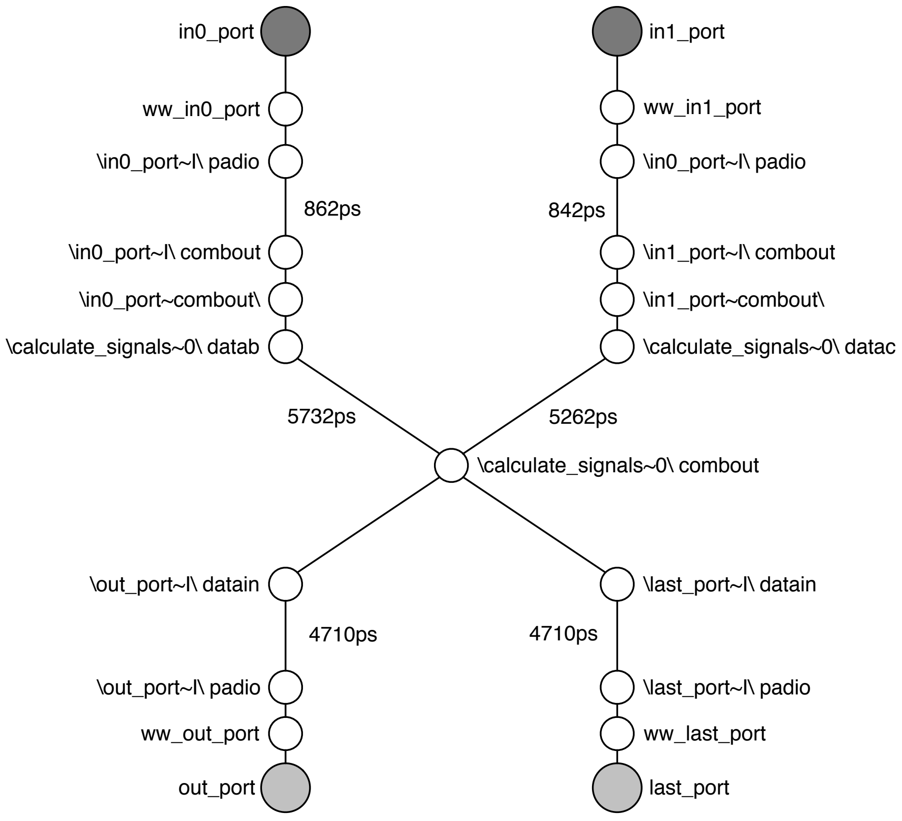

Figure 6.3: Manually derived graph from the timing files of the and-gate.

6.1.6 Testing on a Development Board

This simple and-gate design has been tested on the development board and it works as specified. Please note, when using the same test environment, the buttons on the Altera DE2 Development and Education Board are low-active.

6.1.7 Semantic Interpretation

The full semantic interpretation of the simulation code can be found in Appendix B.1.4. The history h of the final configuration $\langle \text{TERM}, h \rangle$ is equivalent to the simulation trace presented earlier in Figure 6.1 as required.

6.2 Synchronous Design

To demonstrate the capabilities of Dhdl with respect to synchronous designs, a simple coffee machine is implemented with a finite state machine. The machine starts in the idle state. Eventually, a customer inserts a coin and then presses one of three buttons: “make coffee”, “make tea” or “abort”. On “abort”, the coin is returned and the machine is set to the initial state. When “make coffee” or “make tea” is pressed, the right beverage is made and when it is ready, a beep tone is played until the cup is taken out of the machine.

The Dhdl code can be found in Appendix B.2. Figure 6.4 shows the state machine diagram for this design, annotating the active signals. The state machine coincides with the specification for the coffee machine.

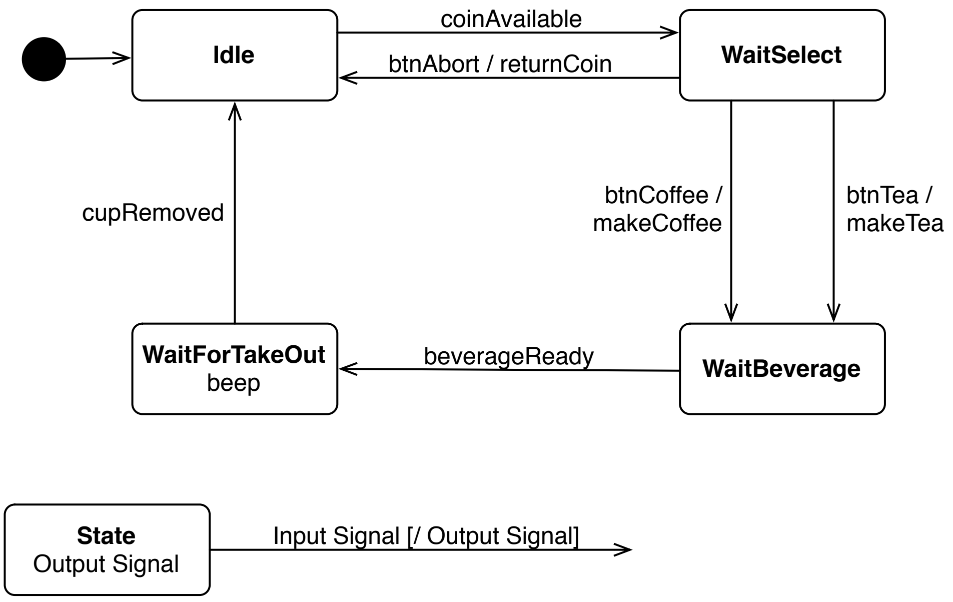

Figure 6.4: State machine diagram of the coffee machine design.

The Dhdl hardware definition (see Listing B.5) is equally simple to describe. With the use of a guarded expression for defining the state transitions, this description is straightforward. Aside from that, it is required that the current state is stored in a register ($state$) and that all output ports are set to false as default. The expression for $state$ is organized as follows. The first six guarding expressions are derived from the state transitions in the diagram and their side conditions set the according output ports. The subexpression that checks only for the “WaitForTakeOut” state ensures that $beep$ is set while being in this state and the last line just keeps the current state stored. In Dhdl all registers are set to 0 on reset and through this “Idle”, with the value 0, is the initial state. As one can see, Dhdl provides a compact but easy to read style for annotating finite state machines.

Simulating the coffee machine in the Dhdl simulator, with the simulation specification for “coffee’ in Listing B.6, results in the signal trace of Figure 6.5. This trace also matches the machine specification. The machine starts in “Idle” (0). Then it is signaled that the coin is inserted and the state switches to “WaitSelect” (1). When the button for coffee is pressed, the state is changed to “WaitBeverage” (2) and the “makeCoffee” signal is set to true for one clock cycle. As the “beverageReady” signal is received, the state is set to “WaitForTakeOut” (3) and “beep” is true until the machine returns to the “Idle” state with the “cupRemoved” signal being true.

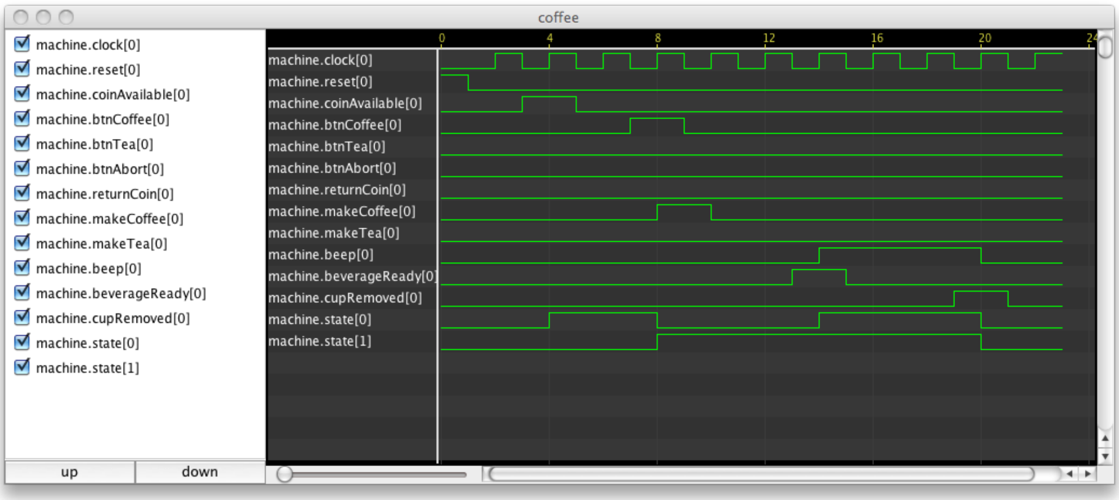

Figure 6.5: Simulation trace of the coffee machine design when making coffee.

With the additional code provided in the appendix, the coffee machine design has been successfully tested on an Altera DE2 Development and Education Board.

6.3 Asynchronous Design

The asynchronous design of choice is DARTS. For a detailed description, please consider [21]. It is sufficiently large to show that Dhdl is capable of modeling and simulating more complex systems, including their timing requirements. One DARTS node of an implementation with eleven nodes in [7] consists of 3218 basic gates and 100 C-Elements.

DARTS is an asynchronous design used for fault tolerant clock generation. Its functionality has been formally proven and successfully implemented on FPGAs and within an ASIC. Figure 6.6 shows the design of a single local clock generation module as implemented in this example. This diagram can be mapped directly to Dhdl. Of course, in practice, a synchronous design would be connected to the DARTS node’s output. Since the focus here is on the DARTS nodes, such a synchronous node is omitted, although its description would be straightforward.

In this thesis, a DARTS network of five nodes has been synthesized for the Altera DE2 Development and Education Board. All further observations refer to this design.

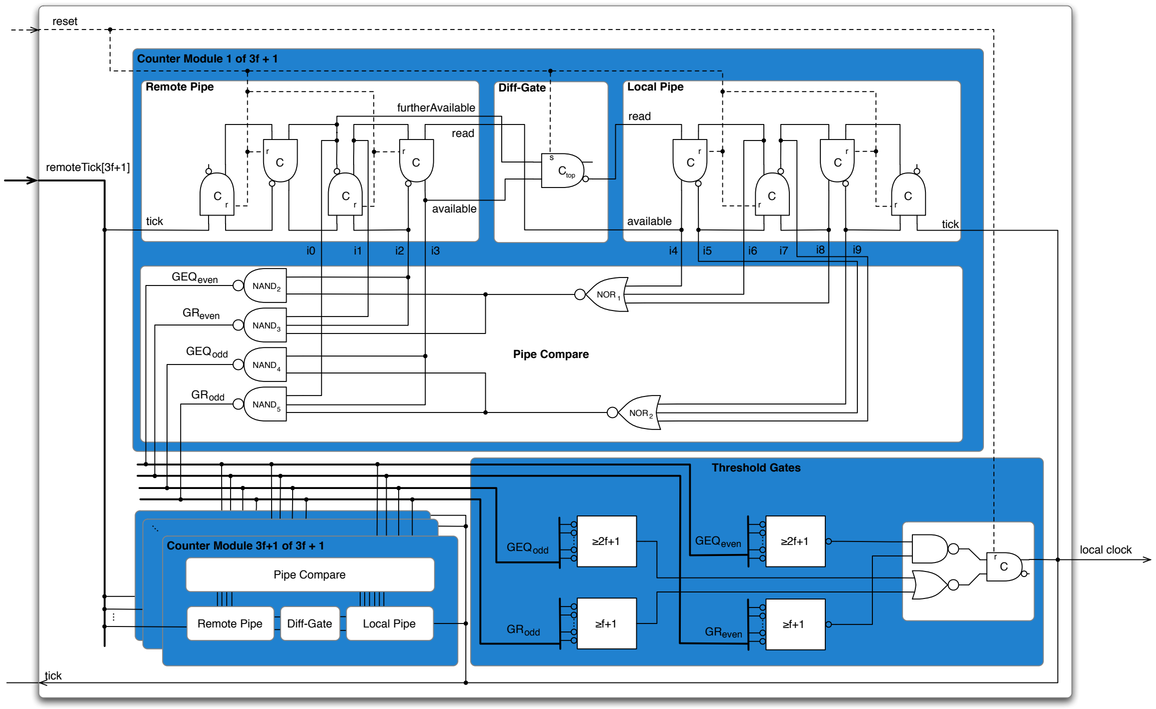

Figure 6.6: DARTS tick generation design.

6.3.1 Simulation

When simulating the tick generation without channels in-between the clocks, the result is just as symmetrical as the design itself. The traces in Figure 6.7 show that all DARTS clocks generate the next rising or falling edge at exactly the same time. Since all have the same state in the beginning, as well as the same architecture and distance, this corresponds to what is expected.

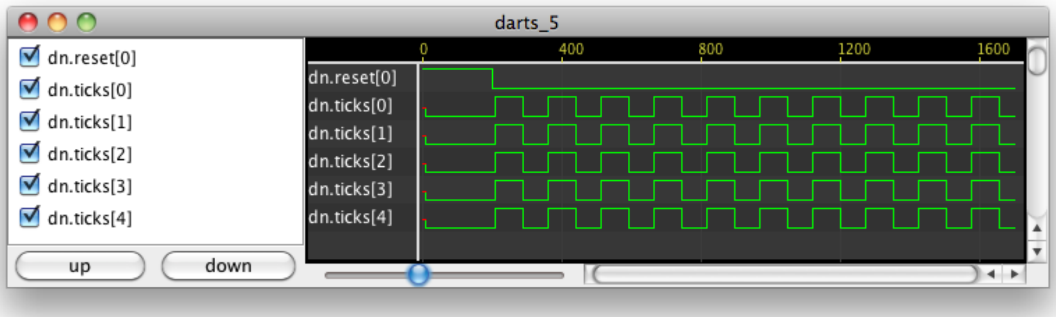

Figure 6.7: DARTS Dhdl simulation without Channels.

However, when adding channels with different delays in between the clock tick signals, one can get a simulation trace as in Figure 6.8. The generated traces still meet the synchronization criteria required by DARTS, but are no longer equal.

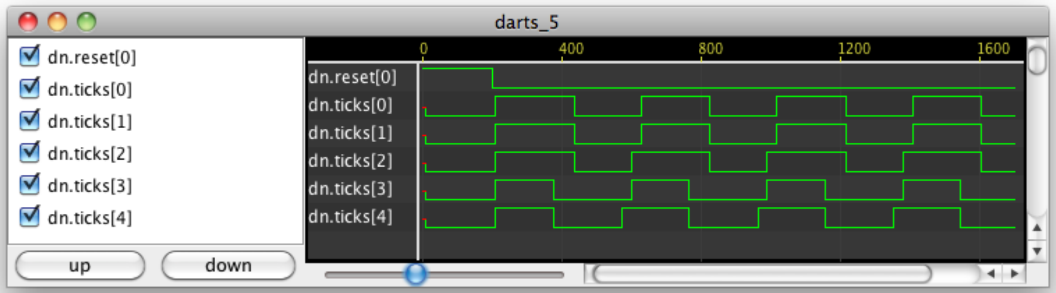

Figure 6.8: DARTS Dhdl simulation with Channels.

6.3.2 Timing Analysis

Two main properties of DARTS have been analyzed with Dhdl. First, the compliance with the DARTS timing constraints, and second, the frequency of the generated clock.

DARTS Timing Constraints

In order for DARTS to function properly, certain timing constraints have to be fulfilled by the hardware. These constraints are specified in [8, pages 56 and 57] and they are interlocking, tick removal and fastest progress. These constraints are translated to the Dhdl timing properties in the following three modules:

- the “CounterModule” module, see Listing 6.5,
- the “DartsClock” module, see Listing 6.6,
- and the “DartsNetwork” module, see Listing 6.7.

Except for the fastest progress constraint, the timing properties are named after the specific constraints and variables in [8]. The timing properties for the fastest progress constraint are defined within wiring code and thus have generic names. In the following, we will take a closer look at each of the timing delays[1]:

$tdiffm$ is the delay of the “DiffGate”.

[1]The name of the property in brackets is used in [8].

$tdiff$ (τ_{Diff}) is the merged delay of the delays from the local "DiffGates".

$tdiffmin$ (τ^{-}_{Diff}) is the minimum delay of $tdiff$.

$tdiffmax$ (τ^{+}_{Diff}) is the maximum delay of $tdiff$.

$tdis$ (T_{dis}) is the merged delay of the local tick generation loop through all "CounterModules".

$tmindis$ ($T_{min,dis}$) is the minimum delay of $tdis$.

$tmaxdis$ ($T_{max,dis}$) is the maximum delay of $tdis$.

$tmin$ (T_{min}) is the minimum delay of $tdis$ plus $tdiff$.

$tloc$ (T^{+}_{loc}) is an alias of $tmaxdis$.

$interlocking$ ($T_{max,dis} \leq T_{min} + T_{min,dis}$) is the interlocking timing constraint. It guarantees that all rising, as well as all falling ticks do not interfere with each other.

$tickRemoval$ ($\tau^{+}_{Diff} \leq T_{min}$) is the tick removal timing constraint. It states that the tick removal is faster than the local generation of new ticks.

$delay$ ($T^{-}_{first} \geq T^{+}_{loc}$) is the tick fastest progress timing constraint. It ensures that the slowest local tick generation delay is smaller than the smallest delay from a remote DARTS node.

The full Dhdl timing analysis output can be found in the appendix in Listing B.10. The hardware generated with Dhdl and synthesized by Quartus does not satisfy the interlocking and fastest progress constraints. Listing 6.4 shows an excerpt of the Dhdl timing analysis output, where a tick removal constraint is validated, and properties for interlocking and fastest progress result in errors. The 25 errors are the sum of the 5 interlocking constraints and the 20 (4*5) fastest progress constraints. The most likely cause for the generated hardware not to comply with the fastest progress constraint is the missing distance in-between the DARTS nodes, since only the DARTS network is synthesized. The interlocking constraint does not hold as there is a big difference between the minimum and maximum delays of $tdis$. This could be a result of the structure of the threshold gates. As they are specified recursively and not necessarily balanced, this might lead to significant delay differences. These problems with synthesis are still unsolved, however, Dhdl is capable of specifying all necessary timing constraints and evaluate them.

