Hardware synthesis in ForSyDe

The design and implementation of a
ForSyDe-to-VHDL Haskell-embedded compiler

ALFONSO ACOSTA

Master's Thesis at KTH/ICT/ECS
Supervisor: Ingo Sander
Examiner: Ingo Sander
Stockholm, June 2007

Abstract

The ForSyDe (*Formal System Design*) methodology is targeted at modelling systems, with the goal of using a high level of abstraction in the speci cation of its models.

Although it is a general system modelling methodology, the initial scope of ForSyDe has speci cally been *Synchronous Systems* (systems in which a global clock is used to synchronize the di erent parts of the system). A well-known type of such system is synchronous hardware, which is the main subject of this thesis. A synchronous system in ForSyDe is based on the concept of *processes* which *“map input signals onto output signals”*.

Currently, the software implementation of ForSyDe is based upon the Haskell programming language. The designer speci es the system model in Haskell as a network of cooperating process constructors with the assistance of the ForSyDe Library.

Until now, there has not been an automated way to synthesize ForSyDe models (i.e. generate an equivalent low-level implementation from which to build real hardware) . However, as a result of this thesis, hardware synthesis is now a feature of ForSyDe, enabling ForSyDe designs to nally reach silicon. That is possible thanks to the development of a ForSyDe-to-VHDL compiler. By using this compiler, a ForSyDe model can be rst translated to synthesizable VHDL93 (one of the two most common hardware design languages) and then, the designer can use any of the existing VHDL-tools to synthesize the model.

This thesis report is aimed at documenting the background, design, implementation and use of the compiler.

Acknowledgements

There are many people who in one form or another contributed to the work described in this thesis.

I would like to thank Ingo Sander, my supervisor, for his flexibility and trust during the thesis. He has always been positive about my, sometimes not-so-frequent and too extensive, thesis status reports. I would also like thank him and give him credit for kindly letting me use some of the gures of his PhD thesis [1].

I am also tremendously thankful to the Haskell community, which has proven to be, by far, the most friendly, helpful and smart programming community I have ever been involved with. Special thanks go to the guys at the `#haskell` IRC channel at `freenode.net` and the `haskell-cafe` mailing list. There are too many community-people to thank but I would like to speci cally mention Oleg Kiselyov for his selfless help xing and designing the compiler's typechecker.

Koen Claessen gave me access to Lava's sourcecode, which turned to be essential during the developmnent of the compiler. Without Lava, the compiler would have been totally di erent and perhaps even unfeasible.

Last but not least, I would like to thank Iván Pérez and Miguel Jiménez who helped reviewing parts of the thesis, my parents, for their support, whithout which I would not had been able to spend my last two academic years in Stockholm and nally, all the Erasmus students which turned my stay in Sweden into one of the best experiences of my life (they know who they are).

Stockholm, June 2007

Alfonso Acosta

Preface

This thesis report is aimed at documenting the background, design, implementation and use of a compiler which translates ForSyDe (*Formal System Development*) speci cations into VHDL93 [2] synthesizable code.

The current implementation of ForSyDe, in which the compiler was embedded, is written in the Haskell [3] programming language. That means the compiler has unavoidably been coded in the same language. For that reason, **it will be assumed that the reader of this report is fluent in Haskell**. Introducing the reader to Haskell is out of the scope of the present text. There are many resources from where to acquire the necessary knowledge. Just to mention three of them, *A Gentle Introduction to Haskell* [4] is available online at no cost and is supposed to be a friendly text for the newcomers, Thomson's book *Haskell: the Craft of Functional Programming* [5] on the other hand is longer but covers the language in greater detail and nally the upcoming *Real-World Haskell* book[1] [6] will be freely available online and will help to get involved with serious real-world Haskell code and practical Haskell programming. Mastering Haskell is not a prerequisite to understand the overall content of the thesis. However, wide Haskell programming-experience and familiarity with its common extensions would de nitively be useful (and is probably needed) to comprehend the compiler's design and implementation details.

In addition, it will be assumed that the reader is familiar with digital hardware design and development through HDLs (*Hardware Design Languages*) or, in the worst case, understands the concepts behind it. This prerequisite is weaker than knowing Haskell, since speci c knowledge of VHDL is not essential to read the report. However, being aware of the purpose of an HDL is vital to understand the goals which were achieved.

This thesis report is divided in ve chapters:

Chapter 1 introduces the reader to ForSyDe and the thesis goals. It also introduces the concept of Embedded DSL (*Domain Specific Language*) which is essential to understand ForSyDe's implementation.

[1]Not ready by the time of writing this preface.

Chapter 2 is targeted at comparing ForSyDe with Lava, a successful Haskell-embedded HDL (*Hardware Description Language*) and veri cation environment. The comparison was written before developing the compiler, in order to acquire the necessary background in the *Hardware Design and Functional Languages* research eld, hoping to be able to later reuse previous research results and provide ForSyDe's compiler with state-of-the-art features.

ForSyDe's translator to VHDL turned out to be strongly influenced by Lava. As a result of the comparison, the compiler is intended to inherit Lava's virtues and overcome some of its problems such as its current lack of component reusability on the compiler-level. **Chapter 3** describes the design of the compiler and the motivation behind it whereas **Chapter 4** is targeted at the end-user and contains a tutorial to help getting familiar with the tool and its API.

Finally **Chapter 5** closes the thesis analyzing its results and outlining potential improvements and further work.

As an add-on, **appendix A** has been written with future developers in mind. They will surely nd this appendix useful to get familiar with the compiler's implementation in rst instance, and later improve it or extend it at will (the sources are available under the BSD licence at `http://www.imit.kth.se/info/FOFU/ForSyDe/HDForSyDe/`).

Contents

List of Figures

List of Abbreviations

ADT	Abstract Data Type
API	Application Programming Interface
AST	Abstract Syntax Tree
BSD	Berkeley Software Distribution
CAD	Computer-Aided Design
DSL	Domain Speci c Language
EDSL	Embedded DSL
ForSyDe	Formal System Design
FPGA	Field-Programmable Gate Array
GHC	Glasgow Haskell Compiler
HDL	Hardware Description Language
IO	Input Output
IRC	Internet Relay Chat
MPTC	Multi-parameter Type Class
RTL	Register Transfer Level
ST	State Transformer
TH	Template Haskell
VHDL	VHSIC HDL
VHSIC	Very-High-Speed Integrated Circuit
VLSI	Very-Large-Scale Integration
XML	eXtensible Markup Language
YACC	Yet Another Compiler Compiler

Chapter 1

Introduction

This chapter is aimed at introducing the reader to ForSyDe and the goals of this thesis.

The main intention of this introduction is to bring the reader to understand ForSyDe in a friendly way. Thus, when necessary, clarity was chosen over formalisms. If a complete and more accurate description of ForSyDe is required, please refer to [7] and [1].

1.1 What is ForSyDe?

ForSyDe, which stands for *Formal System Design*, is a system design methodology *"which has been developed with the objective to move system design to a higher level of abstraction and to bridge the abstraction gap by transformational design refinement"* [1].

ForSyDe targets system modelling in general. However, by the time of writing this thesis, the methodology only covers *Synchronous Systems* (systems in which a global clock is used to synchronize the di erent parts of the system). A well-known type of such system is synchronous hardware, which is the main topic of the thesis.

1.1.1 Why a higher abstraction level?

The systems designed nowadays, with microelectronic systems as a particular example, are tremendously complex due to the increasing feature and functionality demands of the market.

Furthermore, not only designs are more complex, the aggressive competitivity of industry requires companies to also shorten the time-to-market of their products, a ecting the development cycle.

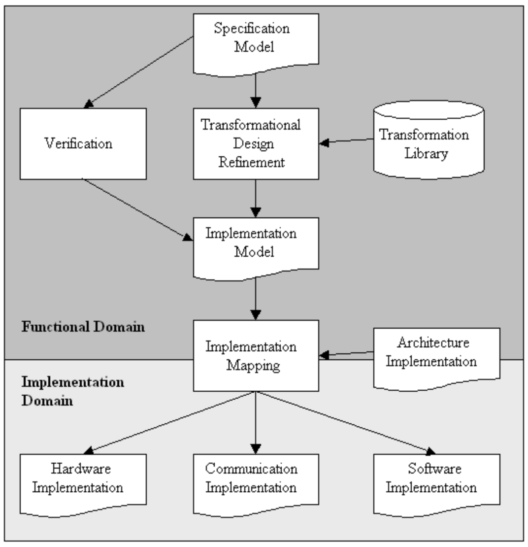


Figure 1.1: ForSyDe's design flow

For the reasons mentioned above, previous techniques, such as the RTL (*Register Transfer Level*) languages developed during the 80's (mainly Verilog and VHDL) give too much detail for the designer to handle. In other words, the level of abstraction of those techniques is too low and hinders the design process. As a result, a big e ort was devoted to raise the abstraction level of design automation tools, resulting in the *System-level Design* research eld, to which ForSyDe belongs.

1.1.2 ForSyDe's design flow

1.1.2.1 Specification model

Figure 1.1 summarizes ForSyDe's design flow. The design is initiated by writing its *specification model.*

As it was previously stated, ForSyDe currently only covers *Synchronous Systems.* For that reason, the speci cation model follows a *synchronous model of computation.* The *specification model* of a synchronous system in ForSyDe is based on the concepts of *synchronous signal* and *process*:

- A **synchronous signal** is a general term in Computer Science and the Telecommunications eld. It can be de ned as an entity transmitting information in a synchronous manner. That is, the transmission of information is arbitrated by a clock and remains static during each clock period, only allowed to change between periods.

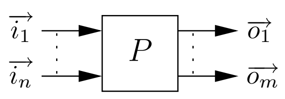


Figure 1.2: Process

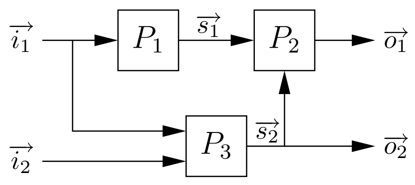


Figure 1.3: A Synchronous system model in ForSyDe

ForSyDe uses the following notation to express a synchronous signal $\overrightarrow{s}$,

$$\overrightarrow{s} = \ll v_0, v_1, v_2, \dots \gg$$

where v_i indicates the value of the signal during period i of the clock.

From now on, this report will use the terms *signal* and *synchronous signal* equivalently.

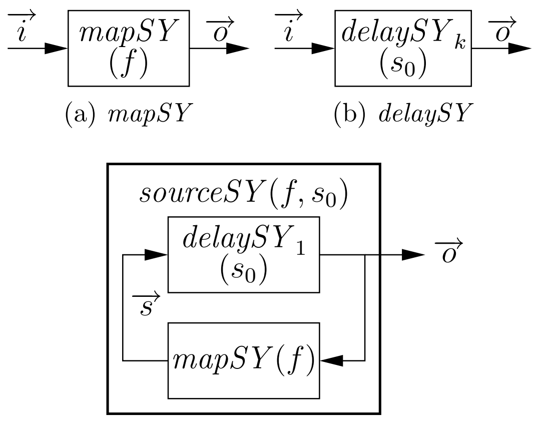
- The meaning of ***process*** is speci c to ForSyDe and is represented in Figure 1.2. A *process* can be de ned as a computational entity which takes $n \in \mathbb{N}_0$ synchronous input signals, might process them and then produce $m \in \mathbb{N}_0$ synchronous signals as output.

A synchronous system in ForSyDe is modelled as a network of interconnected cooperating processes which are in charge of taking the input signals of the system, process them and nally produce its output signals. A simpli ed example can be seen in gure 1.3. As a result, writing the *specification model* consists in de ning the aforementioned process network.

The ForSyDe methodology provides a wide variety of primitive and derived process constructors (entities used to build processes) in which to base a model. Consult [1] for details on the available process constructors.