Listing 6.4: Excerpt of the Dhdl timing analysis output for the timing constraints.

```
...
33 error 5 dhdl/darts/DartsClock.df[line 27, char 31]: Delay not
   satisfied interlocking := (tmaxdis <= ((tmin * 2) + tmindis)) on
   path "clock[4]". The resulting delay is [2.0837E-8,
   2.8685000000000002E-8].
34 dhdl/darts/DartsClock.df[line 28, char 31]: Delay tickRemoval := (
   tdiffmax <= tmin) on path "clock[4]" is satisfied. [7.897E-9,
   9.578E-9]
```

```
35 dhdl/darts/DartsClock.df[line 16, char 15]: Delay tloc := tmaxdis
    on path "clock[1]" is satisfied. [2.8685000000000002E-8,
    2.8685000000000002E-8]
36 error 6 [line 29, char 102]: Delay not satisfied delay 1 := ((
    clock[0].thresholdGates[0].celem[0].bit[0].out[1], clock[1].
    thresholdGates[0].celem[0].bit[0].out[1]) >= clock[1] tloc) on
    path "base component". The resulting delay is [6.572E-9,
    4.0923999999999999E-8].
...
25 errors 0 warnings 0 assertion violations
```

Listing 6.5: Dhdl timing property for DARTS' constraints in the module "CounterModule".

```
delay tdiffm := (diff[0].celem[0].bit[0].invOut[0],
                 diff[0].celem[0].bit[0].invOut[0])
```

Listing 6.6: Dhdl timing property for DARTS' constraints in the module "DartsClock".

```
delay tdiff :=
        (counterModules[0].tdiffm ||
         counterModules[1].tdiffm ||
         counterModules[2].tdiffm ||
         counterModules[3].tdiffm)

delay tdiffmin := min(tdiff)
delay tdiffmax := max(tdiff)

delay tdis :=
        ((thresholdGates[0].celem[0].bit[0].out,
            counterModules[0].local[0].celem[0].bit[0].out) +
            (counterModules[0].local[0].celem[0].bit[0].out,
                thresholdGates[0].celem[0].bit[0].out)) ||
        ((thresholdGates[0].celem[0].bit[0].out,
            counterModules[1].local[0].celem[0].bit[0].out) +
            (counterModules[1].local[0].celem[0].bit[0].out,
                thresholdGates[0].celem[0].bit[0].out)) ||
        ((thresholdGates[0].celem[0].bit[0].out,
            counterModules[2].local[0].celem[0].bit[0].out) +
            (counterModules[2].local[0].celem[0].bit[0].out,
            thresholdGates[0].celem[0].bit[0].out)) ||
        ((thresholdGates[0].celem[0].bit[0].out,
            counterModules[3].local[0].celem[0].bit[0].out) +
            (counterModules[3].local[0].celem[0].bit[0].out,
                thresholdGates[0].celem[0].bit[0].out))

delay tmindis := min(tdis)
delay tmaxdis := max(tdis)
delay tmin := tmindis + tdiffmin
delay tloc := tmaxdis

delay interlocking := tmaxdis <= tmin*2 + tmindis
delay tickRemoval := tdiffmax <= tmin
```

Listing 6.7: Dhdl timing property for DARTS' constraints in the module "DartsNetwork".

```
: foreach clock[i] {
        foreach clock[j] {
            delay (clock[i].thresholdGates.celem[0].bit[0].out, clock[j].
                thresholdGates.celem[0].bit[0].out)  >= clock[j].tloc
        }
    }
```

Clock Frequency

Listing 6.8: Dhdl clock frequency estimation.

```
: foreach clock[i] {
    foreach clock[j] {
        if j < i {
            delay (clock[i].thresholdGates.celem[0].bit[0].out, clock[j].
                counterModules[i-1].remote.celem[3].bit[0].out)
        } else if j > i {
            delay (clock[i].thresholdGates.celem[0].bit[0].out, clock[j].
                counterModules[i].remote.celem[1].bit[0].out)
        }
    }
}
```

Since DARTS is an asynchronous clock generation algorithm, the generated clock frequency is of particular interest. With the timing property of Listing 6.8 in the "DartsNetwork" module, the delays in between the clock signals can be calculated. The Dhdl output can be found in the appendix in Listing B.11. An evaluation of these results follows in Section 6.3.3.

6.3.3 Executing on a Development Board

Figure 6.9 shows the resulting waveform generated on the FPGA of the Altera DE2 Development and Education Board. The frequency of the generated clock signal is 87.72MHz and so it has a period of 11.4ns. Thus, the delays in-between and within the single DARTS clocks have to be about 5.7ns. The Dhdl timing analysis of the delays between all output C-Elements of the clocks returns the values found in Table 6.1. The raw Dhdl output can be found in the appendix in Listing B.11. This indicates that the minimum delays determine the clock frequency in this FPGA implementation, since the highest minimum delay values are closest to the resulting clock impulse width (divided by two).

Table 6.1: Clock delays of the Dhdl DARTS solution on the Altera DE2 Development and Education Board.

From Clock Nr.	To Clock Nr.	Minimum Delay	Maximum Delay
1	0	$3.962 * 10^{-09}$	$1.350 * 10^{-08}$
2	0	$5.191 * 10^{-09}$	$1.544 * 10^{-08}$
3	0	$5.477 * 10^{-09}$	$1.456 * 10^{-08}$
4	0	$5.473 * 10^{-09}$	$1.468 * 10^{-08}$
0	1	$4.154 * 10^{-09}$	$1.651 * 10^{-08}$
2	1	$3.815 * 10^{-09}$	$1.610 * 10^{-08}$
3	1	$4.389 * 10^{-09}$	$1.300 * 10^{-08}$
4	1	$5.669 * 10^{-09}$	$1.620 * 10^{-08}$
0	2	$4.570 * 10^{-09}$	$1.631 * 10^{-08}$
1	2	$5.062 * 10^{-09}$	$1.662 * 10^{-08}$
3	2	$4.132 * 10^{-09}$	$1.565 * 10^{-08}$
4	2	$5.743 * 10^{-09}$	$1.475 * 10^{-08}$
0	3	$4.748 * 10^{-09}$	$1.531 * 10^{-08}$
1	3	$4.694 * 10^{-09}$	$1.454 * 10^{-08}$
2	3	$4.950 * 10^{-09}$	$1.634 * 10^{-08}$
4	3	$5.049 * 10^{-09}$	$1.303 * 10^{-08}$
0	4	$5.408 * 10^{-09}$	$1.415 * 10^{-08}$
1	4	$3.559 * 10^{-09}$	$1.538 * 10^{-08}$
2	4	$4.237 * 10^{-09}$	$1.333 * 10^{-08}$
3	4	$5.147 * 10^{-09}$	$1.589 * 10^{-08}$

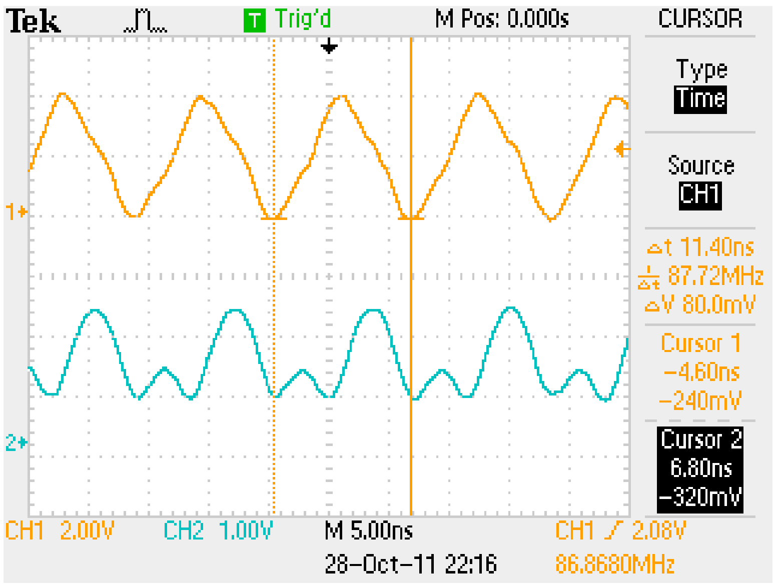

Figure 6.9: Waveform of the Dhdl DARTS solution on the Altera DE2 Development and Education Board.

CHAPTER 7

Conclusions

The aim of this thesis was to establish a direct link between Függer's formal analysis and modeling framework [9] and physical hardware designs. The core foundation for this link is the new programming language Dhdl. It is both close in style to the framework's description and powerful enough to manage larger designs.

Dhdl and the formal framework use the same structural elements and keywords. For timing properties, Dhdl uses all common algebraic operators and a parallel operator on tuples containing minimum and maximum delay values, which is as expressive as the formal framework's timing requirements. A significant subset of Dhdl can be interpreted with a formal semantic.

For simulation and semantic interpretation, a time model, called causal time, is used, that derives the simulation delays directly from the module structure of the hardware description. The advantage of the causal time model is that it reflects dependencies within designs at an early stage. Moreover, the simulation traces generated with it are often accurate enough to predict the overall behavior of physical designs.

The Dhdl formal semantics is structural operational semantics consisting of 14 rules operating on a structure that contains the state of the system and the code for its definition. The semantic interpretation of the expressions is done by a separate algebra, the results of which are not necessarily directly mappable to the domain of the formal framework. It is shown that, if all elements of the interpreted expressions are mappable to the formal framework, so is the result.

A set of representative reference designs were specified in Dhdl, simulated and their implementation checked on the development board. Their timing and behavior on the development board fitted to the Dhdl delay properties results and the simulations. However, only one timing constraint of DARTS could be verified.

Dhdl fulfills all the necessary and requested features and is a language that connects Függer's formal analysis and modeling framework directly to hardware generation. Furthermore, it can be used to describe hardware designs in an efficient way and offers a new kind of simulation with its causal time model. Still, it can only verify and not enforce that the generated hardware satisfies all timing constraints.

Still, this leaves a lot of space for future work. The next big step would be to pass timing constraints to the external tool, if they are not fulfilled. Another improvement could be the integration of a constraint solver that checks whether delay properties are reasonable and satisfiable. This would allow to show erroneous assumptions about the timing behavior at an early stage and independently of the generated netlists. An even more ambitious extension would be the integration of the reasoning from the formal framework and a mechanic check for validity of the stated properties. Aside from that, also Dhdl simulations could be translated to VHDL testbenches and by this used in further stages of the development flow.

APPENDIX A

Dhdl Language Specification

A.1 Introduction

Dhdl is designed for integrated circuit definition and due to this limited in its general descriptiveness of circuit behavior. The philosophy behind Dhdl is to use the best suitable language for each description problem, and so Dhdl consists of three languages. Their targets are hardware, simulation and a small one for packaging description.

A.1.1 Remarks

This document describes the version 0.9 of Dhdl. In this specification, the syntax will be annotated as defined in [3, p. 28]. Please note that optional whitespace characters, that can be placed usually anywhere within the languages, will be ignored in the syntax description for the sake of simplicity. Only required newline characters are annotated.

A.1.2 Structure of this Specification

Definitions common for all languages are defined in Section A.2. Hardware, simulation and package description are covered in Sections A.3, A.4 and A.5. The files generated by Dhdl are described in Section A.6 followed by the full syntax definition in Section A.7. Section A.8 contains the translator guide and finally some code examples are given in Section A.9.

A.2 Common Definitions

Due to the nature of the problem, the definition and simulation languages of Dhdl share some common definitions, as well as the package description. The syntax for all these language constructs can be found in Section A.7.

A.2.1 Basic Language Elements

This section contains the most basic elements that the Dhdl languages consist of.

Whitespace and Newline

For most cases whitespace and newline characters are in general ignored and just used as separators. The whitespace characters are ignored in all syntax definitions. In certain syntactical constructs newline characters are required and annotated, but otherwise they are also ignored.

Identifiers

The definition of Identifiers is as in VHDL [11, p. 179]. They start with a character and can contain characters, numbers, and single underlines, but there must not be an underline at the end. In Dhdl also the following suffixes should be avoided since they are used for VHDL generation:

- _port
- _signal
- _register
- _subcomponent

Numbers

Unsigned integers can be represented in binary, octal, decimal, and hexadecimal annotated like in C. Rational numbers can only be used in delay expressions. They can have us, ns and ps suffixes to be multiplied by the corresponding factor (10^{-6}, 10^{-9} and 10^{-12}). The unit seconds will be ignored, but since it is only used in delay expressions, it is assumed that numbers used without this indication still will be of type seconds and thus the term will be correctly interpreted.

Constants

In Table A.1, the constants with the according values are shown. The definition of the types is found in Section A.2.3

Table A.1: Dhdl constants

Constant	Value	Type
`true`	1	$\mathcal{B}$
`false`	0	$\mathcal{B}$
`high`	1	$\mathcal{B}$
`low`	0	$\mathcal{B}$
`undef`	-	$\mathcal{W}$

A.2.2 Code Structure

All commands are terminated with a newline or the closing brace of a code block. As in many other languages, code blocks start with a left brace and end with a right one. Unlike in other languages, code blocks can only be inserted with certain commands or declarations, as given in the syntax.

A.2.3 Types

Dhdl is a strong typed language. Types in Dhdl are designed to separate between different specification areas. There are five primitive types, $\mathcal{W}$, $\mathcal{B}$, $\mathcal{N}$, $\mathcal{I}$ and $\mathcal{D}$. $\mathcal{W}$ ranges over binary values and undefined. $\mathcal{B}$ is for binary values, $\mathcal{N}$ for natural numbers including zero and $\mathcal{I}$ for integer values. Values in $\mathcal{D}$ define a range over decimal numbers. The order is $\mathcal{B} \subseteq \mathcal{W}$ and $\mathcal{B} \subseteq \mathcal{N} \subseteq \mathcal{I} \subset \mathcal{D}$. Type conversion is performed automatically. Only expressions of the wire type $\mathcal{W}$ can be later generated into hardware. $\mathcal{D}$ is used for checking timings and all other types are for generating hardware structure. The only two entity representations of $\mathcal{W}$ are `Signal` and `Register`. A comprehensive description of the types can be found in Table A.2. The ranges of the primitive datatypes are derived from the types used within the Java implementation of Dhdl. For further information see the definition of long and double in [16]). Due to this, $\mathbb{R}$ means a decimal number with the precision and within the range of double. The undef element represents an unknown value of a wire and is used for $\mathcal{W}$. The delay type $\mathcal{D}$ is a 2-tuple, representing a range where the first element is the minimum and the second element the maximum value. These elements are also annotated with $v.min$ and $v.max$ where v is of type $\mathcal{D}$. Furthermore $\mathcal{I} \subset \mathcal{D}$ where the single value is used as minimum and maximum value, leaving aside the imprecision of the floating point representation.

Table A.2: Dhdl datatypes

Name	Keyword	Symbol	Type	Range
Signal	`Signal,Register`	$\mathcal{W}$	wire	$\mathcal{B} \cup$ undef
Boolean	`bool`	$\mathcal{B}$	generic	$[0,1] \in \mathbb{Z}$
Natural	`nat`	$\mathcal{N}$	generic	$[0,2^{63}-1] \in \mathbb{Z}$
Integer	`int`	$\mathcal{I}$	generic	$[-2^{63},2^{63}-1] \in \mathbb{Z}$
Delay	`delay`	$\mathcal{D}$	delay	$(\mathbb{R},\mathbb{R})$

A.2.4 Operators

Table A.3 shows all the operators that are available in Dhdl. The lower the precedence level of an operator is, the stronger it binds. The evaluation order of operators with the same precedence level is undefined.[1] For delay expressions, the parallel operator is introduced to merge two parallel paths. In the following, the operator definitions on the domains $\mathcal{W}$ and $\mathcal{D}$ are given separately. The definition on all other domains is the common interpretation.

[1]In the current version it is from right to left, e.g. $a + b - c = a + (b - c)$.

Table A.3: Dhdl operators

Operator	Name	Precedence Level	Type	Possible Signatures
`not`	logical not	1	unary	$\mathcal{B} \mapsto \mathcal{B}$ or $\mathcal{W} \mapsto \mathcal{W}$
`and`	and	4	binary	$\mathcal{B} \times \mathcal{B} \mapsto \mathcal{B}$ or $\mathcal{W} \times \mathcal{W} \mapsto \mathcal{W}$
`nand`	nand	4	binary	$\mathcal{B} \times \mathcal{B} \mapsto \mathcal{B}$ or $\mathcal{W} \times \mathcal{W} \mapsto \mathcal{W}$
`or`	or	5	binary	$\mathcal{B} \times \mathcal{B} \mapsto \mathcal{B}$ or $\mathcal{W} \times \mathcal{W} \mapsto \mathcal{W}$
`nor`	nor	5	binary	$\mathcal{B} \times \mathcal{B} \mapsto \mathcal{B}$ or $\mathcal{W} \times \mathcal{W} \mapsto \mathcal{W}$
`xor`	xor	5	binary	$\mathcal{B} \times \mathcal{B} \mapsto \mathcal{B}$ or $\mathcal{W} \times \mathcal{W} \mapsto \mathcal{W}$
`equiv`	equivalent	5	binary	$\mathcal{B} \times \mathcal{B} \mapsto \mathcal{B}$ or $\mathcal{W} \times \mathcal{W} \mapsto \mathcal{W}$
`-`	unary minus	1	unary	$\mathcal{I} \mapsto \mathcal{I}$ or $\mathcal{D} \mapsto \mathcal{D}$
`+`	plus	3	binary	$\mathcal{I} \times \mathcal{I} \mapsto \mathcal{I}$ or $\mathcal{D} \times \mathcal{D} \mapsto \mathcal{D}$
`-`	minus	3	binary	$\mathcal{I} \times \mathcal{I} \mapsto \mathcal{I}$ or $\mathcal{D} \times \mathcal{D} \mapsto \mathcal{D}$
`*`	multiply	2	binary	$\mathcal{I} \times \mathcal{I} \mapsto \mathcal{I}$ or $\mathcal{D} \times \mathcal{D} \mapsto \mathcal{D}$
`/`	divide	2	binary	$\mathcal{I} \times \mathcal{I} \mapsto \mathcal{I}$ or $\mathcal{D} \times \mathcal{D} \mapsto \mathcal{D}$
`%`	modulo	2	binary	$\mathcal{I} \times \mathcal{I} \mapsto \mathcal{I}$ or $\mathcal{D} \times \mathcal{D} \mapsto \mathcal{D}$
`||`	parallel	4	binary	$\mathcal{D} \times \mathcal{D} \mapsto \mathcal{D}$
`!`	not	1	unary	$\mathcal{B} \mapsto \mathcal{B}$
`==`	equals	6	binary	$\mathcal{I} \times \mathcal{I} \mapsto \mathcal{B}$ or $\mathcal{D} \times \mathcal{D} \mapsto \mathcal{B}$
`!=`	not equals	6	binary	$\mathcal{I} \times \mathcal{I} \mapsto \mathcal{B}$ or $\mathcal{D} \times \mathcal{D} \mapsto \mathcal{B}$
`<`	less than	6	binary	$\mathcal{I} \times \mathcal{I} \mapsto \mathcal{B}$ or $\mathcal{D} \times \mathcal{D} \mapsto \mathcal{B}$
`<=`	less or equals	6	binary	$\mathcal{I} \times \mathcal{I} \mapsto \mathcal{B}$ or $\mathcal{D} \times \mathcal{D} \mapsto \mathcal{B}$
`>`	greater	6	binary	$\mathcal{I} \times \mathcal{I} \mapsto \mathcal{B}$ or $\mathcal{D} \times \mathcal{D} \mapsto \mathcal{B}$
`>=`	greater or equals	6	binary	$\mathcal{I} \times \mathcal{I} \mapsto \mathcal{B}$ or $\mathcal{D} \times \mathcal{D} \mapsto \mathcal{B}$

Operators in $\mathcal{D}$

In Table A.4 the definition of the operators in $\mathcal{D}$ is given. The column result defines the final 2-tuple and in the column truth value is the expression for the Boolean evaluation.