The following examples help to understand how the *specification model* is written and how processes work:

- *mapSY* (gure 1.4a) is a primitive process constructor aimed at processing an input signal (i) through a function (f, which must be provided to the constructor in advance) and output it for later processing by other parts of the system.



(a) *mapSY* (b) *delaySY*

(c) *sourceSY*

Figure 1.4: Primitive and derived process constructors

- The $delaySY_k$ primitive process constructor (gure 1.4b), on the other hand takes an initial value s_0 and delays a signal $\overrightarrow{i}$ by k clock periods

  $delaySY_k$ is useful to avoid zero-loops (known as combinational loops in the hardware world), which are not allowed in ForSyDe as it will be seen in next chapter.

- *sourceSY* (gure 1.4c) is a derived process constructor aimed at producing a custom source signal. It is derived from *delaySY* and *mapSY*.

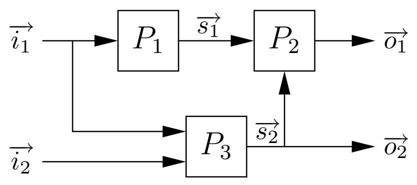
- Figure 1.5 contains a simple speci cation model, in which the reader can see its di erent processes. It is worth to note the use of numerical signals and the $zipWithSY_k$ process constructor, which is a generalization of *mapSY* for k inputs.

1.1.2.2 Implementation model

The next step in the design flow is to transform the speci cation model into the *implementation model.*

The purpose of this intermediate step is to re ne the design and to add low level information details which might be needed for an e cient implementation. As it was stated before, ForSyDe's tries to to *"bridge the abstraction gap by transformational design refinement"*, which is exactly what happens in this stage.

The initial speci cation model is re ned by automatic *design transformation rules* which rely on the formal foundations of ForSyDe.



$$\begin{aligned} P_1 &= delaySY_1(0) \\ P_2 &= zipWithSY_2(*) \\ P_3 &= zipWithSY_2(+) \end{aligned}$$

Figure 1.5: A simple speci cation model

Unfortunately, by the time of writing this thesis, the automatization of this stage has not yet been implemented and constitutes a tremendously complex task by itself due to the wide range of possible transformations.

1.1.2.3 Implementation mapping

The last stage of the design flow consists in transforming the *implementation model* into an architecture-speci c model, such as a software implementation (e.g. C, C++, ...) or hardware speci cations (e.g. VHDL,Verilog,...) from which to synthesize hardware.

In the same way as the *design transformation rules*, the *implementation mapping* stage was not automated before this thesis was written. However, the main goal and outcome of the thesis has been to produce a compiler to translate ForSyDe speci cations into VHDL. From the VHDL model any of the available tools can be used to build or simulate real hardware, making possible to automatically synthesize ForSyDe speci cations.

1.2 ForSyDe's implementation

In order to use ForSyDe in practice, the designer needs a language in which to specify the model. Current implementation of ForSyDe is based on a EDSL (*Embedded Domain Specific (programming) Language*).

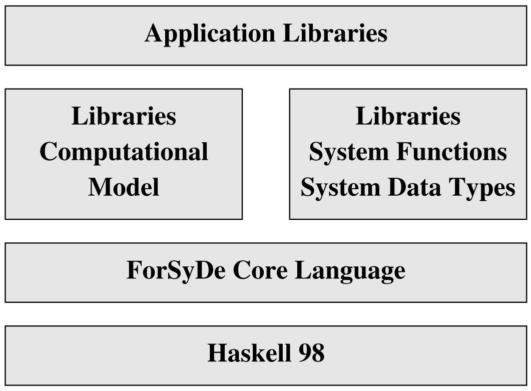


Figure 1.6: ForSyDe's Library

1.2.1 Embedded DSLs

A DSL (*Domain Specific (programming) Language*), in contrast to a general-purpose programming language such as C, is a programming language designed for a speci c kind of task. ForSyDe is speci cally targeted at System Modelling and for that reason, the language chosen to write ForSyDe models must necessarily be a DSL.

On the other hand, an *embedded language* is a programming language which relies on an existing language, called *host language*, as opposed to the embedded language itself which is called *guest language*. In practice, the guest language is embedded by adding a library to the host language.

The main advantage of the language embedding approach is the reutilization of the syntax of the host language, its surrounding tools and documentation. Strongly-typed languages like Haskell, help embedding and are a good choice as host languages due to their encapsulation properties. However, an embedded language has, as well, many disadvantages due to the syntactical and semantical dependence on the host language.

Examples of popular EDSLs are YACC (the C-based parser generator) and EmacsLisp (used as the scripting language of the popular Emacs editor).

1.2.2 ForSyDe's Library

ForSyDe is implemented as a Haskell-embedded DSL. All ForSyDe's process constructors, together with other functionalities, are included in a Haskell library. Figure 1.6 illustrates the structure of ForSyDe's Library which is used by the designer to write the speci cation model in Haskell.

Choosing Haskell as the host language was not an arbitrary decision. Functional languages have proven to t nicely with hardware design [8]. Even if ForSyDe is aimed at system design in general, it matches perfectly with the characteristics of Haskell:

- As it was previously described, a signal in ForSyDe is viewed as stream of values. That makes it possible to model a signal as a list, which is the main data structure of functional languages in general and Haskell in particular.
- Processes take signals as input, process them and output new signals which are later forwarded to other processes, forming a network. That ts perfectly well with functional languages. A process can be modelled as a function which makes computations over lists.
- The majority of ForSyDe's process constructors, such as $mapSY$, take functions as input. Yet again, that can be easily modelled making use of higher order functions, which are available in Haskell. Note that this reflects the initial intention of embedding ForSyDe in Haskell since its process constructors are named after widely-used higher-order Haskell functions (e.g. `map`, `zipWith` ...).

During the rest of this thesis the term ForSyDe refers indistinctively to both the methodology and its library.

1.3 Thesis scope and goals

The task of this master's thesis is to develop a tool that takes a ForSyDe implementation model as input and produces a hardware description in VHDL. The master's thesis must attain the following goals:

1) Study of ForSyDe and related work relevant to this thesis.

2) De nition of a relevant subset of Haskell that is accepted by the synthesis.

3) Development of the synthesis tool according to [1].

4) Evaluation of the tool, identifying and including possible improvements.

5) Detailed documentation of the tool.

It is important to note that it has not been possible to use the implementation model as input due to the lack of an automatic tool to apply the transformation rules. Instead, the compiler takes the speci cation model directly. Automating the *Transformational Design Refinement* constitutes a much larger project by itself and would have required a high percentage of this thesis to be completed, hindering the development of the compiler.

Chapter 2

Lava vs ForSyDe

This chapter is aimed at comparing Lava and ForSyDe, two DSLs (*Domain Specific Languages*) embedded in Haskell. The comparison was made at the initial stage of the thesis, as a means of acquiring a picture of the previous work in the Functional Programming and Hardware Description eld, in order to hopefully reuse earlier design techniques and to provide ForSyDe's compiler with state-of-the-art features. Thus, at the time of writing this comparison the compiler was not yet designed.

During the comparison, the reader will get familiar with the important concept of *Embedded Compiler*. Furthermore, by the end of the chapter he (or she) will ...

- ... be able to understand the problems related to represent circuits in a purely functional programming language such as Haskell.

- ... hopefully have acquired a valuable background in the Hardware design and Functional Languages eld along with an up-to-date view of the previous work related to this thesis.

2.1 Introduction

The complexity of electronic designs has tremendously risen during the past two decades which, together with the Industry's exigency of extremely short time-to-market periods have made RTL-level tools no longer t nowadays circuit design requirements. Thus, there has been a natural move to a higher level of abstraction known as *System Level Design*.

The two most commonly-used RTL-level HDLs (Verilog and VHDL) are far too verbose and concrete to cleanly give a general view of a complex system.

Therefore, new languages are needed to provide a solution for the aforementioned problems. Among other approaches which require creating a new programming language from scratch, the *Embedded Approach* saves the designer having to learn a new syntax and makes possible to reuse all the existing tools (i.e. compilers, interpreters ...) surrounding the *host language*.

Furthermore, the expressiveness of declarative languages such as Haskell seems to suit the abstraction level of *System Level Design* [9] and hardware design semantics [8].

During the rest of this chapter, two Haskell-embedded language approaches will be compared: Lava [10] and ForSyDe [7]. The later targets synchronous systems in general (that is, systems in which a single global clock is used[1]), while the rst speci cally describes synchronous hardware (a concrete type of synchronous system after all) and is the most mature of the two.

2.2 Design flow

Lava and ForSyDe were independently developed by the Swedish universities of Chalmers and KTH.

Figures 2.1 and 2.2 contain diagrams describing the simpli ed design flow to be followed when modelling with Lava and ForSyDe. As it will be later seen, the flow is quite similar to the traditional compilation model of a general-purpose programming language.

It is easy to suspect that both Lava and ForSyDe follow a similar approach, sequentially divided in two stages:

1) The designer models a circuit in Haskell, making use of the Lava or ForSyDe libraries.

2) The obtained model is automatically processed by a compiler in order to attain di erent *goals*: simulation, testing, veri cation and translation to a less abstract HDL (e.g. VHDL) directly synthesizable to hardware. The di erent flavours of this stage are known as *Interpretations* in Lava and *Implementation Mappings* in ForSyde. In the rest of this chapter I will refer to them with the wider spread term: *backends*. The details of these backends will be described in next section.

[1]ForSyDe supports as well a *multi-rate model* by using *multi-rate interfaces*, able to connect di erent synchronous sub-domains working at di erent clock rates.

Figure 2.1: Lava's design flow

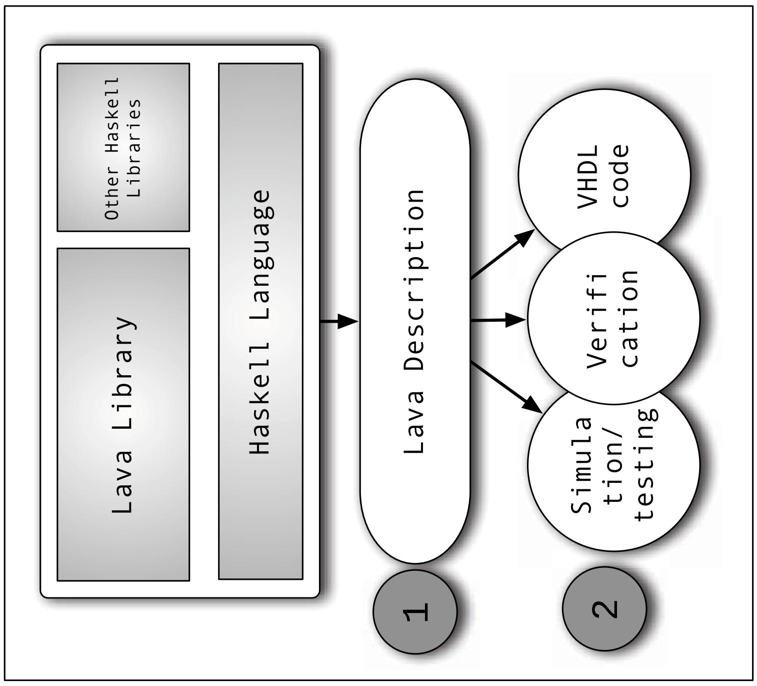
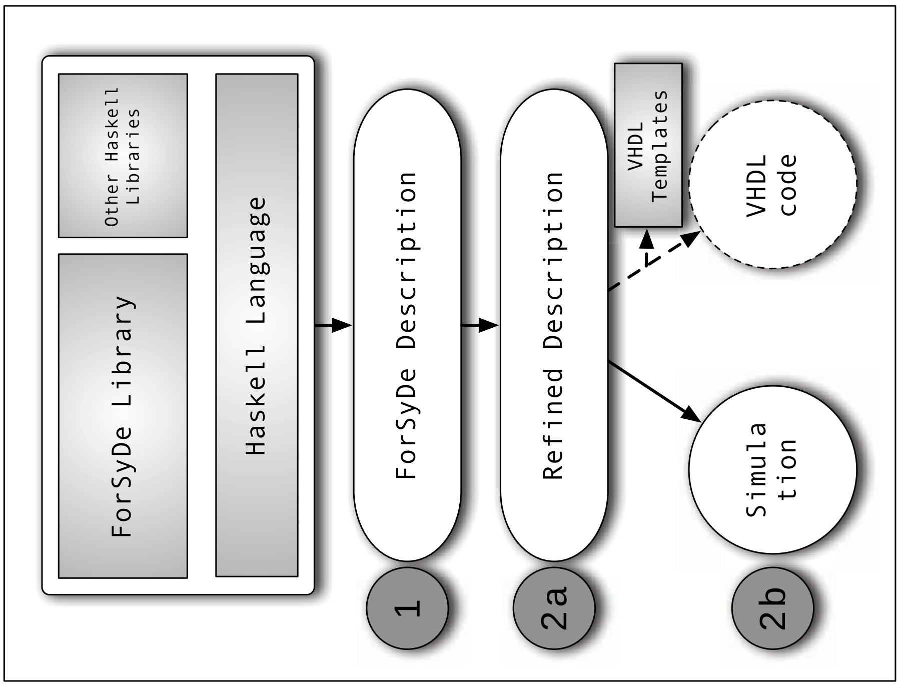