Operators in $\mathcal{W}$

The Tables A.5, A.6, A.7, A.8, A.9, A.10 and A.11 show the truth tables for operators in $\mathcal{W}$.

Table A.4: Operator definition for $\mathcal{D}$

Term	Result	Truth Value
$v_1 + v_2$	$(v_1.min + v_2.min, v_1.max + v_2.max)$	-
$v_1 - v_2$	$(v_1.min - v_2.min, v_1.max - v_2.max)$	-
$v_1 * v_2$	$(v_1.min * v_2.min, v_1.max * v_2.max)$	-
v_1/v_2	$(v_1.min/v_2.min, v_1.max/v_2.max)$	-
$v_1\%v_2$	$(v_1.min\%v_2.min, v_1.max\%v_2.max)$	-
$v_1 \parallel v_2$	$(\min(v_1.min, v_2.min), \max(v_1.max, v_2.max))$	-
$v_1 == v_2$	$(\min(v_1.min, v_2.min), \max(v_1.max, v_2.max))$	$v_1.min == v_2.min \land v_1.max == v_2.max$
$v_1! = v_2$	$(\min(v_1.min, v_2.min), \max(v_1.max, v_2.max))$	$v_1.max < v_2.min \lor v_1.min > v_2.max$
$v_1 < v_2$	$(\min(v_1.min, v_2.min), \max(v_1.max, v_2.max))$	$v_1.max < v_2.min$
$v_1 <= v_2$	$(\min(v_1.min, v_2.min), \max(v_1.max, v_2.max))$	$v_1.max \leq v_2.min$
$v_1 > v_2$	$(\min(v_1.min, v_2.min), \max(v_1.max, v_2.max))$	$v_1.max > v_2.min$
$v_1 >= v_2$	$(\min(v_1.min, v_2.min), \max(v_1.max, v_2.max))$	$v_1.max \geq v_2.min$

Table A.5: Truth Table Logical NOT

Input	Output
0	1
1	0
undef	undef

Table A.6: Truth Table AND

Input 1	Input 2	Output
0	0	0
0	1	0
0	undef	0
1	0	0
1	1	1
1	undef	undef
undef	0	0
undef	1	undef
undef	undef	undef

Table A.7: Truth Table NAND

Input 1	Input 2	Output
0	0	1
0	1	1
0	undef	1
1	0	1
1	1	0
1	undef	undef
undef	0	1
undef	1	undef
undef	undef	undef

Table A.8: Truth Table OR

Input 1	Input 2	Output
0	0	0
0	1	1
0	undef	undef
1	0	1
1	1	1
1	undef	1
undef	0	undef
undef	1	1
undef	undef	undef

Table A.9: Truth Table NOR

Input 1	Input 2	Output
0	0	1
0	1	0
0	undef	undef
1	0	0
1	1	0
1	undef	0
undef	0	undef
undef	1	0
undef	undef	undef

Table A.10: Truth Table XOR

Input 1	Input 2	Output
0	0	0
0	1	1
0	undef	undef
1	0	1
1	1	0
1	undef	undef
undef	0	undef
undef	1	undef
undef	undef	undef

Table A.11: Truth Table EQUIV

Input 1	Input 2	Output
0	0	1
0	1	0
0	undef	undef
1	0	0
1	1	1
1	undef	undef
undef	0	undef
undef	1	undef
undef	undef	undef

A.2.5 Expressions

Dhdl defines different types of expressions:

- Logical expressions
- Guarded expressions
- Arithmetical expressions
- Comparison expressions
- Delay expressions
- Array expressions

Within these expressions only a specific subset of operators might be used, as defined by the syntax. Besides the syntax, also matching array widths and operator types are necessary for an expression to be valid.

Logical Expressions

Logical expressions are used to define circuit logic for signals. Within logic expressions also submodules can be included. A logic expression can also be seen as a single guarded expression.

Guarded Expressions

The guarded expression is the most important element in Dhdl to make case distinctions. They are only used in hardware description and therefore explained in detail in Section A.3.11.

Arithmetical Expressions

The return value of an arithmetical expressions is within $\mathcal{I}$. It has all the logic operators as well but contrary to logic or guarded expressions it is not possible to use a new submodule within it.

Comparison Expressions

The result of a comparison expression is of type $\mathcal{B}$. When comparison expressions are written in a comma separated list, the result is the "and" concatenation of all of them. When only an arithmetic expression is used the result within $\mathcal{B}$ is true if the arithmetic result is one and false otherwise.

Delay Expressions

Delay expressions can access all signals of the current component and its subcomponents. However, not all signals will be actually generated to hardware and so only these can be accessed later for verifying the timing behavior. Luckily, interesting paths usually lead from one storage element to another and thus those path delays are most likely to be computable.

Array Expressions

An array expressions yields a range of natural numbers. When defining an array of signals, like in C, a single index n will give a range from 0 to $n-1$. Using an array expression for access, a single value is interpreted as a single index of an array.

A.3 Hardware Description

The elements of the hardware description in Dhdl reflect the one used in Függer's formal framework (see [9]). This language is more oriented towards functional programming languages since they are naturally closer to the structure of hardware. Dhdl strongly separates between the functions that are mapped to hardware logic and functions that are used to generate structures.

A.3.1 Packages

Dhdl code is organized in packages. A package identifier determines with the first dot separated names the path from the source or library folder and with the last name the filename without suffix. The main module must have a name identical to its package and be defined as public.

A.3.2 Signal

Signals and arrays of signals are used to define wires generated in the final hardware.

Array Using arrays it is possible to define signals over a range of indices. Like in C and Java, an array defined with a single value n will have indices starting from 0 to $n - 1$. With the keywords `to` and `downto` it is possible to define specific ranges. An example can be found in Listing A.1

Listing A.1: Example module using signal arrays.

```
module test {
        in Signal in[5]
        out Signal out[0 to 4] <= in
}
```

Register

Registers are special types of signals. Each register will be connected to the local visible signals named reset and clock, which have to exist for correct translation. In its current implementation the Register is set to 0 on a high reset and triggers on the rising edge of the clock signal.

A.3.3 Module

In Dhdl systems are built out of modules. They are the basic building block for the structure. A module is a collection of ports, signals, submodules and their interconnection by simple wiring or as functions of other signals.

Extending other modules A module can extend other modules. This is done by simply copying the port definitions, which are not defined within the current module, and checking whether the defined ones match. This can have unforeseen side effects, e.g. a module uses a local visible definition, and thus should be used carefully. The modules it extends are simply listed after the keyword `extends`.

Implementing interfaces A module implements an interface if the interface's ports are a subset of its ports. Also, the interface name must be listed after the keyword `implements` within the modules header.

Generic parameters Generic parameters are parameters with a generic type (bool, nat, int). These parameters are passed in angle brackets and can be assigned default values as shown in Listing A.2. Within the module body the value of *gen_param* is then 3, if no other value is provided. Generic parameters can also be used to define the default width of regular parameters.

Listing A.2: Example of a generic parameter definition.

```
module <nat gen_param <= 3> { }
```

Parameters Parameters are in ports that can be connected within an expression. If a parameter can have an arbitrary array width, it is indicated with an identifier in square brackets, like the example in Listing A.3. There the in port *param* has width n, which is 2, when no other value is specified.

Listing A.3: Example of a parameter definition.

```
module (Signal param[n<=2]) { }
```

A.3.4 Port

Ports can hold values of the wire datatype and have a direction (`in` or `out`). Unlike to VHDL, it is possible to read values of output ports. With the keyword `forward`, an input port can be defined to automatically connect to all submodules, which have unassigned ports with the same name and width. This comes in handy when using, e.g. reset lines. Out ports can be set an expression at their definition or later in the wiring code. Every out port must be connected at some point. Since Dhdl describes hardware and these ports represent wires, they can only be assigned once. Also see A.3.5 for further notes on assigning expressions. The Ports of a module can only be accessed from this very module and its enclosing one. An exception is made for timing properties, which can refer to all signals of its module and submodules.

Special Named Ports There are two special ports, namely the input port named in and the out port named out. When a module has such an input port, then this port can be assigned by only using the modules reference name, and likewise the out port can be connected. Also when using a module directly within an expression it must have an output port called out.

A.3.5 Members

There are three types of members and they can be used according to their datatype. Regardless of their type, each member can only be assigned once.

Wire typed Wire typed members behave like ports, but are only visible within the module.

Structure typed Structure typed members, further also called submodule members, define subcomponents. The in ports of these subcomponents must be assigned and can be read. Naturally, their out ports can only be read.

Generic typed Generic typed members must be assigned at their definition and these expressions will be evaluated before executing the wiring code. These members will only appear in symbol lists and never in the final modules.

Assignment on declaration

The expressions assigned to the members are treated like wiring code assignments and are "executed" after all elements of the module body are analyzed. The execution order is not guaranteed.[2] This is only of concern when assigning interface typed members to other interface typed members. To avoid errors, it is recommended to assign these types of members within the wiring code segments of a module.

A.3.6 Interface

An interface itself contains only port information. Like in other programming languages (e.g. Java) it is a placeholder. With the keyword `implements` it is annotated that a module implements an interface. Furthermore, interfaces can have generic parameters, which can also be passed down values that are in the module's scope. A set of sample interface definitions can be found in the appendix A.9.1. During the analysis of the module, it is verified, that the ports of the interface are a subset of the ports of the module.

A.3.7 Macros

There are two different kinds of macros depending on the result they produce. They are defined in Java and called by the translator.

Optimization macros

Optimization macros for logical expression are accessed like modules with only generic parameters. In arithmetical expressions they have simple parentheses. These macros will be evaluated in the optimization stage.

Generation macros

These macros appear like regular modules and will generate its specific simulation code, as well as VHDL code.

A.3.8 Wiring Code

The wiring code takes care of the interconnection of elements that have not been assigned on declaration, e.g. ports of submodules. Wire code statements are executed sequentially in the order of appearance from top down. As they are executed all elements of the used expressions must be fully evaluable at that time. Special care must be taken when assigning a multiline expression, this can only be done in a single line wiring code, since the syntax would be ambiguous

[2]Currently it is in the order of appearance of the members from top to bottom.

otherwise. Also, if it would be necessary to assign a multiline expression within a list of wire statements, it is most likely that a new submodule would do a better job.

Here, ports and members will be referred to as variables.

Definition of Members

Wire code always defines members and never ports. On definition they can be assigned an expression.

Assignment Statements

As previously described, all out ports and members can be assigned exactly once. Of course, out ports of submodules cannot be assigned. Generic typed members can also only be assigned once.

A special case are the submodule members. If it is of a module type the assigning operator acts as described in the previous paragraph. Otherwise, the first assignment must determine the module referenced by the member. This can be done by referencing a member that is defined with a module type or a member that is of interface type, but already refers to some component. The example in Listing A.4 can be executed assuming that moduleComponent implements interfaceComponent. After execution, there is a sub component of type moduleComponent called “a” and two references to that, namely “b” and “c”.

Listing A.4: Interconnect example for component members.

```
: moduleComponent a
: interfaceComponent b <= a
: interfaceComponent c <= b
```

Foreach

This construct is the only loop available in the wiring code and since its purpose is to wire signals, it is also the only one needed.

The code within the “foreach” body is executed as many times as the range of the array of the variable wide is. In each execution the index variable defined in the “foreach” head has the value of the current array index. The order is dependent on the definition of the array. Arrays defined as [3] or [0 to 2] yield executions like 0,1,2. Likewise, an array defined as [2 downto 1] yields 2,1,0.

If

Like in all imperative languages, the first condition in the construct that evaluates to true determines the execution of the wire statements in the body. If none is true, the body of else will be executed, if it exists.

A.3.9 Delay Assertions

Delay assertions have names for better identifications. During translation, the expressions for measuring delays are checked, whether the referenced signals exist. On delay analysis their values will be calculated and an error reported when an assertion fails.

A.3.10 Constraints

Constraints given in this style will be checked before any wiring code is executed. This is especially helpful to assert that generic parameters are within a valid range and the connected signals have valid array widths.

A.3.11 Defining Circuit Logic

The key construct for defining circuit logic in Dhdl is the guarded expression. These expressions are suitable for computing different functions, as well as defining finite state machines. A guarded expression consists of subexpressions, where each has its own guard expression and a resulting logic expression. In the most trivial case, the guarding expression is empty and thus true. It can also be a list of expressions, where, if and only if all expressions evaluate to true, so does the guarding expression. The result of a guarded expression is the result of the first logic expression for which its guarding expression evaluate to true.

Recursive Computation

With this type of expressions many computational problems can be solved recursively. Listing A.5 shows a module that compares two arrays of signals for equality. The code is straightforward. The assertion at the beginning of the module guarantees that both signal arrays, $value1$ and $value2$, have the same width. The expression for the result reads like this: If the signal arrays are one signal wide, then the equivalence of those two signals is computed. Or, if the signals at index 0 are equal, then the result depends on the rest of the of signals to be equal. Otherwise, these signal arrays are not equal. Clearly, the structure of guarded expressions favors recursive definitions.

Listing A.5: Full equality example

```
module isEqual(Signal value1[n], Signal value2[m]) {
        :- n == m

        out Signal out <=
                n == 1 ->
                        value1[0] equiv value2[0]

                value1[0] == value2[0] ->
                        isEqual(value1[1 to n-1], value2[1 to n-1])

                -> false
}
```

Finite State Machines

Another feature is that guarded expressions can also have side effects. When a guarding expression evaluates to true also the value of other signals can be set, provided that those signals have a general expression for determining their value otherwise. If that would not be the case, latches could easily be created by accident. Please note that cross-referencing guarded expressions is not recommended, but it is currently not checked during translation. A simple synchronous example for a traffic light state machine is given in Listing A.6. The output ports red, $yellow$, $green$ represent when the according light is turned on. The false values assigned to these signals are their default values. These values can be overridden in the side effects of the guarded expression for the state register. So, if the state is RED, the signal red will be true since it is assigned in both side effects of the corresponding guarding expressions. Similarly, if the state is $REDYELLOW$, the red and yellow signals will be true. An example with identical simulation traces, but without side effects can be found in Section A.9.4.

Listing A.6: Synchronous traffic light.

```
import dhdl.lang

define WIDTH 2

define logic RED                toArray<0, WIDTH>
define logic REDYELLOW          toArray<1, WIDTH>
define logic GREEN              toArray<2, WIDTH>
define logic YELLOW             toArray<3, WIDTH>

module TrafficLights {
        forward in Signal clock
        forward in Signal reset

        in Signal next

        out Signal red <= false
        out Signal yellow <= false
        out Signal green <= false

        Register state[WIDTH] <=
                state == RED, next == true
                        ->      red <= true,
                                REDYELLOW

                state == RED
                        ->      red <= true,
                                state

                state == REDYELLOW, next == true
                        ->      red <= true,
                                yellow <= true,
                                GREEN

                state == REDYELLOW
                        ->      red <= true,
                              yellow <= true,
                              state

               state == GREEN, next == true
                       ->     green <= true,
                              YELLOW

               state == GREEN
                       ->     green <= true,
                              state

               state == YELLOW, next == true
                       ->     yellow <= true,
                              RED

               state == YELLOW
                       ->     yellow <= true,
                              state
}
```

A.4 Simulation Description

Naturally, simulations and tests are described in a sequential style since inputs are applied and outputs checked. Therefore, the language for defining simulations in Dhdl is an imperative language.

A.4.1 Simulation

A simulation consists of three phases. The setup phase, where modules are added to the simulator and variables are initialized. This phase is ended with the simulator-start command. Then follows the register phase where, with the simulator-listen command, signals are added to the listening list. This list specifies all signals of which the traces are recorded. Afterwards, the run phase starts where signal values are assigned, time is advanced and outputs are checked.

A.4.2 Simulator

The simulator itself is generated and compiled to Java code with the simulator-start command. This is done to achieve fast runtimes. The simulator controls the calculation of the signal values. As the signal takes a step, time is advanced by one tick and the new signal values are calculated purely based on the previous values. The simulator also records these values, if they are on the listening list. Also, each signal state can be accessed and a string representation of the module structure generated. The simulator is controlled with the simulator commands.

A.4.3 Commands

The commands are interpreted sequentially and only the syntax is checked before execution. If an error occurs during execution of a command, the simulation is aborted.

Variable

Naturally, variables are treated differently in the simulation than in the hardware definition. Here, each variable can be assigned values more than once and this is emphasized with the set symbol.

Definition Variables can be defined at any point within the simulation.