Figure 2.2: ForSyDe's design flow



This stage slightly di ers depending on the language used, being more complex, and potentially more likely to include optimizations in the case of ForSyDe, due to the application of *Design Transformations.*

ForSyDe has a strong formal base which allows the designer to work at a highly abstract functional level. The abstraction level is lowered[2] by applying *Transformation Rules* to the provided hardware descriptions. That is captured in gure 2.2 as the transition from 2a to 2b.

The internal details of ForSyDe's *Refinement* are out of the scope of this chapter but is worth to mention that the *Transformation Rules* can be divided in *semantic preserving* and *design decisions.* The later ones change the semantics of the model and thus, its application must be supervised by the designer.

2.3 Backends

As it was mentioned previously, Lava is in a more mature state than ForSyDe, having a whole set of tools surrounding it.

On the other hand, ForSyDe only currently supports simulation by direct execution of its Haskell models[3]. Even with that, a *template system*, in which every library function has a preassigned VHDL template, has been already planed and a few prove-of-concept examples [1, Chapter 6] back it as a promising approach.

As of the time of writing this thesis, Lava has three available backends:

- **Simulation and random testing**. A circuit simulation is carried out by interpreting each component of its structure[4].

  In addition, Lava allows to test properties of a design on random data through a language which is highly inspired in QuickCheck [12].

  QuickCheck is as well embedded in Haskell and has been developed independently of the Lava system. That makes it suitable of being used for other purposes than testing circuit properties. Indeed, QuickCheck has been successfully employed in many other projects and could certainly be integrated into ForSyDe if desired.

[2]ForSyDe currently lacks an automatic tool for this purpose and transformations need to be applied manually. Either the case, [1] proves it would not only feasible but quite straight-forward to implement.

[3]*Zero-delays*, circuit feedback loops without delays, are forbidden in ForSyDe and thus cannot be simulated.

[4]Zero-delays are only permitted when performing a *constructive* simulation by using `simulateCon`, see Lava s documentation [11] for details.

- **Verification**. Lava is able to generate a logical formula representing the circuit. That formula, along with properties de ned by the designer, are given to an external theorem prover which can prove or disprove their validity.

  Despite how promising veri cation can seem at rst sight, there is a well-known underlying theoretic limitation within First Order Logic which makes it semi-decidable[5]. Furthermore, First Order Logic veri cation is computationally expensive (a NP-complete problem) and usually the prover has to be helped by splitting proves in smaller ones.

  Fortunately random testing is still at hand and, although it does not answer the validity question, it can be good enough in most of the cases.

  Nevertheless, theorem provers can give an answer about the validity of many circuit properties, being especially valuable in critical design parts[6]. As opposed to simulation and random testing, validation provides certainty over circuit properties. Thus, it is more desirable but frequently more di cult to apply.

- **RTL-level language generation**. In order to synthesize a design to hardware, Lava includes a VHDL backend. As it will shown in the next section, such kind of translation is quite straight-forward to achieve in Lava due to the way in which circuits are internally represented. ForSyDe opts for more-behavioural semantics making the translation more di cult to implement.

2.4 Language features

Both Lava and ForSyDe can be used to describe synchronous circuits[7]. The functional paradigm invites to model circuits as functions which receive signals as arguments, process them and nally return them or forward them to other functions. Furthermore, higher-order types allow having functions (circuits)

[5]Any valid theorem can be proven but invalid clauses are not always identi ed.

[6]Intel designers surely regret not making extensive use of formal veri cation methods in an earlier stage. Famous bugs as the Pentium® s `FOOf` [13] and `FDIV` [14], could probably have been avoided by applying formal methods. Nowadays Intel makes use of the *Forte Verification Environment* [15], currently based on *reFL^{ect}* [16], and formerly on FL [17], two functional programming languages.

[7]ForSyDe also allows having synchronous subsections working at di erent clock rates, the so-called *synchronous sub-domains*. However, those domains are generated during the *Refinement* stage, which means they cannot be included in the initial hardware description or *Specification Model*.

as rst-class citizens, permitting to combine and nest them in an elegant and intuitive way.

Thus, a hardware model in both Lava and ForSyDe can be viewed as an interconnected set of functions which interact and process signals.

A signal in ForSyDe is de ned with the following recursive algebraic type.

```
data Signal a = NullS | a :- Signal a
```

It is worth to note that

- Signals are represented using a *data stream* metaphor. Its de nition is isomorphic to a Haskell list without syntactic sugar (i.e. the surrounding box brackets *[ ]* and interspersed commas).

  A similar de nition can be found in *Hawk* [18], a Haskell-embedded DSL aimed at microprocessor design. Unfortunately the development of the language seems to be dead at the moment of writing the present thesis.

- Signals are polymorphic. The fact that signal can contain values of any type makes them flexible and provides them with abstraction capabilities.

- Following the signal/data-stream metaphor, it is natural to process signals in the same way as lists. In fact, ForSyDe provides higher order functions similar to the Haskell broadly-used list traversers `map`, `zipWith`, ...

- The lack of encapsulation of the signal type (i.e. its de nition is not hidden to the programmer) makes it really flexible but as it will be later discussed, it does not permit embedded compilation to other representations such as VHDL.

On the other hand, a signal in Lava is more complex than a stream of values, and is hidden to the programmer through an abstract data type.

```
newtype Signal a = Signal Symbol
```

The de nition of `Symbol` is not public, and the phantom type parameter `a` is used as a means to provide a type safety layer for signals.

A signal in Lava hiddenly represents the internal structure of the circuit:

“Instead of implementing signals as streams of booleans, we implement it as a datatype which explicitly keeps track of which gates were used to construct it” [11, section 1.6].

That has three immediate implications

1) A circuit description in Lava is unavoidably and deliberately structural[8]. On the other hand, ForSyDe is inherently structural as well[9], but the representation of signals as streams allows the designer to describe systems in a more behavioural manner if required.

   For that reason, it can be said that ForSyDe stands on a higher abstraction level. As a drawback, processing the netlist of a circuit is potentially much harder to achieve.

2) A component-wise signal eases the task of processing, translating and transforming a circuit (i.e. simulation, veri cation, translation to a di erent target language . . .), permitting to include a translator within the language library.

   This technique (known as *Embedded Compiling* [20]) has been successfully used in many other speci c domains such as databases [21], music composition [22] and image processing [23].

3) A circuit is naturally represented as a graph, whereas the closest data structure directly o ered by Haskell is a tree through the use of *algebraic types*.

   In order to represent the structure of a circuit and to avoid in nite recursion problems related to circuit loops, Lava o ers two solutions: Monads (currently discarded) and Observable Sharing which will be described later on.

Among other di erences, it should be remarked how both languages make use of Haskell characteristics. Lava makes an elegant use of type classes whereas ForSyDe's programming style is closer to the one used by a general-purpose Haskell programmer:

- Curry ed functions are used in ForSyDe while Lava uses an uncurry ed style.

- ForSyDe makes use of higher order functions and polymorphic signals which aids reusability. Furthermore, a ForSyDe description consists of a network of cooperating processes joined together through *process constructors* which isolate computation and communication. Each constructor has the capability of using a di erent computational model if desired [24].

[8]Part of the Lava team has proposed another imperative behavioural language called *Flash* [19].

[9]A model in ForSyDe is presented as the result of connecting di erent processes.

On the other hand, instead of o ering traditional higher order functions (in the sense of data processing callbacks), Lava o ers circuit combinators such as serial and parallel composition. Unfortunately, even if the Lava's signal type is polymorphic, only monomorphic `Int` and `Bool` signals can be used in practice[10].

- The functions o ered by the ForSyDe library follow Haskell's philosophy and naming scheme (e.g. `mapSY`, `zipWithSY`, `scanlSY` ...).

2.4.0.1 Layout-oriented Lava: *Xilinx-Lava*

Throughout the rest of this thesis the term *Lava* will refer to the main branch of the HDL designed in Chalmers university, also known as *Chalmers-Lava*. However, Satnam Singh, one of the Lava researchers developed a layout-oriented branch of the language, known as *Xilinx-Lava*, which is aimed at describing circuits for implementation of Xilinx's Virtex family of FPGAs [25].

As opposed to *Chalmers-Lava*, *Xilinx-Lava* provides a combinator library to build circuits in a way which allows controlling the nal layout of the FPGA (i.e. how the FPGA blocks are allocated and interconnected) without losing Lava's elegance. So much so that it beats traditional HDLs when it comes to optimizing floorplanning [26].

Due to its layout capabilities, *Xilinx-Lava* is unavoidably less abstract than *Chalmers-Lava* and, unfortunately, as its name clearly indicates, is speci c to Xilinx's technology.

2.4.1 Representing a circuit in a pure functional language

Hardware design with functional languages has been a matter of research for many years. Its history is neatly summarized by paper [27], which is de nitely recommended to read in order to acquire a deeper background in functional HDLs. More speci cally, the problem of representing a circuit in a pure functional programming language has been addressed and extensively discussed for more than 20 years, mainly by O'Donnell [28, 29, 30, 31, 32, 33].

"The problem is that circuits are finite graphs - but viewing them as algebraic (lazy) data types makes them indistinguishable from potentially infinite regular trees." [31]. In other words, there is no way to directly detect feedback loops within a circuit if an algebraic type is chosen to represent it.

[10]The encapsulation of the signal type in addition to the aforementioned type-safety layer only allows `Int` and `Bool` signals to be created and propagated. That ensures type correctness and saves the trouble of having to add typechecker to the embedded compiler.

In order to solve that problem, there are four main alternatives: *explicit labeling*, *monads*, *observable sharing* and *host language transformations.*

2.4.1.1 Explicit labeling

This method was proposed by O'Donnell in [29]. In order to prepare the circuit for later traversing, the designer explicitly chooses a label for each node (component) of the circuit.

The approach has many problems, described by O'Donnell himself later on:

"The use of labeling solves the problem of traversing circuit graphs, at the cost of introducing two new problems.
It forces a notational burden onto the circuit designer which has nothing to do with the hardware, but is merely an artifact of the embedding technique. Even worse, the labeling must be done correctly and it cannot be checked by the traversal algorithms.
Suppose that a specification contains two different components that were mistakenly given the same label. Simulation will not bring out this error, but the netlist will actually describe a different circuit than the one that was simulated. Later on the circuit will be fabricated using the erroneous netlist. No amount of simulation or formal methods will help if the circuit that is built doesn't match the one that was designed." [33]

2.4.1.2 Monads

This approach was initially adopted and later discarded by the creators of Lava. The labels of the circuit are uniquely and automatically generated through a state monad and stored in the signal abstract data type.

It solves the main problems caused by *explicit labeling* but its main drawback is that, by introducing monads, the syntax in which circuits are expressed changes completely[11] making circuit descriptions less intuitive for the designer.

Furthermore, feedback cannot longer be expressed by means of equational recursion (because of the loss of local naming), and *loop*, a special monadic combinator is required. Again, O'Donnell analyzed the disadvantages of this approach and justi ed why it was not included in his functional HDL: Hydra [35]

"there are two disadvantages of using monads for labeling [..] The first problem is that monads introduce new names one at a time, in a sequence of nested scopes, while Hydra requires the labels to come into scope recursively,

[11]Monads have long been one of the biggest learning barriers for Haskell [34], being deeply confusing for the newcomers.

all at once, so that they are all visible throughout the scope of a circuit definition.[..] A more severe problem is that the circuit specification is no longer a system of simultaneous equations, which can be manipulated formally just by 'substituting equals for equals'. Instead, the specification is now a sequence of computations that —when executed— will yield the desired circuit. It feels like writing an imperative program to draw a circuit, instead of defining the circuit directly." [33]

An alternative to Monads: Arrows Arrows [36] are a computation abstraction similar to Monads. Furthermore, Arrows are more general than Monads and are semantically closer to a circuit since they are often introduced from the perspective of stream processors.

Contrary to Monads, they o er combinator primitives which can be targeted at parallel stream processing. Hughes and Paterson even suggested to use Arrows to simulate synchronous circuits [37, 38], nonetheless, no Haskell-embedded HDL has so far made use of them.

Arrows are not covered by the Haskell standard. However, GHC (*Glasgow Haskell Compiler*) o ers a notation extension [37] which provides extra syntactic sugar to treat Arrows in a similar way as Monads. The same result can be achieved by preprocessing the code with the compiler-independent *Arrows bundle.*

Even with its semantical advantages, Arrows are prone to su er the same syntactic problems as Monads since they make use of a very similar notation and a *loop* xpoint combinator is still required to express feedback within the circuit.

2.4.1.3 Observable Sharing

This is the currently preferred approach in Lava [31].

The method consists in using references[12] (pointers) to represent the nodes within the graph structure of the circuit (like it would naturally be done in an imperative language). Then, during the graph traversal, loops are detected by comparing the reference of current node against the one of every visited node, whose reference must have been properly saved in advance.

In order to perform such equality comparison, the language needs to be extended with a side-e ecting operation known as `unsafePerformIO`[13].

[12]References are only implicitly used within the signal ADT and are transparently handled for the programmer. Making the reference comparison explicit would constitute a di erent solution known as *Pointer Equality.*

[13]All current up-to-date Haskell implementations o er this feature.

Observable Sharing allows to design circuits using recursive equations, without the drawbacks of explicit labeling nor the inconvenient monadic syntax. As a tradeo , Haskell needs to be extended into a language which violates referential-transparency, making equational reasoning unsound.

Furthermore, the programmer needs to be aware of such extension since it a ects the way in which connections are shared by the di erent components of the circuit.

2.4.1.4 Host language transformations

The *host* language (Haskell in this case) is preprocessed in order to add the node labels.

As result of the translation an equivalent circuit description is obtained, with correct automatically-added labels, not prone to designer errors, without side-e ects[14] nor unsuitable monad notation.

As it can be suspected, this approach entails the extra e ort of parsing the language and translating it, loosing the pleasant reusability of machinery expected from an embedded language. Furthermore, supporting the full language syntax can be an enormous task and could make the resulting tool di cult to maintain.

The advantages of the embedded language approach, in and of themselves normally questioned [39], are reduced. Syntax reusability would be the only remaining advantage, not being clear if a stand-alone language (as opposed to embedding) would be preferable.

However, a subtle transformation of Hydra was carried out by O'Donnell [33] by making use of TH (*Template Haskell*).

TH [40] is a Haskell extension that provides type-safe (and type-aware) compile-time meta-programming. In his paper, O'Donnell makes use of TH as a macro system to automate the node labeling of the circuit.

Contrary to other popular macro systems, not only is TH type-safe but also gives parsing and AST data structures for free. That allowed O'Donnell to forget about parsing and code his translator as a simple compiler backend, avoiding an otherwise tremendous e ort.

Nonetheless, O'Donnell's approach su ers from various problems:

- **Obfuscation**. As result of preprocessing, the original design is obfuscated and di cult to understand at rst sight.

[14]The translated code is pure but it could be said that the original description includes side-e ects anyway, implicitly carried out by the translation.

- **GHC-specific**. Template Haskell is not part of the Haskell standard [3] and is only currently supported by GHC. Nevertheless, GHC is the current *de facto* reference Haskell compiler.
- **Host language limitations** The publicly available Hydra/TH implementation is very limited and does not support the full feature set of Haskell (not even lambda abstractions are supported). Of course, TH supports the whole Haskell standard and Hydra could be extended to supported. However this shows that supporting it would have made the host language transformations more di cult to implement.
- **Maintainability**. The dependency on an experimental tool as TH and the wide scope of its use in this approach (traversing the full Haskell AST) makes Hydra/TH di cult to maintain. As a matter of fact, the latest Hydra/TH public version available is outdated at the moment of writing this thesis due to changes in the API of TH.

2.5 Lava and ForSyDe in practice

The general characteristics of ForSyDe and Lava have been so far discussed and compared. Even with that, it is di cult to get an overall impression of both languages without having a look at a practical example.

As it was previously stated, ForSyDe is aimed at designing synchronous systems in general, being synchronous hardware just an example of such systems. However, for comparison purposes, a hardware design example (a simple bit adder) was picked from the Lava tutorial [41].

Here is a half adder design in Lava.

```
halfAdd :: (Signal Bool,Signal Bool) -> (Signal Bool,Signal Bool)
halfAdd (a,b) = (sum, carry)
  where sum   = xor2 (a,b)
        carry = and2 (a,b)
```

And here is the full adder, making use of the de nition of `halfAdd`.

```
fullAdd :: (Signal Bool,(Signal Bool,Signal Bool))
           -> (Signal Bool,Signal Bool)
fullAdd (carryIn, (a,b)) = (sum, carryOut)
  where
   (sum1, carry1) = halfAdd (a, b)
   (sum, carry2) = halfAdd (carryIn, sum1)
   carryOut = xor2 (carry1, carry2)
```

Lava can *interpret* (i.e. analyse) the model in three di erent ways.

- Simulating it

```
> simulate fullAdd (high,(low,high))
(low,high)
```

- Verifying certain properties. For instance, this property is aimed at checking that the adder is commutative. It is out of scope to explain the details of how this is internally done.

  ```
  prop_c (c, (a,b)) = ok
   where  out1 = fullAdd (c, (a, b))
          out2 = fullAdd (c, (b, a))
          ok   = out1 <==> out2
  ```

  The property can be easily validated from a Haskell interpreter.

  ```
  > verify prop_c
  Proving: ... Valid.
  ```

- Generating equivalent VHDL code.

  ```
  > writeVhdl "fullAdd" fullAdd
  Writing to file "fullAdd.vhd" ... Done.
  ```

This small example already leads to a few important conclusions

- **Circuit ports**

  As it was previously mentioned, Lava circuits are uncurry ed. It can seem unnatural to a Haskell programmer, but this design decision was not arbitrarily made.

  An uncurry ed function takes only an argument and thus, it can be encapsulated through typeclass contexts, allowing to treat inputs (outputs) in a uniform way.

  The inputs (outputs) admitted by a Lava-interpretable circuit can be de ned by induction as the set I where:

  - $\forall s \in Signals . s \in I$
  - $\forall i \in I . [i] \in I$
  - $\forall i_1, i_2 \in I . (i_1, i_2) \in I$
  - $\forall i_1, i_2, i_3 \in I . (i_1, i_2, i_3) \in I$

    $\vdots$
  - $\forall i_{1,2,\ldots,7} \in I . (i_1, i_2, \ldots, i_7) \in I$

Where *Signals* represents the set of all the valid Signal types of Lava and the brackets keep the same meaning as in Haskell.

The above de nition leads to multiple representations of the same circuit input (output). For instance, let's picture the argument of a circuit taking three signals, a `Bool` signal, followed by an `Int` signal and lastly a `Bool` signal.

There are as well, three possible types for the argument of the circuit's function:

1) `(Signal Bool, Signal Int, Signal Bool)`
2) `((Signal Bool, Signal Int), Signal Bool)`
3) `(Signal Bool, (Signal Int, Signal Bool))`

This feature can be considered redundant and confusing rather than flexible, since it requires a convention on how to structure the circuit inputs. Furthermore it is impossible to avoid the use of nested tuples if the number of input signals is higher than seven. The largest tuple size could be incremented, of course, but it would always remain being nite, and considering the number of inputs required by large VLSI chips nowadays, it does not seem the best solution.

The inputs (outputs) of a circuit are more intuitively expressed with a *port*, in the same way as it is done in traditional HDLs. A good representation for a *port* could be a xed size heterogeneous collection (i.e. a collection whose elements can have di erent types).

Haskell, due to its type strictness and unlike some dynamically-typed functional languages such as Lisp, does not directly support heterogeneous collections. Nonetheless, heterogeneous lists are possible in Haskell as shown by HList [42], a library which relies on common extensions of the language.

Ports could be implemented either through heterogeneous collections or a self made ADT, aware of the internal representation of signals.

The price to pay for using *ports* would be some extra verbosity in the circuit descriptions (negligible if the design is big enough) and the e ort of including a typechecker in case the ADT option is chosen (ports, due to their heterogeneous nature, would no longer be able to take advantage of the type safety layer provided by the phantom signal types).

However, *circuit ports* would make the treatment of inputs (outputs) uniform, scalable and intuitive.

- **Component hierarchy**

  Reusability and hierarchical structures are two major needs of *System Design*. Lava provides solutions to those needs through mechanisms already available in Haskell. Reusability is achieved by factoring code in functions and hierarchy is o ered by delegating parts of the design to subcircuits (which are functions after all).

  In this way, `fullAdd` makes use of `halfAdd` without needing to replicate its code and delegating a smaller task to it.

  However, the internal representation of the circuit is not aware of such delegation and therefore, the di erent backends are not able to reflect the use of hierarchy in the target language.

  That is not a major problem in the case of HDL backends (the components are anyway replicated when the model is synthesized[15]) but it certainly is undesirable in other cases. Suppose that, for instance, a C backend was available. Then code of `halfAdd` would be replicated in the target source le, making it more di cult to understand than if a subroutine was generated instead. Furthermore, compilation would later lead to a larger binary.

  In order to make the backends aware of the circuit hierarchy, the designer would be required to make reusability explicit to Lava. That entails providing component instantiation primitives in the same way it is done in traditional HDLs. As a tradeo , the verbosity of designs would be increased.

Getting back to ForSyDe, an adder could be directly designed as

```
addSY :: (Ord a, Num a) => Signal a -> Signal a -> Signal (a,Bool)
addSY = zipWithSY add
 where add a b = let sum   = a + b
                     carry = sum < a || sum < b
                 in (sum, carry)
```

This behavioural model admits any pair of numbers (in the Haskell sense) as input and calculates their sum and carry (assuming both numbers are unsigned).

The function `zipWithSY` behaves exactly in the same way as Haskell-Prelude's `zipWith`. Remember that ForSyDe models signals as data-streams.