Assignment Contrary to the hardware language, when assigning values to signal arrays or Boolean variable arrays, it is possible to use values within $\mathcal{N}$. This value is then automatically split up in a Boolean array and leading zeros are added. If the destination array is to small, an error is printed and the current simulation aborted.

General Access Variables defined within the simulation have the array access as usual, but signals taken from the simulator are assumed to be accessed at array index 0, if no array is specified. This is necessary due to the current nature of the simulator, but might change in further versions of Dhdl.

Simulator Commands

Simulator commands explicitly control the simulator.

Add Adds a new module to the simulator, which can then be accessed after executing the start command with the provided name.

Start Generates the simulator with all the previously added modules and compiles it to Java binary code for fast execution.

Listen Adds a signal to the listening list in order to record its trace.

Plot The plot command prints the recorded signal traces to the specified output stream. The values 0 and 1 represent false and true, whereas 2 stands for undef.

Show The show command opens the wave viewer and displays the so far recorded signal traces. See Section A.4.4 for more information on the wave viewer.

Print

Prints the specified expression.

If

Executes the first statements block, which preceding clauses evaluate to true. Otherwise, the else block is executed. Except for the first if block, all others are optional.

While

Executes the statement block while the given clauses evaluate to true.

Do-while

Executes the statement block once and then continues evaluating it if the while clauses evaluate to true.

Do-until

Executes the statement block once and then continues evaluating it if the until clauses evaluate to false.

For

The for loop defines a variable on a range in its header. The body is executed as often as the range is wide and each time with the variable having the current index number. With the keywords `to` and `downto` the sequence of the iteration values is controlled.

Foreach

See the definition of the wiring code for-each in Section A.3.8.

A.4.4 Wave viewer

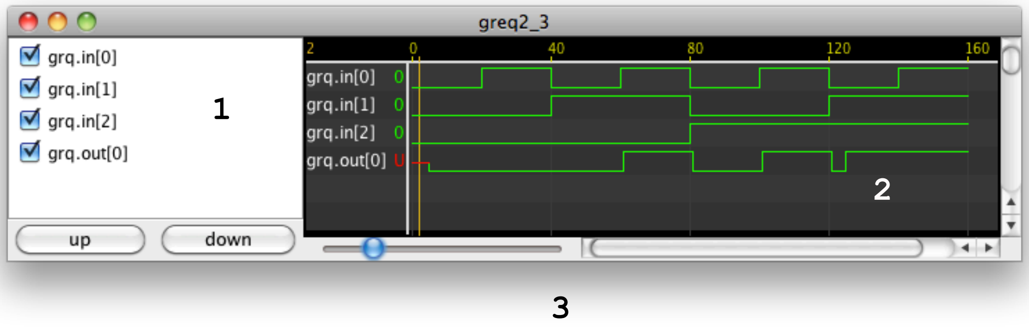

Figure A.1: Gui elements of the wave viewer with indices.

The wave viewer provides a convenient and fast way for displaying the current simulation results. Please keep in mind that looking at traces should never replace writing assertions. Figure A.1 shows an example trace within a wave viewer window. The signal list (1) contains all the recorded signals within the simulation run. The checkboxes tell whether they are included in the plot area (2) or not. With the right mouse button it is possible to mark signals in the list and move them with the up and down buttons below. In the plot area it is possible to set three different cursors, one with each mouse button. The values of the signals at the current cursor position are displayed on the right side of them. In this context U stands for undef. The zoom can be controlled with the slider (3) below.

A.5 Package Description

Listing A.7: Syntax for a package description file.

```
package ::= package_property*

package_property ::=
        Identifier ':=' port_value |
        'pin' port_access ':=' Identifier

port_access ::=
        Identifier '.' Identifier array?

port_value ::= [^(Newline | ' ' | '\r')]

array ::= '[' [0-9]+ ']'
```

Package description in the current version of Dhdl is a simple configuration file for defining the module to generate and to which pins it should be connected, as well as which FPGA to use. From this information a tcl-script is generated to drive the external tool for synthesis. The reason not to simply use the generated VHDL files for the module is to make the task easier for the designer. The generated components in VHDL have similar, but still different names to the ones in the Dhdl files. With this method, it is also possible to call the external tool from within Dhdl and perform checks beforehand.

A.6 Generated Files

During the whole design flow usually two types of files are generated by Dhdl itself. The hardware description is translated to VHDL and the package file to a tcl file for the external tool.

A.6.1 VHDL File

For each separate parameter setting of a module, a VHDL file is generated. These VHDL files contain the full entity and architecture description for each of these used modules together with their specific parameter settings.

The names are changed for VHDL in the following style. The entity name is the module name followed by an underscore and an integer identifier. The ports of the entity have the port names of the module followed by "_port". For all signals including ports and registers, a signal with the same name followed by "_signal" is defined. For registers, also a signal with that name and "_register" appended is defined. The expressions are assigned to the port or signal names extended with “_signal”, or to the ones with “_register” for registers. The values are always read from signals with the suffix “_signal”.

The architecture contains three or four processes, depending on whether or not registers are used. The processes “in_ports” and “out_ports” connect the input and output ports, respectively, with the according signal. In “register_sync”, the register logic is defined and finally, “calculate_signals” contains the specified logic. In that process, all signals with simple logic expressions are assigned and afterwards, all guarded expressions are translated to if-then-else structures, including the side effects.

A.6.2 Tcl File

The generated tcl file of this Dhdl version is intended to be run by Quartus, which is currently the only supported external tool. It contains the information retrieved from the package file, namely FPGA family, the specific device and the pin assignments. Also, the VHDL top level entity is included, as well as some settings to generate timing files that are needed later within Dhdl for delay verification.

A.7 Syntax

This section gives an overview of the syntax that is used within the hardware and simulation definition.

A.7.1 Common Definitions

Whitespace and Newline

Listing A.8: Whitespace and Newline.

```
Whitespace ::=
        ' ' | '\t'

Newline ::=
        (('\r')? '\n')
```

Identifiers

Listing A.9: Syntax definition of identifiers.

```
Identifier ::=
        [a-zA-Z] ('_' Identifier_character |
                Identifier_character)*

Identifier_character ::= [a-zA-Z0-9]
```

Numbers

Listing A.10: Syntax definition of numbers.

```
value ::=
        number | constant

number ::=
        Binary_number | Octal_number |
        Decimal_number | Hexadecimal_number

Binary_number ::= '0b' ('0'|'1')+
Octal_number ::= '0o' [0-7]+
Decimal_number ::= '0d'? [0-9]+
Hexadecimal_number ::= '0x' [0-9a-fA-F]+
```

Listing A.11: Syntax definition of rational numbers.

```
Rational_number ::=
        Dec_number '.' Dec_number
                ('e' ('+' | '-')? Dec_number)?
                ('us' | 'ns' | 'ps')? |
        Dec_number ('us' | 'ns' | 'ps')?

Dec_number ::= [0-9]+
```

Constants

Listing A.12: Constants in Dhdl.

```
constant ::=
        'true' | 'false' |
        'high' | 'low'
```

String

Listing A.13: Strings in Dhdl.

```
String ::=
        '"' character* '"'

character ::=
        [^\"] |
        '\' [nrtbf/\"']"
```

Type Names

Listing A.14: Possible type names.

```
wire_type ::= 'Register' | 'Signal'
generic_type ::= 'bool' | 'nat' | 'int'
structure_type ::= Identifier
variable_type ::= wire_type | generic_type | structure_type
```

Expressions

Listing A.15: Syntax definition of logical expressions.

```
logical_expr :=
        logical_un_op logical_expr |
        logical_expr logical_bin_op logical_expr |
        '(' logical_expr ')' |
        value |
        logical_variable |
        module_reference

logical_un_op ::=
        'not'

logical_bin_op ::=
        'and' | 'nand' | 'or' | 'nor' | 'xor' | 'equiv'

logical_variable ::=
        Identifier array? ('.' Identifier array?)?

module_reference ::=
        Identifier generic_parameter_list? parameter_list?

generic_parameter_list ::=
        '<' arithm_expr_list '>'

parameter_list ::=
        '(' logical_expr_list ')'
```

Listing A.16: Syntax definition of multiline expressions.

```
multiline_expr ::=
        logical_expr Newline |
        guarded_expr+

guarded_expr ::=
        clause_list '->' side_effects? logical_expr Newline

clause_list ::=
        comp_expr (',' comp_expr)*

condition ::=
        clause_list |
        '(' clause_list ')'

side_effects ::=
        (logical_expr ',')+
logical_expr_list ::=
        logical_expr (',' logical_expr)*
```

Listing A.17: Syntax definition of arithmetical expressions.

```
arithm_expr ::=
        arithm_un_op arithm_expr |
        arithm_expr arithm_bin_op arithm_expr |
        '(' arithm_expr ')' |
        value |
        arithm_variable |
        macro_access

arithm_un_op ::=
        '-' | 'not' | '!'

arithm_bin_op ::=
        '+' | '-' | '*' | '/' | '%' |
        'and' | 'nand' | 'or' | 'nor' | 'xor' | 'equiv'

arithm_variable ::=
        Identifier array? ('.' Identifier array?)?

macro_access ::=
        Identifier '(' arithm_expr_list ')'

arithm_expr_list ::=
        arithm_expr (',' arithm_expr)*
```

Listing A.18: Syntax definition of comparison expressions.

```
comp_op ::=
        '==' | '!=' | '<' | '<=' | '>' | '>='

comp_expr ::=
        arithm_expr
        (comp_op arithm_expr)*

comp_expr_list ::=
        comp_expr (',' comp_expr)*
```

Listing A.19: Syntax definition of delay expressions.

```
delay_expr ::=
        delay_un_op delay_expr |
        delay_expr delay_bin_op delay_expr |
        '(' delay_expr ')' |
        value |
        delay_measure

delay_un_op ::= '-' | 'not' | '!'

delay_bin_op ::=
        '+' | '-' | '*' | '/' | '%' |  '<=' | '==' |
        '!=' | '>=' | '!' |'>' | '<' | '||'

delay_measure ::=
        '(' delay_variable, delay_variable ')'

delay_variable ::=
        Identifier array? ('.' Identifier array?)*
```

Listing A.20: Syntax definition of array expressions.

```
array ::= '[' array_range ']'
array_range ::= arithm_expr ('to'|'downto' arithm_expr)?
```

Packages

Listing A.21: Package identifier definition.

```
package_identifier ::= Identifier ('.' Identifier)
```

Defines

Listing A.22: Syntax definition of defines.

```
define ::=
        'define' Identifier comp_expr |
        'define' 'logic' Identifier logical_expr
```

Imports

Listing A.23: Syntax definition of imports.

```
import ::=
        'import' package_identifier
```

A.7.2 Hardware Definition

Listing A.24: Syntax definition of hardware definition files.

```
hardware_definition_file ::=
        hardware_definition*

hardware_definition ::=
        import |
        define |
        interface_definition |
        module_definition
```

Module

Listing A.25: Syntax definition of modules.

```
module_definition ::=
        'public'? 'module' Identifier
        module_attributes
        module_body

module_attributes ::=
        ('<' generic_parameters '>')?
        ('(' parameters ')')?
        ('extends' identifier_list)?
        ('implements' identifier_list)?

module_body ::= '{' module_element* '}'

module_element ::=
        port |
        member |
        wire |
        constraint |
        delay
```

Listing A.26: Syntax definition of ports.

```
port ::= ('in'|'out') wire_type array? Identifier
        ('<=' multiline_expr)? Newline
```

Listing A.27: Syntax definition of members.

```
member ::=
        wire_type Identifier array? ('<=' multiline_expr)? Newline |
        structure_type Identifier array? ('<=' logical_expr)? Newline |
        generic_type Identifier '<=' arithm_expr Newline
```

Listing A.28: Syntax definition of wiring code.

```
wire ::=
        ':' single_wire_statement |
        ':' '{' wire_statement* '}'

single_wire_statement ::=
        define_statement_multiline |
        assign_statement_multiline |
        wire_statement

wire_statement ::=
        define_statement |
        assign_statement |
        foreach_statement |
        if_statement
```

Listing A.29: Syntax definition of define statements.

```
define_statement_multiline ::=
        (wire_type | structure_type) logical_variable
                ('<=' multiline_expr)?
                Newline |
        generic_type Identifier
                ('<=' arithm_expr)?
                Newline

define_statement ::=
        (wire_type | structure_type) logical_variable
                ('<=' logical_expr)?
                Newline |
        generic_type Identifier
                ('<=' arithm_expr)?
                Newline
```

Listing A.30: Syntax definition of assignment statements.

```
assign_statement ::=
        logical_variable '<='
```

```
                multiline_expr | arithm_expr
                Newline

assign_statement_ ::=
        logical_variable_multiline '<='
                logical_expr | arithm_expr
                Newline
```

Listing A.31: Syntax definition of foreach statements.

```
foreach_statement ::=
        'foreach' variable_name '[' index ']' '{'
                wire_statement*
        '}' Newline

variable_name ::= Identifier
index ::= Identifier
```

Listing A.32: Syntax definition of if statements.

```
if_statement ::=
        'if' condition '{'
                wire_statement*
        '}' ('else' 'if' clause_list '{'
                wire_statement*
        '}')* ('else' '{'
                wire_statement*
        '}')?  Newline
```

Listing A.33: Syntax definition of delay assertions.

```
delay ::=
        'delay' delay_list Newline |
        'delay' '{' delay_list? '}'

delay_list ::= delay_property (',' delay_property)*

delay_property ::= (Identifier ':=') delay_expr
```

Listing A.34: Syntax definition of constraints.

```
constraint ::=
        ':-' comp_expr_list Newline |
        ':-' '{' comp_expr_list? '}'
```

Interface

Listing A.35: Syntax definition of interfaces.

```
interface_definition ::=
        'public'? 'interface' Identifier
        interface_attributes
        interface_body

interface_attributes ::=
        ('<' generic_parameters '>')?
        ('extends' identifier_list)?

interface_body ::= '{' port* '}'
```

A.7.3 Simulation Definition

Listing A.36: Syntax definition of simulation definition files.

```
simulation_definition_file ::=
        simulation_definition*

simulation_definition ::=
        import |
        define |
        simulation_definition |
```

Listing A.37: Syntax definition of simulations.

```
simulation_definition ::=
        'simulation' Identifier?
        statement_block

statement_block ::=
        '{' simulation_statement* '}'

simulation_statement ::=
        assert |
        variable_definition |
        variable_assign |
        simulator_statement |
        print_statement |
        control_statement

control_statement ::=
        if_statement |
        for_loop |
        foreach_loop |
        do_loop |
        while_loop
```

Assert

Listing A.38: Syntax definition of assert.

```
assert ::=
        'assert' comp_expr_list
                ',' print_statement)?
```

Variable Definition

Listing A.39: Syntax definition of variable definition.

```
variable_definition ::=
        (wire_type | structure_type) logical_variable
                (':=' arithm_expr)?
```

Variable Assign

Listing A.40: Syntax definition of variable assign.

```
variable_assign ::=
        arithm_variable ':=' arithm_expr
```

Simulator

Listing A.41: Syntax definition of simulator statement.

```
simulator_statement ::=
        'simulator' Identifier arithm_expr |
        'simulator' 'add' Identifier ':=' module_reference
```

Print

Listing A.42: Syntax definition of print statement.

```
print_statement ::=
        print_element ('+' print_element)*
print_element ::=
        String |
        arithm_expr
```

If

Listing A.43: Syntax definition of if statements.

```
if_statement ::=
        'if' condition statement_block
        ('else' 'if' condition statement_block)*
        ('else' statement_block)?
```

For

Listing A.44: Syntax definition of for loop.

```
for_loop ::=
        'for' for_expr statement_block

for_expr ::=
        iterator |
        '(' iterator ')'

iterator ::=
        Identifier := array_expr
```

Do

Listing A.45: Syntax definition of do loop.

```
do_loop ::=
        'do' statement_block
                'while' | 'until' condition
```

While

Listing A.46: Syntax definition of while loop.

```
while_loop ::=
        'while' condition statement_block
```

A.8 Translator Guide

This section describes the Dhdl command line interface and its translation procedure. The expression in brackets is the option for the interface. In general, Dhdl's interface is designed to comply with the UNIX/Linux guidelines. For a quick summary of the commands, please see the Dhdl man-page.

A.8.1 Modes

The Dhdl program transforms the input files according to the selected mode. The input files can either be hardware, simulation or package definitions. In each mode all necessary files are read, e.g. the files on which the specified ones depend.

Check (-c)

This mode accepts hardware definition files. The specified hardware descriptions are checked for sanity. The top most module, in this case, cannot have generic parameters without default values.

Generate VHDL (-v)

This mode accepts hardware definition files. The specified hardware descriptions are checked for sanity and, for each module configuration, a VHDL component is generated. Here, the top most module also cannot have generic parameters without default values.

Generate Tcl-Script (-q)

This mode accepts package files. The packages are read and the specified hardware descriptions are checked for sanity and, for each module configuration, a VHDL component is generated. Also, a tcl-script for the external synthesizer is generated.

Simulate (-s)

This mode accepts simulation files. The specified simulations are executed sequentially and modules are checked and translated to the simulation representation as they appear. If one simulation fails, the whole procedure is aborted and an error message returned. To suppress the interpretation of output commands of the simulations “-nso” can be used.

A.8.2 Timing Verification (-d)

When the previously selected mode did not read the package description, it is read for the timing verification separately with all its modules. Then, the timing files are read and the check of the timing properties is performed. With “-ps” a list of accessible signals for timing properties is printed.

A.8.3 More Options

Other than the core features, Dhdl provides more options for convenient usage.

Run Quartus (-rq)

Runs the external synthesizer, Quartus in this version, after the generation of the tcl-script. With this option it is possible to generate the netlist and check its timing with one command line.

Folder Settings

Generated files that belong together are usually grouped within the same folder by Dhdl. The folders are the source folder, the VHDL file folder, the synthesis folder and the timing information folder. These folders can be changed in the command line with the options “–src”, “–vhdl”, “–syn”, “–time” and the new folder to use. Folders for generated files are also generated if they do not exist prior to the execution. All files are overwritten without check back and with the option “-e” it is possible to empty all folders before the translation starts.

Verbose Output

With the option “-v”, Dhdl reports about the current progress in translation and its result.

A.9 Code Examples

Below you find a useful set of code examples for the Dhdl language. In general, these examples are easy to understand and little explanatory text is provided.

A.9.1 Declaring Structures

The general structure in Dhdl is achieved with modules and interfaces.

Modules

Here are some examples of module definitions.

Basic Module A basic module with one input and one output port which are directly connected.

```
module BasicModule {
        in Signal in
        out Signal out <= in
}
```

Module with Generic Parameter This module takes one generic parameter $width$ to determine the array width of the input and output ports.

```
module ModuleWithGenericParameter<nat width> {
        in Signal in[width]
        out Signal out[width] <= in
}
```

Module with Signal Parameters A module with signal parameters can be used within an expression of another module, as the following example shows.

```
module Connect (Signal in) {
        out Signal out <= in
}

module UsingModule {
        in Signal in
        out Signal out <= Connect(in)
}
```

Interfaces

In this section it is shown how to declare and correctly implement an interface in a module.

Basic Interface The following code shows a very simple interface and an implementing module.

```
public interface BasicInterface {
        in Signal inPort
        out Signal outPort
}

public module BasicModule implements BasicInterface {
        in Signal inPort
        out Signal outPort <= inPort
}
```

Passing Generic Parameters to an Interface If the module itself implements only an interface with a specific generic parameter, then one can set this parameter via a member variable within the module having the same name, but not necessarily the same type. Nonetheless, the value must be within the right range as shown in the following example.

```
public interface PassGenericInterface1 <nat width> {
        in Signal inPort[width]
        out Signal outPort[width]
}

public module PassGenericModule1 implements PassGenericInterface1 {
        int width <= 2

        in Signal inPort[2]
        out Signal outPort[2] <= inPort
}
```

If a generic parameter within the module is used, it has to have the same name, then it will be passed on automatically, like in the next example.

```
public interface PassGenericInterface2 <nat width> {
        in Signal inPort[width]
        out Signal outPort[width]
}

public module PassGenericModule2 <bool width <= 1>
                implements PassGenericInterface2 {
        in Signal inPort[width]
        out Signal outPort[width] <= inPort
}
```

A.9.2 Recursion

Recursion is the key element for specifying modules that perform more advanced calculations. Here are some example from the library for using recursion in Dhdl in a simple and clear manner.

Equals The “equals” library element checks whether all signals in the array in are the same value as $value$.

```
module equals(Signal in[n], Signal val) {

        out Signal out <=
                n == 1 -> in equiv val
                -> (in[0] equiv val) and equals(in[1 to n - 1], val)
}
```

Greater or Equals The “greator or equals” library element checks whether $number$ or more signals in the array in are the same value as $value$.

```
import dhdl.base.equals

module grequals<nat number>(Signal in[n] , Signal value) {

        out Signal out <=
                number == 0     -> true
                number > n      -> false
                n == number -> equals (in, value)
                in[0] == value -> grequals<number-1>(in[1 to n-1], value)
                in[0] != value -> grequals<number>(in[1 to n-1], value)
}
```

Is Equal The "isEqual" module of the library checks whether all signals $value1$ and $value2$ are bitwise equivalent.

```
module isEqual(Signal value1[n], Signal value2[m]) {
        :- n == m

        out Signal out <=
                n == 1 ->
                        value1[0] equiv value2[0]

                value1[0] == value2[0] ->
                        isEqual(value1[1 to n-1], value2[1 to n-1])

                -> false
}
```

A.9.3 Interconnect

In hardware designs it is often the case that logic modules of the same type have to be interconnected in a certain manner. Here are some examples how to use wiring code for an efficient solution.

Increment The library module "increment" uses basic cells to increment the value of the input port in. These cells are simple one bit adder and the first one is assigned true on its $carry$ port to increment.

```
module incrementCell {
        in Signal in
        in Signal carry

        out Signal out <= in xor carry
        out Signal carryOut <= in and carry
}

public module increment (Signal in[n]) {

        out Signal out[n]

        incrementCell inc[n]

        : foreach out[i] {
                out[i] <= inc[i].out
                inc[i].in <= in[i]

                if i == 0 {
                        inc[i].carry <= true
                } else {
                        inc[i].carry <= inc[i-1].carryOut
                }
        }
}
```

Full Connect When it is necessary to connect all modules of a certain type with each other, a nested foreach loop is a good method. In this example, all *element* modules have the same pattern of the other elements' output ports at their input ports.

```
module Element<nat n> {
        in Signal in[n]
        out Signal out <= true
}

module FullConnect<nat n <= 5> {
        Element<n> element[n]

        : foreach element[i] {
                foreach element[j] {
                        element[i].in[j] <= element[j].out
                }
        }
}
```

A.9.4 State Machine

In synchronous designs, finite state machines are the core elements. Here is an example state machine encoding the behavior of traffic lights. It has the (one bit) states of the lights as output and a signal *next* as input to determine when to compute the next state.

```
import dhdl.lang

define WIDTH 2

define logic RED              toArray<0, WIDTH>
define logic REDYELLOW        toArray<1, WIDTH>
define logic GREEN            toArray<2, WIDTH>
define logic YELLOW           toArray<3, WIDTH>


module TrafficLights {
        forward in Signal clock
        forward in Signal reset

        in Signal next

        out Signal red <=
                state == RED            -> true
                state == REDYELLOW      -> true
```

```
                                        -> false

        out Signal yellow <=
                state == YELLOW         -> true
                state == REDYELLOW      -> true
                                        -> false

        out Signal green <=
                state == GREEN          -> true
                                        -> false

        Register state[WIDTH] <=
                state == RED, next == true
                        -> REDYELLOW

                state == REDYELLOW, next == true
                        -> GREEN

                state == GREEN, next == true
                        -> YELLOW

                state == YELLOW, next == true
                        -> RED

                -> state
}
```

APPENDIX B

Reference Design Details

This chapter contains the code and, provided it is interesting, also some generated files of the reference designs explained in Chapter 6.

B.1 The Small And-Gate Design

For the small and-gate design, the Dhdl simulation code, the VHDL testbench, the generated timing files and the full semantic interpretation is given here.

B.1.1 Dhdl Simulation Code

Listing B.1: Dhdl simulation code for the and-gate design.

```
import And

simulation andTest {

        simulator add gate := And
        simulator start

        simulator listen gate.in0
        simulator listen gate.in1
        simulator listen gate.out
        simulator listen gate.last

        gate.in0 := false
        gate.in1 := false
        simulator step
        assert gate.out == false

        gate.in0 := true
        gate.in1 := false
        simulator step
        assert gate.out == false
        assert gate.last == false

        gate.in0 := false
        gate.in1 := true
        simulator step
        assert gate.out == false
        assert gate.last == false

        gate.in0 := true
        gate.in1 := true
        simulator step
        assert gate.out == true
        assert gate.last == false

        simulator step
        assert gate.out == true
        assert gate.last == true

        simulator show
}
```

B.1.2 VHDL Testbench Code

Listing B.2: The VHDL code for the testbench of the and-gate design.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.numeric_std.all;

library WORK;
use WORK.all;

entity tb_and is
end entity;

architecture tb_behavior of tb_and is

        component And_0
                port(
                        in0_port : in std_logic;
                        in1_port : in std_logic;
                        out_port : out std_logic;
                        last_port : out std_logic
                );
        end component;

        signal in0_signal : std_logic;
        signal in1_signal : std_logic;
        signal out_signal : std_logic;
        signal last_signal : std_logic;

begin
```

```

        DUT : And_0
                port map(
                        in0_port => in0_signal,
                        in1_port => in1_signal,
                        out_port => out_signal,
                        last_port => last_signal
                        );

        testrun: process

                procedure tick is
                begin
                        wait for 1 ns;
                end tick;

        begin
                tick;

                in0_signal <= '0';
                in1_signal <= '0';
                tick;
                assert out_signal = '0' severity FAILURE;

                in0_signal <= '1';
                in1_signal <= '0';
                tick;
                assert out_signal = '0' severity FAILURE;
                assert last_signal = '0' severity FAILURE;

                in0_signal <= '0';
                in1_signal <= '1';
                tick;
                assert out_signal = '0' severity FAILURE;
                assert last_signal = '0' severity FAILURE;

                in0_signal <= '1';
                in1_signal <= '1';
                tick;
                assert out_signal = '1' severity FAILURE;
                assert last_signal = '1' severity FAILURE;

                tick;
                assert out_signal = '1' severity FAILURE;
                assert last_signal = '1' severity FAILURE;

        end process;

end architecture tb_behavior;
```

B.1.3 Timing Files

Listing B.3: The standard delay format file generated for the and-gate design.

```
// Copyright (C) 1991-2010 Altera Corporation
// Your use of Altera Corporation's design tools, logic functions
// and other software and tools, and its AMPP partner logic
// functions, and any output files from any of the foregoing
// (including device programming or simulation files), and any
// associated documentation or information are expressly subject
// to the terms and conditions of the Altera Program License
// Subscription Agreement, Altera MegaCore Function License
// Agreement, or other applicable license agreement, including,
// without limitation, that your use is for the sole purpose of
// programming logic devices manufactured by Altera and sold by
// Altera or its authorized distributors.  Please refer to the
// applicable agreement for further details.


//
// Device: Altera EP2C35F672C6 Package FBGA672
//

//
// This SDF file should be used for Custom VHDL only
//

(DELAYFILE
  (SDFVERSION "2.1")
  (DESIGN "And_0")
  (DATE "07/14/2011 19:40:13")
  (VENDOR "Altera")
  (PROGRAM "Quartus II")
  (VERSION "Version 9.1 Build 304 01/25/2010 Service Pack 1 SJ Web Edition")
  (DIVIDER .)
  (TIMESCALE 1 ps)

  (CELL
    (CELLTYPE "cycloneii_asynch_io")
    (INSTANCE \\in0_port\~I\\.asynch_inst)
    (DELAY
      (ABSOLUTE
        (IOPATH padio combout (862:862:862) (862:862:862))
      )
    )
  )
  (CELL
    (CELLTYPE "cycloneii_asynch_io")
    (INSTANCE \\in1_port\~I\\.asynch_inst)
    (DELAY
      (ABSOLUTE
        (IOPATH padio combout (842:842:842) (842:842:842))
      )
    )
  )
  (CELL
    (CELLTYPE "cycloneii_lcell_comb")
    (INSTANCE \\calculate_signals\~0\\)
    (DELAY
      (ABSOLUTE
        (PORT datab (5313:5313:5313) (5313:5313:5313))
        (PORT datac (4987:4987:4987) (4987:4987:4987))
        (IOPATH datab combout (419:419:419) (419:419:419))
        (IOPATH datac combout (275:275:275) (275:275:275))
      )
    )
  )
  (CELL
    (CELLTYPE "cycloneii_asynch_io")
    (INSTANCE \\out_port\~I\\.asynch_inst)
    (DELAY
      (ABSOLUTE
        (PORT datain (1902:1902:1902) (1902:1902:1902))
        (IOPATH datain padio (2808:2808:2808) (2808:2808:2808))
      )
    )
  )
  (CELL
    (CELLTYPE "cycloneii_asynch_io")
    (INSTANCE \\last_port\~I\\.asynch_inst)
    (DELAY
      (ABSOLUTE
        (PORT datain (1902:1902:1902) (1902:1902:1902))
        (IOPATH datain padio (2808:2808:2808) (2808:2808:2808))
      )
    )
  )
)
```

Listing B.4: The generated netlist for the use with the standard delay format file for the and-gate design.

```
-- Copyright (C) 1991-2010 Altera Corporation
-- Your use of Altera Corporation's design tools, logic functions
-- and other software and tools, and its AMPP partner logic
-- functions, and any output files from any of the foregoing
-- (including device programming or simulation files), and any
-- associated documentation or information are expressly subject
-- to the terms and conditions of the Altera Program License
-- Subscription Agreement, Altera MegaCore Function License
-- Agreement, or other applicable license agreement, including,
-- without limitation, that your use is for the sole purpose of
-- programming logic devices manufactured by Altera and sold by
-- Altera or its authorized distributors.  Please refer to the
-- applicable agreement for further details.

-- VENDOR "Altera"
-- PROGRAM "Quartus II"
-- VERSION "Version 9.1 Build 304 01/25/2010 Service Pack 1 SJ Web Edition"

-- DATE "07/14/2011 19:40:13"

--
-- Device: Altera EP2C35F672C6 Package FBGA672
--

--
-- This VHDL file should be used for Custom VHDL only
--

LIBRARY CYCLONEII;
LIBRARY IEEE;
USE CYCLONEII.CYCLONEII_COMPONENTS.ALL;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY  And_0 IS
    PORT (
        in0_port : IN std_logic;
        in1_port : IN std_logic;
        out_port : OUT std_logic;
        last_port : OUT std_logic
        );
END And_0;

-- Design Ports Information
-- out_port     =>  Location: PIN_AE22,  I/O Standard: 3.3-V LVTTL,
    Current Strength: 24mA
-- last_port    =>  Location: PIN_AF22,  I/O Standard: 3.3-V LVTTL,
    Current Strength: 24mA
-- in0_port     =>  Location: PIN_G26,   I/O Standard: 3.3-V LVTTL,
    Current Strength: Default
-- in1_port     =>  Location: PIN_N23,   I/O Standard: 3.3-V LVTTL,
    Current Strength: Default

ARCHITECTURE structure OF And_0 IS
SIGNAL gnd : std_logic := '0';
SIGNAL vcc : std_logic := '1';
SIGNAL unknown : std_logic := 'X';
SIGNAL devoe : std_logic := '1';
SIGNAL devclrn : std_logic := '1';
SIGNAL devpor : std_logic := '1';
SIGNAL ww_devoe : std_logic;
SIGNAL ww_devclrn : std_logic;
SIGNAL ww_devpor : std_logic;
SIGNAL ww_in0_port : std_logic;
SIGNAL ww_in1_port : std_logic;
SIGNAL ww_out_port : std_logic;
SIGNAL ww_last_port : std_logic;
SIGNAL \in0_port~combout\ : std_logic;
SIGNAL \in1_port~combout\ : std_logic;
SIGNAL \calculate_signals~0_combout\ : std_logic;

BEGIN
```

```

ww_in0_port <= in0_port;
ww_in1_port <= in1_port;
out_port <= ww_out_port;
last_port <= ww_last_port;
ww_devoe <= devoe;
ww_devclrn <= devclrn;
ww_devpor <= devpor;

-- Location: PIN_G26,    I/O Standard: 3.3-V LVTTL,      Current Strength:
    Default
\in0_port~I\ : cycloneii_io
-- pragma translate_off
GENERIC MAP (
        input_async_reset => "none",
        input_power_up => "low",
        input_register_mode => "none",
        input_sync_reset => "none",
        oe_async_reset => "none",
        oe_power_up => "low",
        oe_register_mode => "none",
        oe_sync_reset => "none",
        operation_mode => "input",
        output_async_reset => "none",
        output_power_up => "low",
        output_register_mode => "none",
        output_sync_reset => "none")
-- pragma translate_on
PORT MAP (
        devclrn => ww_devclrn,
        devpor => ww_devpor,
        devoe => ww_devoe,
        oe => GND,
        padio => ww_in0_port,
        combout => \in0_port~combout\);

-- Location: PIN_N23,    I/O Standard: 3.3-V LVTTL,      Current Strength:
    Default
\in1_port~I\ : cycloneii_io
-- pragma translate_off
GENERIC MAP (
        input_async_reset => "none",
        input_power_up => "low",
        input_register_mode => "none",
        input_sync_reset => "none",
        oe_async_reset => "none",
        oe_power_up => "low",
        oe_register_mode => "none",
        oe_sync_reset => "none",
        operation_mode => "input",
        output_async_reset => "none",
        output_power_up => "low",
        output_register_mode => "none",
        output_sync_reset => "none")
-- pragma translate_on
PORT MAP (
        devclrn => ww_devclrn,
        devpor => ww_devpor,
        devoe => ww_devoe,
        oe => GND,
        padio => ww_in1_port,
        combout => \in1_port~combout\);

-- Location: LCCOMB_X61_Y19_N8
\calculate_signals~0\ : cycloneii_lcell_comb
-- Equation(s):
-- \calculate_signals~0_combout\ = (\in0_port~combout\ & \in1_port~combout\)

-- pragma translate_off
GENERIC MAP (
        lut_mask => "1100000011000000",
        sum_lutc_input => "datac")
-- pragma translate_on
PORT MAP (
        datab => \in0_port~combout\,
        datac => \in1_port~combout\,
        combout => \calculate_signals~0_combout\);

-- Location: PIN_AE22,   I/O Standard: 3.3-V LVTTL,      Current Strength: 24
    mA
\out_port~I\ : cycloneii_io
-- pragma translate_off
GENERIC MAP (
        input_async_reset => "none",
        input_power_up => "low",
        input_register_mode => "none",
        input_sync_reset => "none",
        oe_async_reset => "none",
        oe_power_up => "low",
        oe_register_mode => "none",
        oe_sync_reset => "none",
        operation_mode => "output",
        output_async_reset => "none",
        output_power_up => "low",
        output_register_mode => "none",
        output_sync_reset => "none")
-- pragma translate_on
PORT MAP (
        datain => \calculate_signals~0_combout\,
        devclrn => ww_devclrn,
        devpor => ww_devpor,
        devoe => ww_devoe,
        oe => VCC,
        padio => ww_out_port);

```