Due to its numerical-representation independence, the model accepts numbers of any length and is highly abstract. However, for the same reason, `addSY`

[15]However, the lack of reusability could a ect the behaviour of the synthesizer. For instance, there is no way to split the design in various parts, which can give a hard time to the synthesizer if the circuit is big enough.

is not really useful in practice (at least regarding hardware design): there is not a clear way in which the circuit could be easily synthesizable.

Furthermore, this example would not be fair to Lava, whose `fullAdd` is proven to be translatable to hardware. A closer approach would be the following

```
halfAddSY :: Signal Bool -> Signal Bool -> Signal (Bool,Bool)
halfAddSY = zipWithSY halfAdd
  where halfAdd a b = let sum   = a /= b
                          carry = a && b
                      in (sum,carry)

fullAddSY :: Signal Bool -> Signal Bool -> Signal Bool
             -> Signal (Bool, Bool)
fullAddSY  carryIn a b = zipSY sum carryOut
    where (sum1, carry1) = unzipSY $ halfAddSY a b
          (sum, carry2)  = unzipSY $ halfAddSY carryIn sum1
          carryOut       = zipWithSY (/=) carry1 carry2
```

Note that, even if ForSyDe makes use of curry ed functions, their outputs su er from the same structural redundancy problem as in Lava (i.e. nested tuples).

Furthermore, ForSyDe allows signals to be compound and contain tuples, which can be lifted and unlifted through *unzipSY* and *zipSY*, adding even more redundancy and demanding another convention to be followed by the designer.

Due to the previous reasons, ForSyDe could also bene t from the circuit *port* approach. However, the *component hierarchy* remark does not apply in this case, since ForSyDe descriptions are still not synthesizable and thus, not yet representable in terms of components.

2.6 Conclusions

The features o ered by functional languages are suitable to embed a *System Level* HDL. Both Lava and ForSyDe follow a similar approach. However, Lava has a whole set of tools surrounding it while ForSyDe only currently o ers simulation. Even though, [1] establishes concrete guidelines on how to automate the re nement stage and the VHDL backend for ForSyDe.

Both Lava and ForSyDe can be considered structural languages. Nonetheless Lava is intrinsically structural due to its internal representation of signals while ForSyDe admits behavioural descriptions of a higher abstraction level if desired. As a drawback, the translation from ForSyDe to other languages (e.g. input for a theorem prover or hardware synthesizer) is potentially more di cult to achieve.

Chapter 3

Design of the compiler

The rst part of this chapter describes the compiler's design, including the decisions and arguments which have leaded to it. In advance, it can be said that it follows Lava's embedded compilation model.

The second part of this chapter explains some improvements which were incorporated to ForSyDe in order to overcome the drawbacks of Lava analyzed in chapter 2.

3.1 Design alternatives

After studying ForSyDe's background and related work, it was necessary to choose between three design alternatives before starting to implement the compiler:

1) **Traditional stand-alone compiler**. This alternative implies coding a full Haskell-to-VHDL compiler from scratch.

2) **Customizing an existing compiler**. The Haskell-to-VHDL compiler would be added to an existing tool by incorporating a custom VHDL back-end.

3) Using an **embedded compilation model**. As it was described in chapter 2, the embedded compilation model involves including the compiler in the language library. This model is used in Lava based on the fact that Lava-signals are component-wise and store a structural description of the circuit.

The third alternative was chosen over the others for various reasons explained in next section.

3.1.1 Why an embedded compiler?

There are various reasons for which the embedded compilation model was chosen:

- It is **realistic**.

  The time and man-power resources of a master's thesis are very limited.

  GHC, the popular Haskell compiler, in its current version (6.6) is composed of around 150.000 lines of Haskell code (or 72 years of estimated one-man full-time dedication)[1]. Even trying to implement a less ambitious compiler, would anyway had been impossible.

  A custom backend would obviously had required a much smaller e ort, but it would have been impossible as well. GHC's code generation modules are composed of more than 5.000 lines of code (14 estimated months of one-man full-time dedication).

  On the other hand, the embedded approach was expected to be feasible for one person. In fact, the whole compiler is composed of less than 2.000 lines of Haskell code.

- **Saves unnecessary effort**.

  The goal of the compiler is to translate ForSyDe speci cations to VHDL. Customizing a backend or coding a full compiler would have allowed to translate any Haskell le (including ForSyDe descriptions in particular).

  Such a general translation is not the goal of this thesis. Mapping any Haskell program to VHDL would have been unnecessary and extremely di cult (if not impossible).

- **Previous success**.

  The embedded compilation model was adopted from Lava, where it was previously used to build a successful hardware development and veri cation environment.

  A previous success always gives a good initial point from were to start working.

- It is easily **maintainable**.

  Using the embedded compilation model, the compiler is included in ForSyDe's Library making it easy to maintain and distribute.

[1]measured with the `sloccount` command, (`http://www.dwheeler.com/sloccount/`)

- It is **independent of the internal design of third-party tools**.

  Modifying an existing compiler to implement a VHDL backend would have caused ForSyDe to depend on a third party tool, making it prone to get outdated by internal design changes in the tool.

3.2 Differences with Lava's implementation

Some di erences between Lava and ForSyDe made impossible to simply replicate Lava's compilation model.

Those di erences had to be analyzed and taken into account in order to be able to preserve Lava's approach. The di erences were identi ed in last chapter and fortunately, an appropriate solution was found for each case:

- **ForSyDe already defines a `Signal` type**.

  The embedded compilation model requires to use a signal type which stores the structure of the circuit.

  However, ForSyDe's library implements the `Signal` type as a stream of values. Furthermore, the `Signal` type is not encapsulated (that is, the data type is not hidden to the programmer), making impossible to modify it without causing regressions.

  There were two alternatives

  a) Transform the original `Signal` type of ForSyDe, into an ADT which kept track of the circuit structure, causing a regression.
  b) Create an alternative signal type which ful lled the requirements of embedded compiling and which could live together with the old `Signal` type.

  Option b was chosen, mainly because ForSyDe is targeted at design of systems in general not simply hardware. The details about the new signal type are explained in section 3.3.

- **ForSyDe is more behavioural than Lava**.

  The behaviour of Lava's gates (e.g `and`, `or`, `xor` ...) is hardcoded and cannot be modi ed, which is perfectly natural for a purely structural language.

  On the other hand, most of ForSyDe's process constructors are implemented as higher-order functions, which behave in one way or another depending on the function passed as argument.

Keeping the structure of the circuit in a the signal ADT was no longer enough to perform the translation to VHDL. In addition, a way to store the body of the functions passed to the process constructors had to be found.

The biggest problem was that an embedded compiler does not have direct access to AST of the host language. The only information it can work with has to be provided by the library in which the compiler is embedded.

One possible solution would have been developing yet another embedded language in which to express the body of those functions. However, it would have implied designing a new sublanguage, with its new syntax, which would had to be learned by the designer.

Instead, inspired by Hydra's implementation [33], it was decided to make use of Template Haskell through a new ADT: `HDFun`. `HDFun` is covered in section 3.4.

- **Polymorphism**.

  Even if the Lava's signal type is polymorphic, only monomorphic `Int` and `Bool` signals can be used in practice.

  On the other hand ForSyDe's original `Signal` is polymorphic and so are its process constructors. There is no automatic way to transform every kind of signal value into VHDL code, and thus, the subset of supported ForSyDe signals has to be controlled.

  `HDPrimType` type-class constraints were used in process constructors as way to control the supported signal subset. The details about the `HDPrimType` typeclass are described in section 3.5.

The next sections give explicit details about how Lava's embedded compilation model was adapted to ForSyDe.

In order to make it more intuitive for the reader. Each section incrementally refers to the speci c changes which were made to `mapSY`, originally de ned in ForSyDe's Library as:

```
mapSY :: (a -> b) -> Signal a -> Signal b
mapSY _ NullS    = NullS
mapSY f (x:-xs) = f x :- (mapSY f xs)
```

Remember that the de nition of `Signal` is isomorphic to Haskell lists and is de ned as:

```
data Signal a = NullS
              | a :- Signal a
```

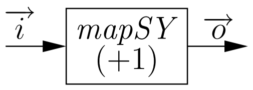


Figure 3.1: *plus1*

To illustrate how the design modi cations a ect the end-user, the transformations su ered by the original de nition of a simple system will be used as example. *plus1* (gure 3.1) is a trivial system which merely adds 1 to its numerical input and returns the resulting value. Its code, using the original ForSyDe Library is:

```
plus1 :: Num a => Signal a -> Signal a
plus1 = mapSY (+1)
```

3.3 `HDSignal`: The Hardware Description Signal

In order to avoid regressions, instead of modifying the original `Signal` type, an alternative signal type, `HDSignal` (*Hardware Description Signal*) was introduced.

`HDSignal` works similarly to signals in Lava. It is an ADT hidden to the user which *secretly* stores the structure of the circuit.

`HDSignal` makes use of *Observable Sharing* [31] (introduced in chapter 2) to achieve its goal. Understanding the concrete implementation details of *Observable Sharing* requires an advance knowledge of Haskell which is out of the scope of this report. Therefore, `HDSignal` will simply be treated as an abstract type with unknown implementation.

```
data HDSignal a = ... -- hidden
```

Providing an alternative version of `Signal` required as well to provide new process constructors, which were initially renamed to avoid name-clashes.

Thus, an alternative de nition of `mapSY` using `HDSignal` was created:

```
hdMapSY :: (a->b) -> HDSignal a -> HDSignal b
hdMapSY ... -- hidden implementation
```

plus1 also needed to renamed and modi ed to make use of `HDSignals`

```
hdPlus1:: Num a =>  HDSignal a -> HDSignal b
hdPlus1 = hdMapSY (+1)
```

3.4 HDFun: the Hardware Description Function

Using the de nition of previous section, hdMapSY takes a standard Haskell function as argument:

```
hdMapSY :: (a->b) -> HDSignal a -> HDSignal b
```

However, there is no possible way of generating VHDL code from a (a->b) value. Instead, a new type, HDFun (*Hardware Description Function*), capable of storing the de nition of a function was introduced.

HDFun, contrary to a standard Haskell function, contains a syntax tree thanks to the use of Template Haskell.

3.4.1 Introduction to Template Haskell

TH [40] (*Template Haskell*) was previously mentioned in chapter 2. It is a Haskell extension that enables to use compile-time meta-programming. Unfortunately, as it was stated before, TH is only currently supported by GHC.

Meta-programming allows to write programs which manipulate and/or generate other programs. In Template Haskell, that is done by processing the AST (*Abstract Syntax Tree*) of Haskell declarations or expressions.

In the case of HDFuns, Template Haskell gives access to the AST of function declarations, allowing ForSyDe's embedded compiler to translate them to VHDL.

The key abstractions that Template Haskell operates on are Expressions, Declarations, and Types. Fragments of concrete Haskell code can be *lifted* into the meta-world through the use of quasi-quotations. Expression, declaration and type fragments each have their own variation on the quotation syntax:

```
[|  expr |] -- lifts a concrete expression
[d| decl |] -- lifts a concrete declaration
[t| type |] -- lifts a concrete type
```

As a result of lifting, the AST of the quoted expression, declaration or type is obtained for later processing.

In order to generate code, Template Haskell provides *splices* which are executed at compile-time and return an AST. GHC automatically merges the resulting AST with the rest of the source code as if the programmer actually wrote it. Informally, it can said that splices work like C macros, but are type-safe, ensuring the generation of correct code.

Splices have their own notation. A splice is written $x, where x is an identi er, or $(...), where ... is an arbitrary Haskell expression. This use of $ overrides its meaning as an in x operator. To be interpreted as an operator, $ needs to be surrounded by spaces.

The biggest advantage of using Template Haskell is the reutilization Haskell syntax, which saves the designer from learning a new embedded language. As a drawback, TH introduces syntax extensions (quotes and $) which anyway require certain learning, and makes ForSyDe GHC-dependant.