```
-- Location: PIN_AF22,    I/O Standard: 3.3-V LVTTL,      Current Strength: 24 mA
\last_port~I\ : cycloneii_io
-- pragma translate_off
GENERIC MAP (
        input_async_reset => "none",
        input_power_up => "low",
        input_register_mode => "none",
        input_sync_reset => "none",
        oe_async_reset => "none",
        oe_power_up => "low",
        oe_register_mode => "none",
        oe_sync_reset => "none",
        operation_mode => "output",
        output_async_reset => "none",
        output_power_up => "low",
        output_register_mode => "none",
        output_sync_reset => "none")
-- pragma translate_on
PORT MAP (
        datain => \calculate_signals~0_combout\,
        devclrn => ww_devclrn,
        devpor => ww_devpor,
        devoe => ww_devoe,
        oe => VCC,
        padio => ww_last_port);
END structure;
```

B.1.4 Semantic Interpretation

For better readability of the semantic interpretation, the Dhdl code will be represented with C^T_{lines}, where $T \in \{S, D\}$ which stand for the and-gate simulation and definition code respectively. The index $lines$ contains the included line numbers comma separated, as well as ranges of line numbers annotated with colons.

1. Initial configuration: $\langle (C^S_{5:38}, \epsilon, \emptyset), (\emptyset, e_i, \emptyset)\rangle$
   - $e_i = \{\text{sim\_start} \mapsto \text{init}, mod\_curr \mapsto \epsilon, And \mapsto C^D_{1,5:14}\}$
2. $\langle (\text{'simulator add gate := And'} : C^S_{6:38}, \epsilon, \emptyset), (\emptyset, e_i, \emptyset)\rangle \overset{\text{[add]}}{\Rightarrow} \langle (C^S_{6:38}, C^D_{1,5:14}, \emptyset), (\emptyset, e_2, \emptyset)\rangle$
   - $e_3 = \{\text{sim\_start} \mapsto \text{init}, mod\_curr \mapsto \text{'gate'}, And \mapsto C^D_{1,5:14}\}$
3. $\dfrac{\langle (C^S_{6:38}, C^D_{5:13}, \emptyset), (\emptyset, e_2, \emptyset)\rangle \Rightarrow^{\text{a to d}} \langle (C^S_{6:38}, \epsilon, T), (s_3, e_2, \emptyset)\rangle}{\langle (C^S_{6:38}, C^D_{1,5:14}, \emptyset), (\emptyset, e_2, \emptyset)\rangle \Rightarrow \langle (C^S_{6:38}, \epsilon, T), (s_3, e_2, \emptyset)\rangle}$ [module]

   a. $\langle (C^S_{6:38}, \text{'in Signal in0'} : C^D_{6:13}, \emptyset), (\emptyset, e_2, \emptyset)\rangle \overset{\text{[input]}}{\Rightarrow} \langle (C^S_{6:38}, C^D_{6:13}, \emptyset), (s_a, e_2, \emptyset)\rangle$
   - $s_a = \{\text{gate.in0} \mapsto \text{undef}\}$

   b. $\langle (C^S_{6:38}, \text{'in Signal in1'} : C^D_{7:13}, \emptyset), (s_a, e_2, \emptyset)\rangle \overset{\text{[input]}}{\Rightarrow} \langle (C^S_{6:38}, C^D_{8:13}, \emptyset), (s_b, e_2, \emptyset)\rangle$

- $s_b = \{\text{gate.in0} \mapsto \text{undef}, \text{gate.in1} \mapsto \text{undef}\}$

c. $\langle (C^S_{6:38}, C^D_{8:11} : C^D_{12:13}, \emptyset), (s_b, e_2, \emptyset)\rangle \overset{\text{[output]}}{\Rightarrow} \langle (C^S_{6:38}, C^D_{13}, T_c), (s_c, e_2, \emptyset)\rangle$

- $s_c = \{\text{gate.in0} \mapsto \text{undef}, \text{gate.in1} \mapsto \text{undef}, \text{gate.out} \mapsto \text{undef}\}$
- $T_c = \{\text{gate.out} \mapsto \text{'in0 == true, in2 == true -> true \textbackslash n -> false'}\}$

d. $\langle (C^S_{6:38}, \text{'out Signal last <= out'}, T_c), (s_c, e_2, \emptyset)\rangle \overset{\text{[output]}}{\Rightarrow} \langle (C^S_{6:38}, \epsilon, T), (s_3, e_2, \emptyset)\rangle$

- $s_3 = \left\{ \begin{array}{ll} \text{gate.in0} \mapsto \text{undef}, & \text{gate.in1} \mapsto \text{undef}, \\ \text{gate.out} \mapsto \text{undef}, & \text{gate.last} \mapsto \text{undef} \end{array} \right\}$
- $T = \left\{ \begin{array}{l} \text{gate.out} \mapsto \text{'in0 == true, in2 == true -> true \textbackslash n -> false'}, \\ \text{gate.last} \mapsto \text{'out'} \end{array} \right\}$

4. $\langle (\text{'simulator start'} : C^S_{7:38}, \epsilon, T), (s_3, e_2, \emptyset)\rangle \overset{\text{[start]}}{\Rightarrow} \langle (C^S_{8:38}, \epsilon, T), (s_3, e_4, \emptyset)\rangle$

   - $e_4 = \{\text{sim\_start} \mapsto \text{listen}, mod\_curr \mapsto \text{'gate'}, And \mapsto C^D_{1,5:14}\}$

5. $\langle (\text{'simulator listen gate.in0'} : C^S_{9:38}, \epsilon, T), (s_3, e_4, \emptyset)\rangle \overset{\text{[listen]}}{\Rightarrow} \langle (C^S_{9:38}, \epsilon, T), (s_3, e_4, h_5)\rangle$

   - $h_5 = \{\text{gate.in0} \mapsto \epsilon\}$

6. $\langle (\text{'simulator listen gate.in1'} : C^S_{10:38}, \epsilon, T), (s_3, e_4, h_5)\rangle \overset{\text{[listen]}}{\Rightarrow} \langle (C^S_{10:38}, \epsilon, T), (s_3, e_4, h_6)\rangle$

   - $h_6 = \{\text{gate.in0} \mapsto \epsilon, \text{gate.in1} \mapsto \epsilon\}$

7. $\langle (\text{'simulator listen gate.out'} : C^S_{11:38}, \epsilon, T), (s_3, e_4, h_6)\rangle \overset{\text{[listen]}}{\Rightarrow} \langle (C^S_{11:38}, \epsilon, T), (s_3, e_4, h_7)\rangle$

   - $h_7 = \{\text{gate.in0} \mapsto \epsilon, \text{gate.in1} \mapsto \epsilon, \text{gate.out} \mapsto \epsilon\}$

8. $\langle (\text{'simulator listen gate.last'} : C^S_{12:38}, \epsilon, T), (s_3, e_4, h_7)\rangle \overset{\text{[listen]}}{\Rightarrow} \langle (C^S_{13:38}, \epsilon, T), (s_3, e_4, h_8)\rangle$

   - $h_8 = \left\{ \begin{array}{ll} \text{gate.in0} \mapsto \epsilon, & \text{gate.in1} \mapsto \epsilon, \\ \text{gate.out} \mapsto \epsilon, & \text{gate.last} \mapsto \epsilon \end{array} \right\}$

9. $\langle (\text{'gate.in0 := false'} : C^S_{14:38}, \epsilon, T), (s_3, e_4, h_7)\rangle \overset{\text{[set]}}{\Rightarrow} \langle (C^S_{14:38}, \epsilon, T), (s_9, e_4, h_8)\rangle$

   - $s_9 = \left\{ \begin{array}{ll} \text{gate.in0} \mapsto \text{false}, & \text{gate.in1} \mapsto \text{undef}, \\ \text{gate.out} \mapsto \text{undef}, & \text{gate.last} \mapsto \text{undef} \end{array} \right\}$

10. $\langle (\text{'gate.in1 := false'} : C^S_{15:38}, \epsilon, T), (s_9, e_4, h_7)\rangle \overset{\text{[set]}}{\Rightarrow} \langle (C^S_{15:38}, \epsilon, T), (s_{10}, e_4, h_8)\rangle$

    - $s_{10} = \left\{ \begin{array}{ll} \text{gate.in0} \mapsto \text{false}, & \text{gate.in1} \mapsto \text{false}, \\ \text{gate.out} \mapsto \text{undef}, & \text{gate.last} \mapsto \text{undef} \end{array} \right\}$

11. $\langle (\text{'simulator step'} : C^S_{16:38}, \epsilon, T), (s_{10}, e_4, h_8)\rangle \overset{\text{[step]}}{\Rightarrow} \langle (C^S_{16:38}, \epsilon, T), (s_{11}, e_{11}, h_{11})\rangle$

- $e_{11} = \{\text{sim\_start} \mapsto \text{run}, mod\_curr \mapsto \text{'gate'}, And \mapsto C^D_{1,5:14}\}$
- $s_{11} = \left\{ \begin{array}{ll} \text{gate.in0} \mapsto \text{false}, & \text{gate.in1} \mapsto \text{false}, \\ \text{gate.out} \mapsto \text{false}, & \text{gate.last} \mapsto \text{undef} \end{array} \right\}$
- $h_{11} = \left\{ \begin{array}{ll} \text{gate.in0} \mapsto \text{'false'}, & \text{gate.in1} \mapsto \text{'false'}, \\ \text{gate.out} \mapsto \text{'undef'}, & \text{gate.last} \mapsto \text{'undef'} \end{array} \right\}$

12. $\langle(\text{'assert gate.out == false'} : C^S_{17:38}, \epsilon, T), (s_{11}, e_{11}, h_{11})\rangle \overset{[\text{assert}^{\text{t}}]}{\Rightarrow}$
$\langle(C^S_{18:38}, \epsilon, T), (s_{11}, e_{11}, h_{11})\rangle$

13. $\langle(\text{'gate.in0 := true'} : C^S_{19:38}, \epsilon, T), (s_{11}, e_{11}, h_{11})\rangle \overset{[\text{step}]}{\Rightarrow} \langle(C^S_{19:38}, \epsilon, T), (s_{12}, e_{11}, h_{11})\rangle$

- $s_{12} = \left\{ \begin{array}{ll} \text{gate.in0} \mapsto \text{true}, & \text{gate.in1} \mapsto \text{false}, \\ \text{gate.out} \mapsto \text{false}, & \text{gate.last} \mapsto \text{undef} \end{array} \right\}$

14. $\langle(\text{'gate.in1 := false'} : C^S_{20:38}, \epsilon, T), (s_{12}, e_{11}, h_{11})\rangle \overset{[\text{set}]}{\Rightarrow} \langle(C^S_{20:38}, \epsilon, T), (s_{12}, e_{11}, h_{11})\rangle$

15. $\langle(\text{'simulator step'} : C^S_{21:38}, \epsilon, T), (s_{12}, e_{11}, h_{11})\rangle \overset{[\text{set}]}{\Rightarrow} \langle(C^S_{21:38}, \epsilon, T), (s_{15}, e_{11}, h_{15})\rangle$

- $s_{15} = \left\{ \begin{array}{ll} \text{gate.in0} \mapsto \text{true}, & \text{gate.in1} \mapsto \text{false}, \\ \text{gate.out} \mapsto \text{false}, & \text{gate.last} \mapsto \text{false} \end{array} \right\}$
- $h_{15} = \left\{ \begin{array}{ll} \text{gate.in0} \mapsto \text{'false:true'}, & \text{gate.in1} \mapsto \text{'false:false'}, \\ \text{gate.out} \mapsto \text{'undef:false'}, & \text{gate.last} \mapsto \text{'undef:undef'} \end{array} \right\}$

16. $\langle(\text{'assert gate.out == false'} : C^S_{22:38}, \epsilon, T), (s_{15}, e_{11}, h_{15})\rangle \overset{[\text{assert}^{\text{t}}]}{\Rightarrow}$
$\langle(C^S_{22:38}, \epsilon, T), (s_{15}, e_{11}, h_{15})\rangle$

17. $\langle(\text{'assert gate.last == false'} : C^S_{23:38}, \epsilon, T), (s_{15}, e_{11}, h_{15})\rangle \overset{[\text{assert}^{\text{t}}]}{\Rightarrow}$
$\langle(C^S_{24:38}, \epsilon, T), (s_{15}, e_{11}, h_{15})\rangle$

18. $\langle(\text{'gate.in0 := false'} : C^S_{25:38}, \epsilon, T), (s_{15}, e_{11}, h_{15})\rangle \overset{[\text{set}]}{\Rightarrow}$
$\langle(C^S_{25:38}, \epsilon, T), (s_{18}, e_{11}, h_{15})\rangle$

- $s_{18} = \left\{ \begin{array}{ll} \text{gate.in0} \mapsto \text{false}, & \text{gate.in1} \mapsto \text{false}, \\ \text{gate.out} \mapsto \text{false}, & \text{gate.last} \mapsto \text{false} \end{array} \right\}$

19. $\langle(\text{'gate.in1 := true'} : C^S_{26:38}, \epsilon, T), (s_{18}, e_{11}, h_{15})\rangle \overset{[\text{set}]}{\Rightarrow}$
$\langle(C^S_{26:38}, \epsilon, T), (s_{19}, e_{11}, h_{15})\rangle$

- $s_{19} = \left\{ \begin{array}{ll} \text{gate.in0} \mapsto \text{false}, & \text{gate.in1} \mapsto \text{true}, \\ \text{gate.out} \mapsto \text{false}, & \text{gate.last} \mapsto \text{false} \end{array} \right\}$

20. $\langle(\text{'simulator step'} : C^S_{27:38}, \epsilon, T), (s_{19}, e_{11}, h_{11})\rangle \overset{[\text{set}]}{\Rightarrow} \langle(C^S_{27:38}, \epsilon, T), (s_{19}, e_{11}, h_{20})\rangle$

- $h_{20} = \left\{ \begin{array}{l} \text{gate.in0} \mapsto \text{'false:true:false'}, \\ \text{gate.in1} \mapsto \text{'false:false:true'}, \\ \text{gate.out} \mapsto \text{'undef:false:false'}, \\ \text{gate.last} \mapsto \text{'undef:undef:false'} \end{array} \right\}$

21. $\langle(\text{'assert gate.out == false'} : C^S_{28:38}, \epsilon, T), (s_{19}, e_{11}, h_{20})\rangle \overset{[\text{assert}^t]}{\Rightarrow}$
$\langle(C^S_{28:38}, \epsilon, T), (s_{19}, e_{11}, h_{20})\rangle$

22. $\langle(\text{'assert gate.last == false'} : C^S_{29:38}, \epsilon, T), (s_{19}, e_{11}, h_{20})\rangle \overset{[\text{assert}^t]}{\Rightarrow}$
$\langle(C^S_{30:38}, \epsilon, T), (s_{19}, e_{11}, h_{20})\rangle$

23. $\langle(\text{'gate.in0 := true'} : C^S_{31:38}, \epsilon, T), (s_{19}, e_{11}, h_{20})\rangle \overset{[\text{set}]}{\Rightarrow} \langle(C^S_{31:38}, \epsilon, T), (s_{23}, e_{11}, h_{20})\rangle$

    - $s_{23} = \left\{ \begin{array}{ll} \text{gate.in0} \mapsto \text{true}, & \text{gate.in1} \mapsto \text{true}, \\ \text{gate.out} \mapsto \text{false}, & \text{gate.last} \mapsto \text{false} \end{array} \right\}$

24. $\langle(\text{'gate.in1 := true'} : C^S_{32:38}, \epsilon, T), (s_{23}, e_{11}, h_{20})\rangle \overset{[\text{set}]}{\Rightarrow} \langle(C^S_{32:38}, \epsilon, T), (s_{23}, e_{11}, h_{20})\rangle$

25. $\langle(\text{'simulator step'} : C^S_{33:38}, \epsilon, T), (s_{23}, e_{11}, h_{20})\rangle \overset{[\text{step}]}{\Rightarrow} \langle(C^S_{33:38}, \epsilon, T), (s_{25}, e_{11}, h_{24})\rangle$

    - $s_{25} = \left\{ \begin{array}{ll} \text{gate.in0} \mapsto \text{true}, & \text{gate.in1} \mapsto \text{true}, \\ \text{gate.out} \mapsto \text{true}, & \text{gate.last} \mapsto \text{false} \end{array} \right\}$
    - $h_{25} = \left\{ \begin{array}{l} \text{gate.in0} \mapsto \text{'false:true:false:true'}, \\ \text{gate.in1} \mapsto \text{'false:false:true:true'}, \\ \text{gate.out} \mapsto \text{'undef:false:false:false'}, \\ \text{gate.last} \mapsto \text{'undef:undef:false:false'} \end{array} \right\}$

26. $\langle(\text{'assert gate.out == true'} : C^S_{34:38}, \epsilon, T), (s_{25}, e_{11}, h_{25})\rangle \overset{[\text{assert}^t]}{\Rightarrow}$
$\langle(C^S_{34:38}, \epsilon, T), (s_{25}, e_{11}, h_{25})\rangle$

27. $\langle(\text{'assert gate.last == false'} : C^S_{35:38}, \epsilon, T), (s_{25}, e_{11}, h_{25})\rangle \overset{[\text{assert}^t]}{\Rightarrow}$
$\langle(C^S_{36:38}, \epsilon, T), (s_{25}, e_{11}, h_{25})\rangle$

28. $\langle(\text{'simulator step'} : C^S_{37:38}, \epsilon, T), (s_{25}, e_{11}, h_{25})\rangle \overset{[\text{step}]}{\Rightarrow} \langle(C^S_{37:38}, \epsilon, T), (s_{28}, e_{11}, h_{28})\rangle$

    - $s_{28} = \left\{ \begin{array}{ll} \text{gate.in0} \mapsto \text{true}, & \text{gate.in1} \mapsto \text{true}, \\ \text{gate.out} \mapsto \text{true}, & \text{gate.last} \mapsto \text{true} \end{array} \right\}$
    - $h_{28} = \left\{ \begin{array}{l} \text{gate.in0} \mapsto \text{'false:true:false:true:true'}, \\ \text{gate.in1} \mapsto \text{'false:false:true:true:true'}, \\ \text{gate.out} \mapsto \text{'undef:false:false:false:true'}, \\ \text{gate.last} \mapsto \text{'undef:undef:false:false:false'} \end{array} \right\}$

29. $\langle(\text{'assert gate.out == true'} : C^S_{38}, \epsilon, T), (s_{28}, e_{11}, h_{28})\rangle \overset{[\text{assert}^t]}{\Rightarrow}$
$\langle(C^S_{38}, \epsilon, T), (s_{28}, e_{11}, h_{28})\rangle$

30. $\langle(\text{'assert gate.last == true'}, \epsilon, T), (s_{28}, e_{11}, h_{28})\rangle \overset{[\text{assert}^t]}{\Rightarrow}$
$\langle(\epsilon, \epsilon, T), (s_{28}, e_{11}, h_{28})\rangle$

31. $\langle(\epsilon, \epsilon, T), (s_{28}, e_{11}, h_{28})\rangle \overset{[\text{term}]}{\Rightarrow} \langle\text{TERM}, h\rangle$

- $h = \left\{ \begin{array}{l} \text{gate.in0} \mapsto \text{'false:true:false:true:true:true'}, \\ \text{gate.in1} \mapsto \text{'false:false:true:true:true:true'}, \\ \text{gate.out} \mapsto \text{'undef:false:false:false:true:true'}, \\ \text{gate.last} \mapsto \text{'undef:undef:false:false:false:true'} \end{array} \right\}$

B.2 Synchronous Design

The synchronous design, a coffee machine, consists of five Dhdl files. CoffeeFSM.df contains the finite state machine definition and CoffeeFSM.sf, its simulation test cases. In Coffee.df, the finite state machine is wrapped. There button debouncers are added and the *cupRemoved* signal is set to true to limit the buttons used. Also, further modules are connected to make a one tick long high visible on LEDs for the human eye. In Machine.df, a port for a reset indicator LED is added and all other ports are directly connected. Finally, Machine.pf specifies the pin assignment and the used FPGA device. A state diagram and a detailed specification of the tasks performed can be found in Section B.2.

Listing B.5: Hardware specification of the coffee finite state machine.