3.4.2 HDFuns in practice

With the introduction of HDFun, the type of hdMapSY changes to

```
hdMapSY :: HDFun (a->b) -> HDSignal a -> HDSignal b
```

which is still quite similar to the original de nition of mapSY.

Furthermore, *plus1* needs as well to be readapted:

```
hdPlus1 :: Num a => HDSignal a -> HDSignal a
hdPlus1 = hdMapSY doPlus1
 where doPlus1 = $(mkHDFun [d| doPlus1 :: Num a => a -> a
                              doPlus1 a = a + 1          |])
```

The internal HDFun of hdPlus1 is generated at compile-time as follows:

1) The declaration of doPlus1 is lifted to its AST thanks to the enclosing brackets ([| ... d|]).

2) Its AST is processed by a splice in which the mkHDFun constructor function creates the HDFun.

It is worth to note that doPlus1 is de ned as

```
doPlus1 a = a + 1
```

instead of simply

```
doPlus1 = (+1)
```

due to the limitations of mkHDFun.

Those limitations are:

- The number of formal parameters in the function must equal its number of arguments.
- The signature of the function is mandatory and only one de nition clause is admitted.
- Pattern matching is only supported with literals, variables and the wildcard _ pattern. No other kind of pattern matching is allowed.
- where clauses are not supported.

- The only valid expressions are: variables, constructors, in x operations (excluding in x constructors and sections), `if`, `case` and the expresions resulting form their combination. `let`, lambda abstractions, and the rest kinds of expressions are not supported.

These limitations can be considered excessive. However, they provide a su cient inital functionality. Thanks to the use of TH, this limitations can be easily overcome by extending the supported Haskell subset of `HDFun` (see A for details).

3.5 Controllling polymorphism: the `HDPrimType` type class

As it was previously said, there is no automatic way to transform every Haskell type into VHDL, and thus, the subset of supported ForSyDe signals has to be controlled somehow.

Indeed, only instances of `HDPrimType` (*Hardware Description Primitive Type*) can be handled by the compiler. Currently, only `Int` and `Bool` instantiate `HDPrimType`.

The process constructors have to reflect that limitation. For that reason, a `HDPrimType` constraint was introduced in each of its parameters. In the case of `hdMapSY`, its type changes to:

```
hdMapSY :: (HDPrimType a, HDPrimType b) =>
           HDFun (a->b) -> HDSignal a -> HDSignal b
```

That means as well that *plus1* can no longer support numerical signals in general. A concrete type has to be speci ed so that the compiler can chose an adequate VHDL representation.

```
hdPlus1 :: HDSignal Int -> HDSignal Int
hdPlus1 = hdMapSY doPlus1
 where doPlus1 = $(mkHDFun [d| doPlus1 :: Int -> Int
                               doPlus1 a = a + 1          |])
```

The implementation details of `HDPrimType` are not signi cant. However, it is worth to note that it shares certain similarities with Haskell's `Data.-Typeable.Typeable` class. Thus, `HDPrimType` can be considered a particular implementation of dynamic types.

3.6 Integration with ForSyDe's Library

So far, the original mapSY function has been renamed to hdMapSY and modi ed to include all the requirements of the embedded compiler.

Both functions use di erent types but are intended to attain a similar goal. Therefore, it would be desirable to make them share the same name (mapSY), without causing a name-clash.

Thanks to a Haskell extension called MPTC [43] (*Multi Parameter Type Classes*), it is possible to make them share mapSY as their name:

```
class SynchronousM f_a_b signal a b | signal a b -> f_a_b where
 mapSY :: f_a_b -> signal a -> signal b

instance SynchronousM (a->b) Signal a b where
  mapSY _ NullS = NullS
  mapSY f (x:-xs)       = f x :- (mapSY f xs)

instance (HDPrimType a, HDPrimType b)
  => SynchronousM (HDFun (a->b)) HDSignal a b where
 mapSY = -- hidden details
```

SynchronousM is a multi-parameter type class. As its name clearly indicates, a MPTC, in contrast to a traditional type class, admits multiple parameters and serves as a way of associating various types with a group of operations.

However, the use of multiple parameters leads to ambiguity problems in the type inference algorithm (i.e. makes di cult to resolve whether a set of types conform a MPTC instance or not). Thus, it is necessary to help the type inferer with certain constraints named functional dependencies. In this case, signal a b -> f_a_b is a functional dependency which tells the inferer that signal, a and b depend on f_a_b, or, in other words, providing the signal, a and b parameters of a SynchronousM instance, the inferer can automatically determine f_a_b because the dependency guarantees that it will be unique.

SynchronousM permits to use the name mapSY both for the original Signal process constructor and the HDSignal one. The same scheme was applied to the other two basic process constructors: delaySY and zipWithSY, leading to the SynchronousD and SynchronousZ MPTCs.

The use of MPTCs has a sweet and intended consequence. The existent code which purely relies on the aforementioned process constructors behaves nicely both for the Signal and HDSignal worlds without any modi cations.

For example, sourceSY is derived from mapSY and delaySY, and its original de nition is

```
sourceSY f s0 = o
 where o = delaySY s0 s
```

```
       s = mapSY f o
```

which indeed works for both Signal and HDSignal since its type is

```
sourceSY :: (SynchronousD signal a, SynchronousM f_a_b signal a a) =>
            f_a_b -> a -> signal a
```

which satis es both

```
sourceSY :: (a->a) -> a -> Signal a
```

and

```
sourceSY :: HDPrimType a => HDFun (a->a) -> a -> HDSignal a
```

Thanks to MPTCs the new signal type HDSingal was easily integrated with the original ForSyDe Library.

3.7 Improvements over Lava's original design

Chapter 2 analyzes some features which are lacked by Lava but which would certainly be helpful in hardware design: *Ports* and *Hierarchical Structures*. The following two sections explain those concepts in deeper detail and describe how they were incorporated in ForSyDe.

3.7.1 Ports

A Port serves as an interface between the system and the outside world. Ports *hold* signals which the system can accept and produce.

They work similarly to VHDL's *port clause* within an *entity declaration*. However, instead of allowing to mix input and output signals in the same port, it was decided to distinguish between *Input Ports* (gure 3.2a) which provide incoming signals to the system and *Output Ports* (gure 3.2b) which forward output signals from the system to the outside world.

A port in ForSyDe has an associated descriptor which indicates the name and signal types of the port. The constructor functions mkInPort and mkOutPort are in charge of creating a *Input* (*Output*) *Port*.

In the case of the trivial hdPlus1 system, its input and output ports are:

```
-- hdPlus1 input port
plus1In :: InPort
plus1In = mkInPort 1 [("plus1Input", Int)]

-- hdPlus1 output port
plus1Out :: OutPort
plus1Out = mkOutPort 1 [("plus1Output", Int)]
```

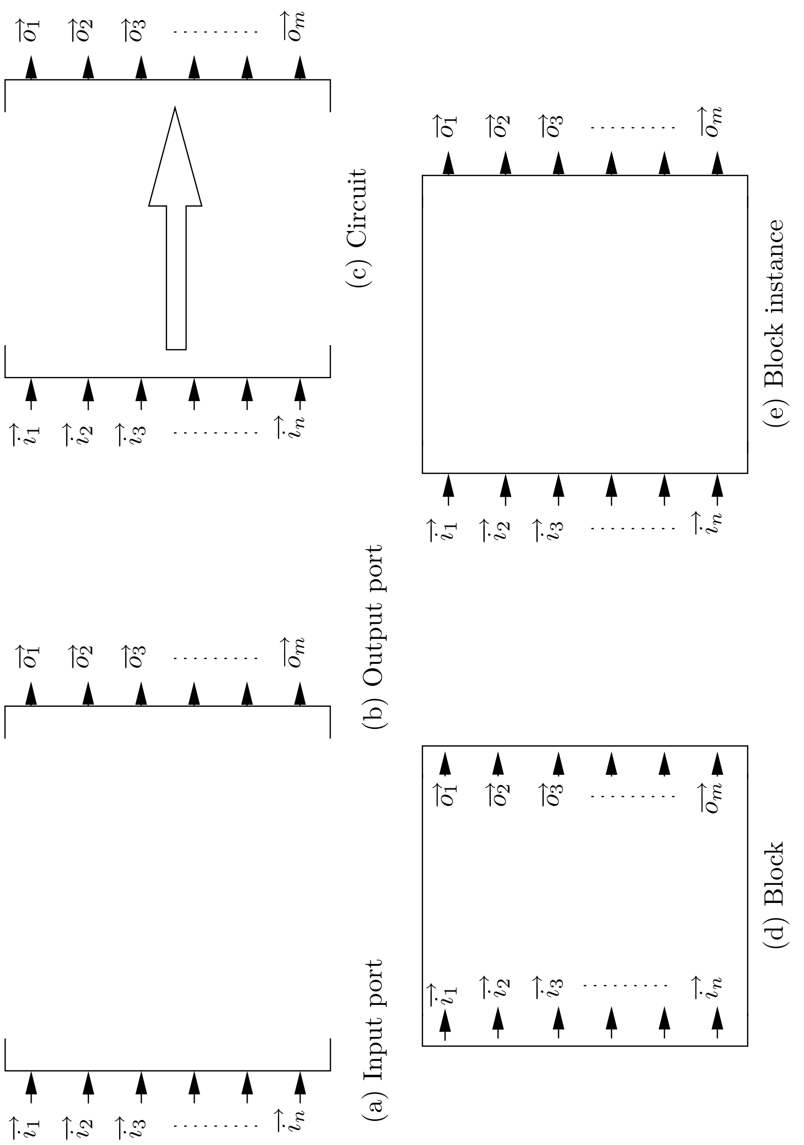

$\vec{i_1}$ $\vec{i_2}$ $\vec{i_3}$ .... $\vec{i_n}$
$\vec{o_1}$ $\vec{o_2}$ $\vec{o_3}$ .... $\vec{o_m}$
(a) Input port
(b) Output port
(c) Circuit
(d) Block
(e) Block instance

Figure 3.2: Hierarchical structures

The main advantage of using ports is readability and organization, which is better observed in bigger systems (i.e. with several inputs and outputs). However, this example is enough to understand the concept of ports.

3.7.2 Hierarchical structures

The *Circuit*, *Block* and *Block Instance* structures provide ForSyDe with hierarchical design capabilities, in a similar way as *components* and *port maps* do in VHDL.

3.7.2.1 Circuit

A circuit (gure 3.2c) is modelled as a function which takes an input port, processes the signals provided by the port and nally generates output signals which are stored in an output port, hence its type

```
type Circuit = (InPort -> OutPort)
```

The following code builds the `Circuit` for the *plus1* system

```
plus1Circ :: InPort -> OutPort
plus1Circ ip = supplySig  plus1Sig "plus1Output" plus1Out
  where plus1Sig = plugSig "plus1Input" ip hdPlus1
```

The code reads as *Given an input port* ***`ip`****, supply* ***`plus1Sig`*** *to entry* ***`"plus1Output"`*** *of output port* ***`plus1Out`****, where* ***`plus1Sig`*** *is the result of plugging the process constructor* ***`hdPlus1`*** *to the entry* ***`"plus1Input"`*** *of* ***`ip`***

`supplySignal` is useful to supply a signal to an output port whereas `plugSignal` *plugs* a process constructor into one the signal *sockets* of an input port.

3.7.2.2 Block

Circuits are not useful by themselves. They need to be transformed into a *Block* (gure 3.2d) before running the compiler to generate VHDL.

A *Block* is a white-box which de nes a sub-system. Its internals are known, but it is isolated from the outside world. All its content exists but is not accessible, there is no way to directly connect a *Block* to other system structures. It is similar to an *entity-architecture* pair in VHDL.