```
import dhdl.lang

define logic Idle                toArray<0, 2>
define logic WaitSelect          toArray<1, 2>
define logic WaitBeverage        toArray<2, 2>
define logic WaitForTakeOut      toArray<3, 2>

public module CoffeeFSM {
        forward in Signal clock
        forward in Signal reset

        in Signal coinAvailable
        in Signal btnCoffee
        in Signal btnTea
        in Signal btnAbort
        in Signal cupRemoved
        in Signal beverageReady

        out Register makeCoffee <= false
        out Register makeTea <= false
        out Register returnCoin <= false
        out Register beep <= false

        Register state[2] <=
                Idle == state, coinAvailable ->
                        WaitSelect

                WaitSelect == state, btnCoffee ->
                        makeCoffee <= true,
                        WaitBeverage

                WaitSelect == state, btnTea ->
                        makeTea <= true,
                        WaitBeverage

                WaitSelect == state, btnAbort ->
                        returnCoin <= true,
                        Idle

                WaitBeverage == state, beverageReady ->
                        beep <= true,
                        WaitForTakeOut

                WaitForTakeOut == state, cupRemoved ->
                        Idle

                WaitForTakeOut == state ->
                        beep <= true,
                        WaitForTakeOut

                -> state
}
```

Listing B.6: Simulation specification for the coffee selection of the coffee finite state machine.

```
import coffee.CoffeeFSM

simulation coffee {

        simulator add machine := CoffeeFSM
        simulator start

        simulator listen machine.clock
        simulator listen machine.reset

        simulator listen machine.state [0 to 1]

        simulator listen machine.coinAvailable
        simulator listen machine.btnCoffee
        simulator listen machine.btnTea
        simulator listen machine.btnAbort
        simulator listen machine.cupRemoved
        simulator listen machine.beverageReady

        simulator listen machine.makeCoffee
        simulator listen machine.makeTea
        simulator listen machine.returnCoin
        simulator listen machine.beep


        machine.clock := false
        machine.reset := true

        machine.coinAvailable := false
        machine.btnCoffee := false
        machine.btnTea := false
        machine.btnAbort := false
        machine.cupRemoved := false
        machine.beverageReady := false

        simulator step
        machine.reset := false
        simulator step

        assert machine.state[0 to 1] == 0
        assert machine.makeCoffee == 0
        assert machine.makeTea == 0
        assert machine.returnCoin == 0
        assert machine.beep == 0


        machine.clock := not machine.clock
        simulator step

        assert machine.state[0 to 1] == 0
        assert machine.makeCoffee == 0
        assert machine.makeTea == 0
        assert machine.returnCoin == 0
        assert machine.beep == 0

        machine.coinAvailable := true

        machine.clock := not machine.clock
        simulator step
        machine.clock := not machine.clock
        simulator step

        assert machine.state[0 to 1] == 1
        assert machine.makeCoffee == 0
        assert machine.makeTea == 0
        assert machine.returnCoin == 0
        assert machine.beep == 0

        machine.coinAvailable := false

        machine.clock := not machine.clock
        simulator step
        machine.clock := not machine.clock
        simulator step

        assert machine.state[0 to 1] == 1
        assert machine.makeCoffee == 0
        assert machine.makeTea == 0
        assert machine.returnCoin == 0
        assert machine.beep == 0

        machine.btnCoffee := true

        machine.clock := not machine.clock
        simulator step
        machine.clock := not machine.clock
        simulator step

        assert machine.state[0 to 1] == 2
        assert machine.makeCoffee == 1
        assert machine.makeTea == 0
        assert machine.returnCoin == 0
        assert machine.beep == 0

        machine.btnCoffee := false

        machine.clock := not machine.clock
        simulator step
        machine.clock := not machine.clock
        simulator step

        assert machine.state[0 to 1] == 2
        assert machine.makeCoffee == 0
        assert machine.makeTea == 0
        assert machine.returnCoin == 0
        assert machine.beep == 0

        machine.clock := not machine.clock
        simulator step
        machine.clock := not machine.clock
        simulator step

        assert machine.state[0 to 1] == 2
        assert machine.makeCoffee == 0
        assert machine.makeTea == 0
        assert machine.returnCoin == 0
        assert machine.beep == 0

        machine.beverageReady := true

        machine.clock := not machine.clock
        simulator step
        machine.clock := not machine.clock
        simulator step

        assert machine.state[0 to 1] == 3
        assert machine.makeCoffee == 0
        assert machine.makeTea == 0
        assert machine.returnCoin == 0
        assert machine.beep == 1

        machine.beverageReady := false

        machine.clock := not machine.clock
        simulator step
        machine.clock := not machine.clock
        simulator step

        assert machine.state[0 to 1] == 3
        assert machine.makeCoffee == 0
        assert machine.makeTea == 0
        assert machine.returnCoin == 0
        assert machine.beep == 1

        machine.clock := not machine.clock
        simulator step
        machine.clock := not machine.clock
        simulator step

        assert machine.state[0 to 1] == 3
        assert machine.makeCoffee == 0
        assert machine.makeTea == 0
        assert machine.returnCoin == 0
        assert machine.beep == 1

        machine.cupRemoved := true

        machine.clock := not machine.clock
        simulator step
        machine.clock := not machine.clock
        simulator step

        assert machine.state[0 to 1] == 0
        assert machine.makeCoffee == 0
        assert machine.makeTea == 0
        assert machine.returnCoin == 0
        assert machine.beep == 0

        machine.cupRemoved := false

        machine.clock := not machine.clock
        simulator step
        machine.clock := not machine.clock
        simulator step

        assert machine.state[0 to 1] == 0
        assert machine.makeCoffee == 0
        assert machine.makeTea == 0
        assert machine.returnCoin == 0
        assert machine.beep == 0

        simulator show
}
```

Listing B.7: Wrapper module for the coffee finite state machine.

```
import dhdl.util.Button
import dhdl.util.StretchImpulse
import dhdl.util.DelayImpulse

import coffee.CoffeeFSM

public module Coffee <nat inCycles <= 128, nat outCycles <= 128>{
        forward in Signal clock
        forward in Signal reset

        in Register tasterCoin
        in Register tasterCoffee
        in Register tasterTea
        in Register tasterAbort

        out Signal ledMakeCoffee <= StretchImpulse<outCycles>(coffeeFSM.
            makeCoffee)
        out Signal ledMakeTea <= StretchImpulse<outCycles>(coffeeFSM.makeTea)
        out Signal ledReturnCoin <= StretchImpulse<outCycles>(coffeeFSM.
            returnCoin)
        out Signal ledReady <= StretchImpulse<outCycles>(coffeeFSM.beep)
        out Signal ledCoin <= StretchImpulse<outCycles>(btnCoin.pressed)


        Button<inCycles> btnCoin
        : btnCoin.in <= tasterCoin

        Button<inCycles> btnCoffee
        : btnCoffee.in <= tasterCoffee

        Button<inCycles> btnTea
        : btnTea.in <= tasterTea

        Button<inCycles> btnAbort
        : btnAbort.in <= tasterAbort

        CoffeeFSM coffeeFSM

        : coffeeFSM.coinAvailable <= btnCoin.pressed
        : coffeeFSM.btnCoffee <= btnCoffee.pressed
        : coffeeFSM.btnTea <= btnTea.pressed
        : coffeeFSM.btnAbort <= btnAbort.pressed
        // No sensor for cup removed:
        : coffeeFSM.cupRemoved <= true
        : coffeeFSM.beverageReady <= DelayImpulse<outCycles>(coffeeFSM.
            makeCoffee
                or coffeeFSM.makeTea)
}
```

Listing B.8: Wrapper module for the coffee module.

```

import coffee.Coffee

define inDelay 25000
define outHold 25000000

public module Machine {

        forward in Signal clock
        forward in Signal reset

        Coffee<inDelay, outHold> coffee

        in Signal tasterCoin
        : coffee.tasterCoin <= not tasterCoin

        in Signal tasterCoffee
        : coffee.tasterCoffee <= not tasterCoffee

        in Signal tasterTea
        : coffee.tasterTea <= not tasterTea

        in Signal tasterAbort
        : coffee.tasterAbort <= not tasterAbort

        out Signal ledCoin <= coffee.ledCoin
        out Signal ledMakeCoffee <= coffee.ledMakeCoffee
        out Signal ledMakeTea <= coffee.ledMakeTea
        out Signal ledReturnCoin <= coffee.ledReturnCoin
        out Signal ledReady <= coffee.ledReady

        out Signal resetLed <= reset
//      out Signal gnd <= false

}
```

Listing B.9: Package file for the coffee machine.

```
// Package description for Coffee Machine
//

component       := coffee.Machine

family          := "Cyclone II"
device          := EP2C35F672C6

pin     Machine.clock           := PIN_D13
pin     Machine.reset           := PIN_V2

pin     Machine.resetLed        := PIN_AD12

pin     Machine.tasterCoin      := PIN_G26
pin     Machine.tasterCoffee    := PIN_N23
pin     Machine.tasterTea       := PIN_P23
pin     Machine.tasterAbort     := PIN_W26

pin     Machine.ledCoin         := PIN_AE22
pin     Machine.ledMakeCoffee   := PIN_AF22
pin     Machine.ledMakeTea      := PIN_W19
pin     Machine.ledReturnCoin   := PIN_V18
pin     Machine.ledReady        := PIN_U18

//pin   Machine.gnd      := PIN_K25
```

B.3 Asynchronous Design

This section contains the listings of the DARTS timing analysis.

B.3.1 Timing Constraints

Listing B.10 shows the Dhdl output of the timing constraint analysis for DARTS.

Listing B.10: Dhdl timing analysis output for the timing constraints.

```
Time Reader: parse time/DartsNetwork_vhd.sdo
Time Reader: parse time/DartsNetwork.vho
[line 8, char 17]: Delay tdiffm := (diff[0].celem[0].bit[0].invOut
    [0], diff[0].celem[0].bit[0].invOut[0]) on path "clock[0].
    counterModules[0]" is satisfied. [7.897E-9, 7.897E-9]
[line 8, char 17]: Delay tdiffm := (diff[0].celem[0].bit[0].invOut
    [0], diff[0].celem[0].bit[0].invOut[0]) on path "clock[0].
    counterModules[1]" is satisfied. [7.797E-9, 7.797E-9]
[line 8, char 17]: Delay tdiffm := (diff[0].celem[0].bit[0].invOut
    [0], diff[0].celem[0].bit[0].invOut[0]) on path "clock[0].
    counterModules[2]" is satisfied. [7.538E-9, 7.538E-9]
[line 8, char 17]: Delay tdiffm := (diff[0].celem[0].bit[0].invOut
    [0], diff[0].celem[0].bit[0].invOut[0]) on path "clock[0].
    counterModules[3]" is satisfied. [7.358E-9, 7.358E-9]
dhdl/darts/DartsClock.df[line 8, char 195]: Delay tdis := (((
    thresholdGates[0].celem[0].bit[0].out[1], counterModules[0].local
    [0].celem[0].bit[0].out[1]) + (counterModules[0].local[0].celem
    [0].bit[0].out[1], thresholdGates[0].celem[0].bit[0].out[1]))
    PARALLEL (((thresholdGates[0].celem[0].bit[0].out[1],
    counterModules[1].local[0].celem[0].bit[0].out[1]) + (
    counterModules[1].local[0].celem[0].bit[0].out[1], thresholdGates
    [0].celem[0].bit[0].out[1])) PARALLEL (((thresholdGates[0].celem
    [0].bit[0].out[1], counterModules[2].local[0].celem[0].bit[0].out
    [1]) + (counterModules[2].local[0].celem[0].bit[0].out[1],
    thresholdGates[0].celem[0].bit[0].out[1])) PARALLEL ((
    thresholdGates[0].celem[0].bit[0].out[1], counterModules[3].local
    [0].celem[0].bit[0].out[1]) + (counterModules[3].local[0].celem
    [0].bit[0].out[1], thresholdGates[0].celem[0].bit[0].out[1])))))
```

```
   on path "clock[0]" is satisfied. [1.6809999999999999E-9,
   2.8685000000000002E-8]
8 dhdl/darts/DartsClock.df[line 8, char 195]: Delay tdis := (((
   thresholdGates[0].celem[0].bit[0].out[1], counterModules[0].local
   [0].celem[0].bit[0].out[1]) + (counterModules[0].local[0].celem
   [0].bit[0].out[1], thresholdGates[0].celem[0].bit[0].out[1]))
   PARALLEL (((thresholdGates[0].celem[0].bit[0].out[1],
   counterModules[1].local[0].celem[0].bit[0].out[1]) + (
   counterModules[1].local[0].celem[0].bit[0].out[1], thresholdGates
   [0].celem[0].bit[0].out[1])) PARALLEL (((thresholdGates[0].celem
   [0].bit[0].out[1], counterModules[2].local[0].celem[0].bit[0].out
   [1]) + (counterModules[2].local[0].celem[0].bit[0].out[1],
   thresholdGates[0].celem[0].bit[0].out[1])) PARALLEL ((
   thresholdGates[0].celem[0].bit[0].out[1], counterModules[3].local
   [0].celem[0].bit[0].out[1]) + (counterModules[3].local[0].celem
   [0].bit[0].out[1], thresholdGates[0].celem[0].bit[0].out[1])))))
   on path "clock[0]" is satisfied. [1.6809999999999999E-9,
   2.8685000000000002E-8]
9 [line 13, char 18]: Delay tmindis := min(tdis) on path "clock[0]"
   is satisfied. [1.6809999999999999E-9, 1.6809999999999999E-9]
10 [line 14, char 18]: Delay tmaxdis := max(tdis) on path "clock[0]"
   is satisfied. [2.8685000000000002E-8, 2.8685000000000002E-8]
11 [line 13, char 18]: Delay tmindis := min(tdis) on path "clock[0]"
   is satisfied. [1.6809999999999999E-9, 1.6809999999999999E-9]
12 [line 8, char 17]: Delay tdiffm := (diff[0].celem[0].bit[0].invOut
   [0], diff[0].celem[0].bit[0].invOut[0]) on path "clock[0].
   counterModules[0]" is satisfied. [7.897E-9, 7.897E-9]
13 dhdl/darts/DartsClock.df[line 19, char 42]: Delay tdiff := (
   counterModules[0] tdiffm PARALLEL (counterModules[1] tdiffm
   PARALLEL (counterModules[2] tdiffm PARALLEL counterModules[3]
   tdiffm))) on path "clock[0]" is satisfied. [7.897E-9, 7.897E-9]
14 [line 24, char 19]: Delay tdiffmin := min(tdiff) on path "clock
   [0]" is satisfied. [7.897E-9, 7.897E-9]
15 dhdl/darts/DartsClock.df[line 15, char 23]: Delay tmin := (tmindis
    + tdiffmin) on path "clock[0]" is satisfied. [9.578E-9, 9.578E-9]
16 [line 14, char 18]: Delay tmaxdis := max(tdis) on path "clock[0]"
   is satisfied. [2.8685000000000002E-8, 2.8685000000000002E-8]
17 dhdl/darts/DartsClock.df[line 16, char 15]: Delay tloc := tmaxdis
   on path "clock[0]" is satisfied. [2.8685000000000002E-8,
   2.8685000000000002E-8]
18 [line 25, char 19]: Delay tdiffmax := max(tdiff) on path "clock
   [0]" is satisfied. [7.897E-9, 7.897E-9]
19 dhdl/darts/DartsClock.df[line 15, char 23]: Delay tmin := (tmindis
    + tdiffmin) on path "clock[0]" is satisfied. [9.578E-9, 9.578E-9]
20 error 1 dhdl/darts/DartsClock.df[line 27, char 31]: Delay not
   satisfied interlocking := (tmaxdis <= ((tmin * 2) + tmindis)) on
   path "clock[0]". The resulting delay is [2.0837E-8,
   2.8685000000000002E-8].
```