`plus1Circ` can be transformed to a *Block* by using the `mkBlock` constructor.

```
plus1Block :: Block
plus1Block = mkBlock "plus1" plus1In plus1Circ
```

mkBlock takes the name of the block, its input port and a circuit, creating the corresponding Block.

To translate the model to VHDL, one can simply use the writeVHDL function, which calls the VHDL compiler.

```
> writeVHDL plus1Block
Writing VHDL code to plus1.vhd ... done!
```

Later, the designer could use that VHDL description to synthesize hardware. The full Haskell source of the *plus1* model, its VHDL translation and a RTL-level hardware model obtained from it, can be seen in appendix B.

3.7.2.3 Block Instance

A *Block Instance* (gure 3.2e) is the equivalent of a *component* in the VHDL world.

A *Block* is a white-box, an isolated section of the system whose internals are known. On the other hand, a *Block Instance* is functionally equivalent to a *Block*, but behaves like a black-box: its content is unknown, but its input and output ports are viewable from the outside world, making it connectable to the rest of the system.

In order to obtain a *Block Instance* from a *Block*, the *Block* needs to be instantiated.

```
plus1Ins :: BlockIns
plus1Ins = instantiate plus1Block
```

The code above creates an instance from plus1Block. Now the functions plugSig and supplySig could be used directly with the plus1Ins to interconnect its ports with other parts of a system.

Chapter 4

User's tutorial

This chapter is aimed at introducing the end-user to the compiler and its API.

First, a simple *identity* system will be used to show how connections are made. Then, a reasonably complex example gives further details on how to use the compiler.

4.1 Prerequisites

In order to be able to follow this tutorial it is required to have a working copy of:

- ForSyDe Standard Library with compiling support[1].
- GHC version 6.6 or higher.
- Haskell's `mtl` (Monad Template Library)[2].

4.2 Identity system

As an example of how to make connections within a model, an *Identity* system (gure 4.1) will be built.

The system could not be simpler: It has six inputs and six outputs which are connected in parallel.

First, the name of the module is declared and the required libraries are imported.

[1]It can be downloaded from `http://www.imit.kth.se/info/FOFU/ForSyDe/HDForSyDe/`.

[2]Normally included in the GHC distributions. It can be downloaded from `http://hackage.haskell.org/`.

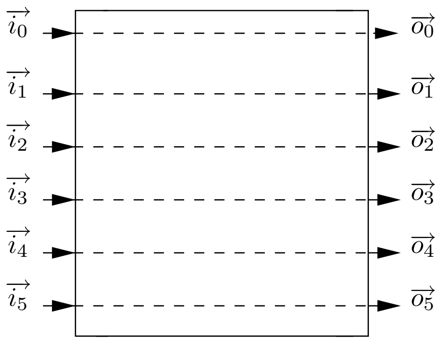


Figure 4.1: Identity system

```
module Identity
import HD
```

Then the input and output ports are de ned.

```
idIn :: InPort
idIn = mkInPort 6 [("input0",Int),
                   ("input1",Int),
                   ("input2",Int),
                   ("input3",Bool),
                   ("input4",Bool),
                   ("input5",Bool)]


idOut :: OutPort
idOut = mkOutPort 6 [("output0",Int),
                     ("output1",Int),
                     ("output2",Int),
                     ("output3",Bool),
                     ("output4",Bool),
                     ("output5",Bool)]
```

The type of the ports is not signi cant, any other type combination would have been valid.

Using the full power of Haskell, those ports could have been de ned as

```
iname  = "input"
oname  = "output"
nInts  = 3
nBools = 3

(idIn,idOut) = (portDesc iname, portDesc oname)
  where types          = replicate nInts Int ++ replicate nBools Bool
        indexes        = iterate (+1) 0
        portDesc  name = zipWith3 join (repeat name) indexes
        join  id ix t  = (id  ++ show ix, t)
```

permitting to easily change the number of inputs/outputs and their names. This example can be a bit obscure, but shows how the embedded compilation model permits making use of Haskell features.

Once the ports are de ned, the circuit description can be written. A way to do so is using the `connectIx` function, which connects an input-port signal with an output-port one, and returns the modi ed output port.

```
idCirc :: Circuit -- (InPort -> OutPort)
idCirc inP = (connectIx "input5" inP "output5".
              connectIx "input4" inP "output4".
              connectIx "input3" inP "output3".
              connectIx "input2" inP "output2".
              connectIx "input1" inP "output1".
              connectIx "input0" inP "output0") idOut
```

`connectIx` can use the signal identi ers as index, but it can also use numerical indexes or both.

```
idCirc :: Circuit
idCirc  inP = (connectIx 5          inP "output5" .
               connectIx "input4"   inP 4         .
               connectIx 3          inP 3         .
               connectIx "input2"   inP "output2" .
               connectIx "input1"   inP "output1" .
               connectIx "input0"   inP "output0" ) idOut
```

The advantage of using numeric indexes is that they work with an input port independently of its signal identi ers.

`connectIx` is easy to understand but makes the design quite verbose.

Subports are a better option. A subport is a range of signals within port. The identity circuit can be built making use of them.

```
idCirc :: Circuit
idCirc inP = connectSP
                  ("input0" , "input5" ) inP
                  ("output0", "output5") idOut
```

`connectSP` is used to connect signals `"input0"` to `"input5")` at the input port with signals `"output0"` to `"output5"` at the output port.

`connectSP` also admits mixing numbers and identi ers in the same way as `connectIx`:

```
idCirc :: Circuit
idCirc inP = connectSP ("input0", "input5")  inP
                       (0       , 5       )  idOut
```

Another possible option is to omit the subport of the origin or destination. In way, it will be implicitly assumed that all the signals of the of the input (output) port take part in the connection.

One can omit the input subport . . .

```
idCirc :: Circuit
idCirc3 inP = connectFrom inP
                         ("output0", "output5") idOut
```

. . . or the output subport

```
idCirc :: Circuit
idCirc inP = connectTo (0, 5) inP
                       idOut
```

Lastly, there is a more elegant way to connect all inputs and outputs of two ports at once.

```
idCirc :: Circuit
idCirc inP = inP ‘connect‘ idOut
```

So far the circuit of the system was de ned. However, the circuit needs to be transformed to a `Block` before translating the design to VHDL.

```
idBlock :: Block
idBlock = mkBlock "identity" idIn idCirc
```

`mkBlock` takes an identi er, an input port and the circuit, and returns the corresponding `Block`.

4.2.1 Compiling the model

To compile the model to VHDL simply

1) Save the model in a le named `Identity.hs`

2) Execute `ForSyDe Identity.hs` making sure that the `bin/` directory of ForSyDe's Library is in the execution path[3].

3) From `ghci`'s prompt, type

```
*Identity> writeVHDL idBlock
Writing VHDL code to identity.vhd ... done!
```

4.3 A more complex system

Figure 4.2 illustrates a still trivial but slighty more complex system descripo- tion.

[3]If you are a windows user this does not apply to you. Load the model in `ghci` making sure that the `src/` directory of ForSyDe s library is in the import directory list (`-i` ag).

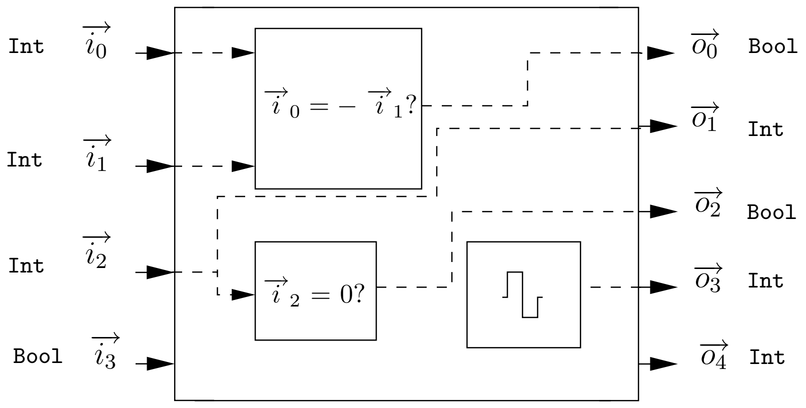

Int $\overrightarrow{i_0}$
Int $\overrightarrow{i_1}$
Int $\overrightarrow{i_2}$
Bool $\overrightarrow{i_3}$
$\overrightarrow{i}_0 = -\overrightarrow{i}_1?$
$\overrightarrow{i}_2 = 0?$
$\overrightarrow{o_0}$ Bool
$\overrightarrow{o_1}$ Int
$\overrightarrow{o_2}$ Bool
$\overrightarrow{o_3}$ Int
$\overrightarrow{o_4}$ Int

Figure 4.2: A more complex system

- It has 4 inputs and 5 outputs.
- The system checks if $\overrightarrow{i_0}$ and $\overrightarrow{i_1}$ are opposite and connects the result to $\overrightarrow{o_0}$.
- Signal $\overrightarrow{i_2}$ is compared with 0. The result is connected to $\overrightarrow{o_2}$.
- Input $\overrightarrow{i_2}$ is directly connected to $\overrightarrow{o_1}$.
- Output $\overrightarrow{i_3}$ is connected to a cycle counter.
- The rest of inputs and outputs remain unused.

First, the name of the module is declared together with the import list

```
{-# OPTIONS_GHC -fth #-}
-- -fth is required due to the use of Template Haskell
module MoreComp where
import HD
import SynchronousLib (mapSY, zipWithSY, sourceSY)
```

Then, the ports of the system are declared:

```
MCIn = mkInPort 4 [("in0",Int),
                   ("in1",Int),
                   ("in2",Int),
                   ("in3",Bool)]

MCOut = mkOutPort 5 [("out0",Bool),
                     ("out1",Int),
                     ("out2",Bool),
                     ("out3",Int),
                     ("out4",Int)]
```

Then, the code related to the computations performed within the system is written.

This function performs the comparison with zero:

```
isZero :: HDSignal Int -> HDSignal Bool
isZero = mapSY doIsZero
 where doIsZero = $(mkHDFun [d| doIsZero :: Int -> Bool
                                doIsZero a = a == 0 |])
```

Note the use of Template Haskell.

In order to check if two values are opposite, rst they will be added and then, the result will be compared with zero (for which `isZero` is reused).

```
plus :: HDSignal Int -> HDSignal Int -> HDSignal Int
plus = zipWithSY doPlus
 where doPlus = $(mkHDFun [d| doPlus :: Int -> Int -> Int
                              doPlus a b = a + b          |])


areOpposite :: HDSignal Int -> HDSignal Int -> HDSignal Bool
areOpposite a b = isZero (plus a b)
```

The only computing part of the system is the cycle counter:

```
cycleCounter :: HDSignal Int
cycleCounter = sourceSY plus1 1
     where plus1 = $(mkHDFun [d| plus1 :: Int -> Int
                                 plus1 i = i + 1     |])
```

The initial value of the counter is 1. Once the system is started `plus1` will be executed in every cycle, incrementing the value of the counter.

The only thing left to complete the system is creating the connections within the circuit and producing a block from where to generate VHDL.

The circuit can be de ned as follows:

```
MCCircuit :: InPort -> OutPort
MCCircuit ip =
   (supplySig (plugSig2 "in0" "in1" ip areOpposite) "out0".
   supplySig (plugSig  "in2"       ip isZero     ) "out2".
   connectIx "in2" ip                              "out1".
   supplySig cycleCounter                          "out3") MCOut
```

`connectIx` was already introduced in previous section. However, there are some new functions which must be commented:

- `supplySig` supplies a signal to an output port. The output port changes and for that reason a new port is returned.
- `plugSig` plugs a function to a signal coming from an input port.
- `plugSig2` is the two-argument variant of `plugSig`. It plugs a two-argument function to two signals of an input port.