```
[line 25, char 19]: Delay tdiffmax := max(tdiff) on path "clock
 [0]" is satisfied. [7.897E-9, 7.897E-9]
dhdl/darts/DartsClock.df[line 28, char 31]: Delay tickRemoval := (
 tdiffmax <= tmin) on path "clock[0]" is satisfied. [7.897E-9,
 9.578E-9]
dhdl/darts/DartsClock.df[line 16, char 15]: Delay tloc := tmaxdis
 on path "clock[1]" is satisfied. [2.8685000000000002E-8,
 2.8685000000000002E-8]
error 2 dhdl/darts/DartsClock.df[line 27, char 31]: Delay not
 satisfied interlocking := (tmaxdis <= ((tmin * 2) + tmindis)) on
 path "clock[1]". The resulting delay is [2.0837E-8,
 2.8685000000000002E-8].
dhdl/darts/DartsClock.df[line 28, char 31]: Delay tickRemoval := (
 tdiffmax <= tmin) on path "clock[1]" is satisfied. [7.897E-9,
 9.578E-9]
dhdl/darts/DartsClock.df[line 16, char 15]: Delay tloc := tmaxdis
 on path "clock[2]" is satisfied. [2.8685000000000002E-8,
 2.8685000000000002E-8]
error 3 dhdl/darts/DartsClock.df[line 27, char 31]: Delay not
 satisfied interlocking := (tmaxdis <= ((tmin * 2) + tmindis)) on
 path "clock[2]". The resulting delay is [2.0837E-8,
 2.8685000000000002E-8].
dhdl/darts/DartsClock.df[line 28, char 31]: Delay tickRemoval := (
 tdiffmax <= tmin) on path "clock[2]" is satisfied. [7.897E-9,
 9.578E-9]
dhdl/darts/DartsClock.df[line 16, char 15]: Delay tloc := tmaxdis
 on path "clock[3]" is satisfied. [2.8685000000000002E-8,
 2.8685000000000002E-8]
error 4 dhdl/darts/DartsClock.df[line 27, char 31]: Delay not
 satisfied interlocking := (tmaxdis <= ((tmin * 2) + tmindis)) on
 path "clock[3]". The resulting delay is [2.0837E-8,
 2.8685000000000002E-8].
dhdl/darts/DartsClock.df[line 28, char 31]: Delay tickRemoval := (
 tdiffmax <= tmin) on path "clock[3]" is satisfied. [7.897E-9,
 9.578E-9]
dhdl/darts/DartsClock.df[line 16, char 15]: Delay tloc := tmaxdis
 on path "clock[4]" is satisfied. [2.8685000000000002E-8,
 2.8685000000000002E-8]
error 5 dhdl/darts/DartsClock.df[line 27, char 31]: Delay not
 satisfied interlocking := (tmaxdis <= ((tmin * 2) + tmindis)) on
 path "clock[4]". The resulting delay is [2.0837E-8,
 2.8685000000000002E-8].
dhdl/darts/DartsClock.df[line 28, char 31]: Delay tickRemoval := (
 tdiffmax <= tmin) on path "clock[4]" is satisfied. [7.897E-9,
 9.578E-9]
dhdl/darts/DartsClock.df[line 16, char 15]: Delay tloc := tmaxdis
 on path "clock[1]" is satisfied. [2.8685000000000002E-8,
 2.8685000000000002E-8]
```

```
error 6 [line 29, char 102]: Delay not satisfied delay 1 := ((
   clock[0].thresholdGates[0].celem[0].bit[0].out[1], clock[1].
   thresholdGates[0].celem[0].bit[0].out[1]) >= clock[1] tloc) on
   path "base component". The resulting delay is [6.572E-9,
   4.0923999999999999E-8].
error 7 [line 29, char 102]: Delay not satisfied delay 2 := ((
   clock[0].thresholdGates[0].celem[0].bit[0].out[1], clock[2].
   thresholdGates[0].celem[0].bit[0].out[1]) >= clock[2] tloc) on
   path "base component". The resulting delay is [6.788E-9,
   4.0836999999999984E-8].
error 8 [line 29, char 102]: Delay not satisfied delay 3 := ((
   clock[0].thresholdGates[0].celem[0].bit[0].out[1], clock[3].
   thresholdGates[0].celem[0].bit[0].out[1]) >= clock[3] tloc) on
   path "base component". The resulting delay is [7.366E-9, 4.6206E
   -8].
error 9 [line 29, char 102]: Delay not satisfied delay 4 := ((
   clock[0].thresholdGates[0].celem[0].bit[0].out[1], clock[4].
   thresholdGates[0].celem[0].bit[0].out[1]) >= clock[4] tloc) on
   path "base component". The resulting delay is [7.716000000000001E
   -9, 3.6886E-8].
error 10 [line 29, char 102]: Delay not satisfied delay 5 := ((
   clock[1].thresholdGates[0].celem[0].bit[0].out[1], clock[0].
   thresholdGates[0].celem[0].bit[0].out[1]) >= clock[0] tloc) on
   path "base component". The resulting delay is [5.870999999999999E
   -9, 3.6154E-8].
error 11 [line 29, char 102]: Delay not satisfied delay 6 := ((
   clock[1].thresholdGates[0].celem[0].bit[0].out[1], clock[2].
   thresholdGates[0].celem[0].bit[0].out[1]) >= clock[2] tloc) on
   path "base component". The resulting delay is [6.761E-9, 5.0526E
   -8].
error 12 [line 29, char 102]: Delay not satisfied delay 7 := ((
   clock[1].thresholdGates[0].celem[0].bit[0].out[1], clock[3].
   thresholdGates[0].celem[0].bit[0].out[1]) >= clock[3] tloc) on
   path "base component". The resulting delay is [6.765E-9, 5.1507E
   -8].
error 13 [line 29, char 102]: Delay not satisfied delay 8 := ((
   clock[1].thresholdGates[0].celem[0].bit[0].out[1], clock[4].
   thresholdGates[0].celem[0].bit[0].out[1]) >= clock[4] tloc) on
   path "base component". The resulting delay is [5.491E-9,
   4.6947000000000003E-8].
error 14 [line 29, char 102]: Delay not satisfied delay 9 := ((
   clock[2].thresholdGates[0].celem[0].bit[0].out[1], clock[0].
   thresholdGates[0].celem[0].bit[0].out[1]) >= clock[0] tloc) on
   path "base component". The resulting delay is [7.064E-9, 4.2492E
   -8].
error 15 [line 29, char 102]: Delay not satisfied delay 10 := ((
   clock[2].thresholdGates[0].celem[0].bit[0].out[1], clock[1].
   thresholdGates[0].celem[0].bit[0].out[1]) >= clock[1] tloc) on
   path "base component". The resulting delay is [5.692E-9, 4.3329E
   -8].
error 16 [line 29, char 102]: Delay not satisfied delay 11 := ((
   clock[2].thresholdGates[0].celem[0].bit[0].out[1], clock[3].
   thresholdGates[0].celem[0].bit[0].out[1]) >= clock[3] tloc) on
   path "base component". The resulting delay is [1.4561000000000002E
   -8, 5.0119999999999995E-8].
error 17 [line 29, char 102]: Delay not satisfied delay 12 := ((
   clock[2].thresholdGates[0].celem[0].bit[0].out[1], clock[4].
   thresholdGates[0].celem[0].bit[0].out[1]) >= clock[4] tloc) on
   path "base component". The resulting delay is [1.4911000000000004E
   -8, 4.0844999999999999E-8].
error 18 [line 29, char 102]: Delay not satisfied delay 13 := ((
   clock[3].thresholdGates[0].celem[0].bit[0].out[1], clock[0].
   thresholdGates[0].celem[0].bit[0].out[1]) >= clock[0] tloc) on
   path "base component". The resulting delay is [1.4583999999999997E
   -8, 4.1457E-8].
error 19 [line 29, char 102]: Delay not satisfied delay 14 := ((
   clock[3].thresholdGates[0].celem[0].bit[0].out[1], clock[1].
   thresholdGates[0].celem[0].bit[0].out[1]) >= clock[1] tloc) on
   path "base component". The resulting delay is [1.5051999999999998E
   -8, 4.7576999999999999E-8].
error 20 [line 29, char 102]: Delay not satisfied delay 15 := ((
   clock[3].thresholdGates[0].celem[0].bit[0].out[1], clock[2].
   thresholdGates[0].celem[0].bit[0].out[1]) >= clock[2] tloc) on
   path "base component". The resulting delay is [1.4829999999999996E
   -8, 4.3352E-8].
error 21 [line 29, char 102]: Delay not satisfied delay 16 := ((
   clock[3].thresholdGates[0].celem[0].bit[0].out[1], clock[4].
   thresholdGates[0].celem[0].bit[0].out[1]) >= clock[4] tloc) on
   path "base component". The resulting delay is [7.862E-9,
   2.8685000000000002E-8].
error 22 [line 29, char 102]: Delay not satisfied delay 17 := ((
   clock[4].thresholdGates[0].celem[0].bit[0].out[1], clock[0].
   thresholdGates[0].celem[0].bit[0].out[1]) >= clock[0] tloc) on
   path "base component". The resulting delay is [7.929E-9,
   3.2628999999999994E-8].
error 23 [line 29, char 102]: Delay not satisfied delay 18 := ((
   clock[4].thresholdGates[0].celem[0].bit[0].out[1], clock[1].
   thresholdGates[0].celem[0].bit[0].out[1]) >= clock[1] tloc) on
   path "base component". The resulting delay is [8.396999999999999E
   -9, 3.874899999999999E-8].
error 24 [line 29, char 102]: Delay not satisfied delay 19 := ((
   clock[4].thresholdGates[0].celem[0].bit[0].out[1], clock[2].
   thresholdGates[0].celem[0].bit[0].out[1]) >= clock[2] tloc) on
   path "base component". The resulting delay is [8.174999999999999E
   -9, 3.4524E-8].
error 25 [line 29, char 102]: Delay not satisfied delay 20 := ((
   clock[4].thresholdGates[0].celem[0].bit[0].out[1], clock[3].
   thresholdGates[0].celem[0].bit[0].out[1]) >= clock[3] tloc) on
```

```
path "base component". The resulting delay is [7.921E-9, 4.5092E-8].
25 errors 0 warnings 0 assertion violations
```

Clock Frequency

Listing B.11 shows the clock timing analysis output of Dhdl for the Darts design.

Listing B.11: The Darts clock timing analysis output.

```
Time Reader: parse time/DartsNetwork_vhd.sdo
Time Reader: parse time/DartsNetwork.vho
[line 34, char 10]: Delay delay 1 := (clock[0].thresholdGates[0].celem[0].bit[0].out[1], clock[1].counterModules[0].remote[0].celem[3].bit[0].out[1]) on path "base component" is satisfied. [4.1539999999999995E-9, 1.6505E-8]
[line 34, char 10]: Delay delay 2 := (clock[0].thresholdGates[0].celem[0].bit[0].out[1], clock[2].counterModules[0].remote[0].celem[3].bit[0].out[1]) on path "base component" is satisfied. [4.57E-9, 1.631E-8]
[line 34, char 10]: Delay delay 3 := (clock[0].thresholdGates[0].celem[0].bit[0].out[1], clock[3].counterModules[0].remote[0].celem[3].bit[0].out[1]) on path "base component" is satisfied. [4.748E-9, 1.5312E-8]
[line 34, char 10]: Delay delay 4 := (clock[0].thresholdGates[0].celem[0].bit[0].out[1], clock[4].counterModules[0].remote[0].celem[3].bit[0].out[1]) on path "base component" is satisfied. [5.408E-9, 1.4153E-8]
[line 26, char 10]: Delay delay 5 := (clock[1].thresholdGates[0].celem[0].bit[0].out[1], clock[0].counterModules[0].remote[0].celem[3].bit[0].out[1]) on path "base component" is satisfied. [3.961999999999999E-9, 1.35E-8]
[line 34, char 10]: Delay delay 6 := (clock[1].thresholdGates[0].celem[0].bit[0].out[1], clock[2].counterModules[1].remote[0].celem[3].bit[0].out[1]) on path "base component" is satisfied. [5.061999999999999E-9, 1.6617E-8]
[line 34, char 10]: Delay delay 7 := (clock[1].thresholdGates[0].celem[0].bit[0].out[1], clock[3].counterModules[1].remote[0].celem[3].bit[0].out[1]) on path "base component" is satisfied. [4.694E-9, 1.4535999999999999E-8]
[line 34, char 10]: Delay delay 8 := (clock[1].thresholdGates[0].celem[0].bit[0].out[1], clock[4].counterModules[1].remote[0].celem[3].bit[0].out[1]) on path "base component" is satisfied. [3.559E-9, 1.5388E-8]
[line 26, char 10]: Delay delay 9 := (clock[2].thresholdGates[0].celem[0].bit[0].out[1], clock[0].counterModules[1].remote[0].celem[3].bit[0].out[1]) on path "base component" is satisfied. [5.190999999999999E-9, 1.5436E-8]
[line 26, char 10]: Delay delay 10 := (clock[2].thresholdGates[0].celem[0].bit[0].out[1], clock[1].counterModules[1].remote[0].celem[3].bit[0].out[1]) on path "base component" is satisfied. [3.815E-9, 1.6096E-8]
[line 34, char 10]: Delay delay 11 := (clock[2].thresholdGates[0].celem[0].bit[0].out[1], clock[3].counterModules[2].remote[0].celem[3].bit[0].out[1]) on path "base component" is satisfied. [4.95E-9, 1.6341E-8]
[line 34, char 10]: Delay delay 12 := (clock[2].thresholdGates[0].celem[0].bit[0].out[1], clock[4].counterModules[2].remote[0].celem[3].bit[0].
    out[1]) on path "base component" is satisfied. [4.237E-9,
    1.3325000000000002E-8]
[line 26, char 10]: Delay delay 13 := (clock[3].thresholdGates[0].celem
    [0].bit[0].out[1], clock[0].counterModules[2].remote[0].celem[3].bit[0].
    out[1]) on path "base component" is satisfied. [5.477E-9,
    1.4557999999999999E-8]
[line 26, char 10]: Delay delay 14 := (clock[3].thresholdGates[0].celem
    [0].bit[0].out[1], clock[1].counterModules[2].remote[0].celem[3].bit[0].
    out[1]) on path "base component" is satisfied. [4.389E-9,
    1.2997999999999998E-8]
[line 26, char 10]: Delay delay 15 := (clock[3].thresholdGates[0].celem
    [0].bit[0].out[1], clock[2].counterModules[2].remote[0].celem[3].bit[0].
    out[1]) on path "base component" is satisfied. [4.131999999999999E-9,
    1.5649999999999998E-8]
[line 34, char 10]: Delay delay 16 := (clock[3].thresholdGates[0].celem
    [0].bit[0].out[1], clock[4].counterModules[3].remote[0].celem[3].bit[0].
    out[1]) on path "base component" is satisfied. [5.147E-9,
    1.5888999999999998E-8]
[line 26, char 10]: Delay delay 17 := (clock[4].thresholdGates[0].celem
    [0].bit[0].out[1], clock[0].counterModules[3].remote[0].celem[3].bit[0].
    out[1]) on path "base component" is satisfied. [5.473E-9, 1.4676E-8]
[line 26, char 10]: Delay delay 18 := (clock[4].thresholdGates[0].celem
    [0].bit[0].out[1], clock[1].counterModules[3].remote[0].celem[3].bit[0].
    out[1]) on path "base component" is satisfied. [5.668999999999999E-9,
    1.62E-8]
[line 26, char 10]: Delay delay 19 := (clock[4].thresholdGates[0].celem
    [0].bit[0].out[1], clock[2].counterModules[3].remote[0].celem[3].bit[0].
    out[1]) on path "base component" is satisfied. [5.742999999999999E-9,
    1.4753E-8]
[line 26, char 10]: Delay delay 20 := (clock[4].thresholdGates[0].celem
    [0].bit[0].out[1], clock[3].counterModules[3].remote[0].celem[3].bit[0].
    out[1]) on path "base component" is satisfied. [5.0489999999999994E-9,
    1.3031E-8]
0 errors 0 warnings 0 assertion violations
```

Bibliography

[1] Altera. Timing considerations with vhdl-based designs. ftp://ftp.altera.com/up/pub/Tutorials/DE2/Digital_Logic/tut_timing_vhdl.pdf.

[2] K. Beck. *Extreme programming explained: embrace change*. Addison-Wesley Professional, 2000.

[3] T. Bray, J. Paoli, C.M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (xml) 1.0. *W3C recommendation*, 6, 2000.

[4] Shigeru Chiba. Javassist. http://www.javassist.org/, 7 2011.

[5] Altera Corporation. Altera. http://www.altera.com/, 7 2011.

[6] P. Eles, K. Kuchcinski, Z. Peng, and A. Doboli. Timing constraint specification and synthesis in behavioral vhdl. In *Proceedings of the conference on European design automation*, pages 452–457. IEEE Computer Society Press, 1995.

[7] G. Fuchs and A. Steininger. Vlsi implementation of a distributed algorithm for fault-tolerant clock generation. *Journal of Electrical and Computer Engineering*, 2011.

[8] Gottfried Fuchs. *Fault-Tolerant Distributed Algorithms for On-Chip Tick Generation: Concepts, Implementations and Evaluations*. PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, 2009.

[9] Matthias Fuegger. *Analysis of On-Chip Fault-Tolerant Distributed Algorithms*. PhD thesis, Vienna University of Technology, 2010.

[10] Flemming Nielson Hanne Riis Nielson. *Semantics with Applications: A Formal Introduction.* Wiley Professional Computing. Wiley, 1992.

[11] IEEE-SA Standards Board. *IEEE Standard VHDL Language Reference Manual*, ieee std 1076, 2000 edition, 2000.

[12] Open Verilog International. *Standard Delay Format Specification Version 3.0.* Open Verilog International, May 1995.

[13] JUnit.org. Junit.org. http://www.junit.org/, 7 2011.

[14] M.R. Kakoee, H. Shojaei, H. Ghasemzadeh, M. Sirjani, and Z. Navabi. A new approach for design and verification of transaction level models. In *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*, pages 3760–3763. IEEE, 2007.

[15] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann, 1997.

[16] Oracle. *Java SE 6 Documentation*, 2010 edition, 2010.

[17] T. Parr. *The definitive ANTLR reference: building domain-specific languages*. Pragmatic Programmers, 2007.

[18] T. Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. Pragmatic Programmers, 2009.

[19] T. Parr. Antlr parser generator. `http://www.antlr.org/`, 7 2011.

[20] A. Peeters, F. te Beest, M. de Wit, and W. Mallon. Click elements: An implementation style for data-driven compilation. In *2010 IEEE Symposium on Asynchronous Circuits and Systems*, pages 3–14. IEEE, 2010.

[21] U. Schmid, A. Steininger, and M. Sust. Fit-it-projekt darts: dezentrale fehlertolerante taktgenerierung. *e & i Elektrotechnik und Informationstechnik*, 124(1):3–8, 2007.

[22] K.S. Stevens, R. Ginosar, and S. Rotem. Relative timing [asynchronous design]. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 11(1):129–140, 2003.

[23] H. Tremblay T. Freese. Easymock. `http://easymock.org/`, 7 2011.

[24] Xilinx. *Constraints Guide*, 10.1 edition, 2008.