Finally, a `Block` is created from which the model can be translated to VHDL.

```
MCBlock :: Block
MCBlock = mkBlock "MC" MCIn MCCircuit
```

```
*MoreComp> writeVHDL MCBlock
Writing VHDL code to MC.vhd ... done!
```

Chapter 5

Conclusions and further work

5.1 Conclusions

ForSyDe provides a methodology to design systems with a high level of abstraction in mind. An initial implementation of ForSyDe, based upon Haskell, was available before the research related to this thesis began. However, the two key stages of ForSyDe's design flow, *Transformational Design Refinement* and *Implementation Mapping* remained unimplemented. As a result of this thesis, ForSyDe now counts with a new VHDL translator which automates the *Implementation Mapping* phase.

An initial study was made to try to take advantage of previous work in the Hardware Design and Functional Programming eld. Consequently, the compiler is inspired on Lava, a successful hardware design environment based upon Haskell too.

The embedded approach, adopted from Lava, has allowed to develop a full compiler during the limited time frame of a master's thesis. On the other hand, implementing a stand-alone Haskell compiler or a customized backend would have been unfeasible, given the associated complexity of such a task together with the time and man-power limitations. In ideal conditions, even if such a compiler was developed, it would certainly had been di cult to maintain. With the embedded approach, the compiler is integrated with ForSyDe's library, making it maintainable, easy to distribute and independent of third party tools.

Furthermore, the compiler was designed to be extensible, providing a framework where to experiment with ForSyDe and easily include other backends such as simulation, veri cation, graphical representation of ForSyDe models etc. . . . In order to ease the incorporation of new developers, the compiler's use and implementation was thoroughly documented (see chapter

4 and appendix A).

In addition to the initial goals of the thesis, an e ort was made add features lacked by Lava to the compiler. Lava, at the moment of writing this report, does not provide a way to create hierarchical designs (in the sense of reusing hierarchical components), and lacks the concept of ports, which make models more intuitive and readable. The compiler implemented during this thesis has overtaken those limitations by including three new structures: *Port*, *Block* and *Block Instance* which where described in chapter 3. Those features have been needed in Lava for a long time. As a matter of fact, Koen Claessen, a member of Lava's research group, is currently developing a new version of Lava in which components are explicit, making them accessible to the designer.

Even if the compiler is based on Lava, the characteristics of ForSyDe's implementation have made impossible to simply replicate Lava's design. ForSyDe is much more behavioural, whereas Lava is purely structural, its functions are monomorphic and the type of signals accepted by the language are limited and hardcoded in the compiler. To overcome those di erences, the compiler makes use of metaprogramming through Template Haskell in order to access the AST (*Abstract Syntax Tree*) of the design. Curiously, in parallel to this thesis a similar solution [44] was designed by IBM's Andrew Martin for the OCaml programming language. Martin's solution makes use of MetaOCaml, which shares certain similarities with Template Haskell.

5.1.1 Goal analysis

In order to objectively conclude whether the goals of this thesis where achieved, this section summarizes the obtained results based on the goals outlined in chapter 1.

1) *Study of ForSyDe and related work relevant to this thesis.*

   In the initial phase of this thesis, a deep study of ForSyDe and its implementation took place. As a result, chapter 1 tries to introduce ForSyDe to the reader in a friendly way.

   Then, a big e ort was devoted to analyze the previous work in the Functional Programming and Hardware Design eld, in order to make use of it during the development of the compiler. Chapter 2 gives an overview of it. Many of the features described in that chapter ended up incorporated in the compiler.

2) *Definition of a relevant subset of Haskell that is accepted by the synthesis.*

   With the exception of `HDFun`, the whole Haskell standard is supported by the compiler, thanks to its embedded design. The concrete limitations of

`HDFun` are described in chapter 3. However, due to the use of Template Haskell, it should be possible to easily extend the supported language subset.

Another unavoidable limitation of the compiler is the set of signal types it can deal with. Due to the time and man-power restrictions of the thesis it was decided to only support `Int` and `Bool` signals. However, in the same way as `HDFun`, the signal-type subset of the compiler was implemented in a way which makes it fairly easy to extend (see appendix A).

3) *Development of the synthesis tool according to [1].*

   [1] presents a template-based translation approach, which was not implemented before this thesis took place.

   The template approach was considered and later replaced with a AST-driven implementation for various reasons:

   - Templates lead to an obscure application design.

     Using simple text to represent the structure of the target language made the compiler design unorganized and more di cult to understand. A structured XML approach would have been substantially better (at least regarding the organization of the program), but it would have entailed transforming XML to usable VHDL at certain point, causing an undesired overhead.

   - Lack of flexibility.

     Using in-memory data structures to represent the target language makes the translation more flexible and easy to extend. Modifying or extending the AST representation of VHDL is easier and cleaner than dealing with plain text.

4) *Evaluation of the tool, identifying and including possible improvements.*

   The tool was tested with a set of source examples written for that purpose. However, due to the research nature of the tool and the lack of a user-base, it was not possible to check if the compiler suits the needs of future users.

   Nevertheless, the compiler was later reviewed and additional features such as circuit ports and hierarchical design were identi ed and incorporated in the compiler.

5) *Detailed documentation of the tool.*

   This thesis reflects the big e ort done to exhaustively document the compiler. Chapters 3 and 4 describe the design and use of the compiler, while appendix A is intended to give implementation details for future developers.

5.2 Further work

The compiler presented in this thesis allows to synthesize hardware from ForSyDe speci cations and establishes a development framework where to experiment with ForSyDe. However, the compiler can still only considered a research tool. In order to obtain a realistic development environment there is still a lot of work to be done. The following list proposes general and speci c tasks which would help to transform the compiler into an industrial-strength solution.

- **Automatization of the *Transformational Design Refinement* stage**

  The compiler implements one of the two main phases of ForSyDe's design flow, the aforementioned *Implementation Mapping* stage. However, the *Transformational Design Refinement* stage has not still been automated. This stage is essential, since it is the way to bridge the high abstraction level imposed by ForSyDe, taking advantage of ForSyDe's strong formal base. Thus, it of major importance to put it in practice through an appropriate tool. A good way to do it, would be extending the compiler by adding an initial transformation phase.

- **Development of a test suite**

  During the development of the compiler, several sample design models were tested. However that cannot be considered a way to test the compiler thoroughly.

  For that reason, it is advisable to create a robust test suite to detect bugs which remained unnoticed or were introduced during development.

- **Inclusion of new backends**

  To date, the compiler can only process a ForSyDe model in order to perform a translation to VHDL. However, other backends would certainly be useful:

  - **Simulation**. Currently, a model developed with `HDSignals` can only be simulated from its VHDL translation. A simulation backend would allow to avoid translating to VHDL and to test models without leaving ForSyDe's development environment. Furthermore it could be used to provide interoperability between the `Signal` and `HDSignal` types.

  – **Graphical representation backend**. This backend would allow to generate a graph of the ForSyDe models, which could be used to easily and automatically document them.

- **Development of a graphical frontend**

  The introduction of a graphical CAD (*Computer Aided Design*) tool would allow to design models by simply and visually connecting process constructors and thus, it would save the designer from having a functional programming background, attracting new users.

- **Support for new signal types**

  The compiler can only deal with signals carrying `Int` and `Bool` values. In order to support the full ForSyDe speci cation [1] it would be necessary to extend that subset, including, for instance, enumerated types. Appendix A gives guidelines to perform this extension.

- **Extension of `HDFun`'s Haskell subset**

  Current limitations of `HDFun`s are outlined in chapter 3. Their supported Haskell subset makes the compiler functional but is probably not enough for a realistic production environment. Again, appendix A documents how to extend `HDFun`.

- **Development of `Block` and `HDFun` combinational libraries**

  Process constructors are easily combinable, through, for example, sequential and parallel composition. The same idea could be applied to the `Block` and `HDFun` types by implementing appropriate combinational libraries.

- **Promotion of ForSyDe**

  Many research projects like ForSyDe remain unknown to the main public due to their closed development model. In order to promote a broader adoption and development it would be necessary to make ForSyDe more appealing for, at least, the Haskell community. That entails opening its development together with making it easier to install, distribute and develop. A few modi cations in ForSyDe's development model would be advisable:

- Document ForSyDe with Haddock[1], the most used Haskell documentation system.
- Create an open mailing list where to discuss the development of ForSyDe and give support to its users.
- Make the latest development version of ForSyDe available through a version control system (Darcs[2] is the most widely-used within the Haskell community).
- Package ForSyDe's stable versions with Cabal[3] and upload them to HackageDB[4] in order to make them more accessible and easier to install. It is worth to note that including the project in HackageDB would require to change the module naming scheme of ForSyDe, taking global hierarchical naming in account, to avoid name clashes with other projects.

[1]`http://www.haskell.org/haddock/`

[2]`http://www.abridgegame.org/darcs/`

[3]`http://www.haskell.org/cabal/`

[4]`http://hackage.haskell.org/`

Appendix A

Hacker's guide to the compiler

Master's theses are, in many cases, just a small piece of e ort taking part in a more-widely scoped project. They generally lead to immature research results which might need to be further-polished or, quite oftenly, consist in continuing the work of yet another thesis. Thus, they are generally subject to be extended and modi ed.

However, when it comes to practical results such us source code, they tend to be weakly documented, hindering the rst steps of a potential future developer.

The main purpose of this guide is to help contributors getting involved with the implementation details of the compiler and to save them having to gure out the steps required to carry out the most common extensions.

A.1 Prerequisites

In order to understand the compiler's code, a developer is obviously required to be fluent in Haskell98 [3]. It is also advisable to have some experience with `mtl` (*Monad Transformer Library*) which is extensively used throughout the sources.

Furthermore, in order to understand the full source code tree, it is assumed that he (or she) will be familiar with common Haskell extensions such as MPTC [43] (*Multi-parameter Type Classes*), Existential Types and Template Haskell [40], which unfortunately, by the time of writing this thesis are not still documented in a friendly way. The best current resource to get familiar with them is the GHC User's Guide [45]. However, the upcoming publication of a promising (and free) book, *Real-World Haskell* [6], covering those extensions will certainly be helpful. Even though, there still could be certain small tasks whose accomplishment should not necessarily rely on the aforementioned extensions.

Being familiar with VHDL is speci cally required to understand the copmiler's VHDL backend. It is advisable to have access of the VHDL93 [2] standard, or other exhaustive reference material before trying to extend this backend.

The compiler was developed and tested with GHC version 6.6 under a GNU/Linux system[1]. The use of non-standard extensions make it GHC-dependant and will eventually lead to regression problems which will have to be addressed before performing extensions or even making use of it.

Lastly, this guide is not self-contained, it complements the rest of the thesis (especially chapter 3) and was isolated in an appendix due to covering speci c implementation details not considered of general-interest. Therefore, in order to understand this guide, it will be assumed that the reader had already gone through the rest of the thesis report.

A.2 Compiler modules overview

Figure A.1 contains the module dependency graph of the compiler.

`HD` is the main module, it stands for *Hardware Description* and its purpose is simply to re-export other modules (simplifying the importing task for the end-user) and to hide certain implementation details to the outside world.

The rest of the modules will be explained in the next sections, where they are grouped according to their functionality.

It is highly advisable to follow the same module-order when getting familiar with the compiler in order to be able to understand it in an incremental manner.

A.2.1 Core modules

They constitute the heart of the compiler and are the ones to be studied in rst place.

- **`HD.Types`**. This module contains the mechanisms and de nitions related to the possible value-types which a signal (or more accurately an `HDSignal`) can carry[2] and the type-representation of a function (`HDFunType`).

  Basically, a signal can transport values of any type belonging to the class `HDPrimType`, standing for *Hardware Description Primary Type.*

[1]Although it was not tested under other operating systems, any development environment with GHC support could in theory be used.

[2]`Bool` and `Int` at the moment of writing this appendix.

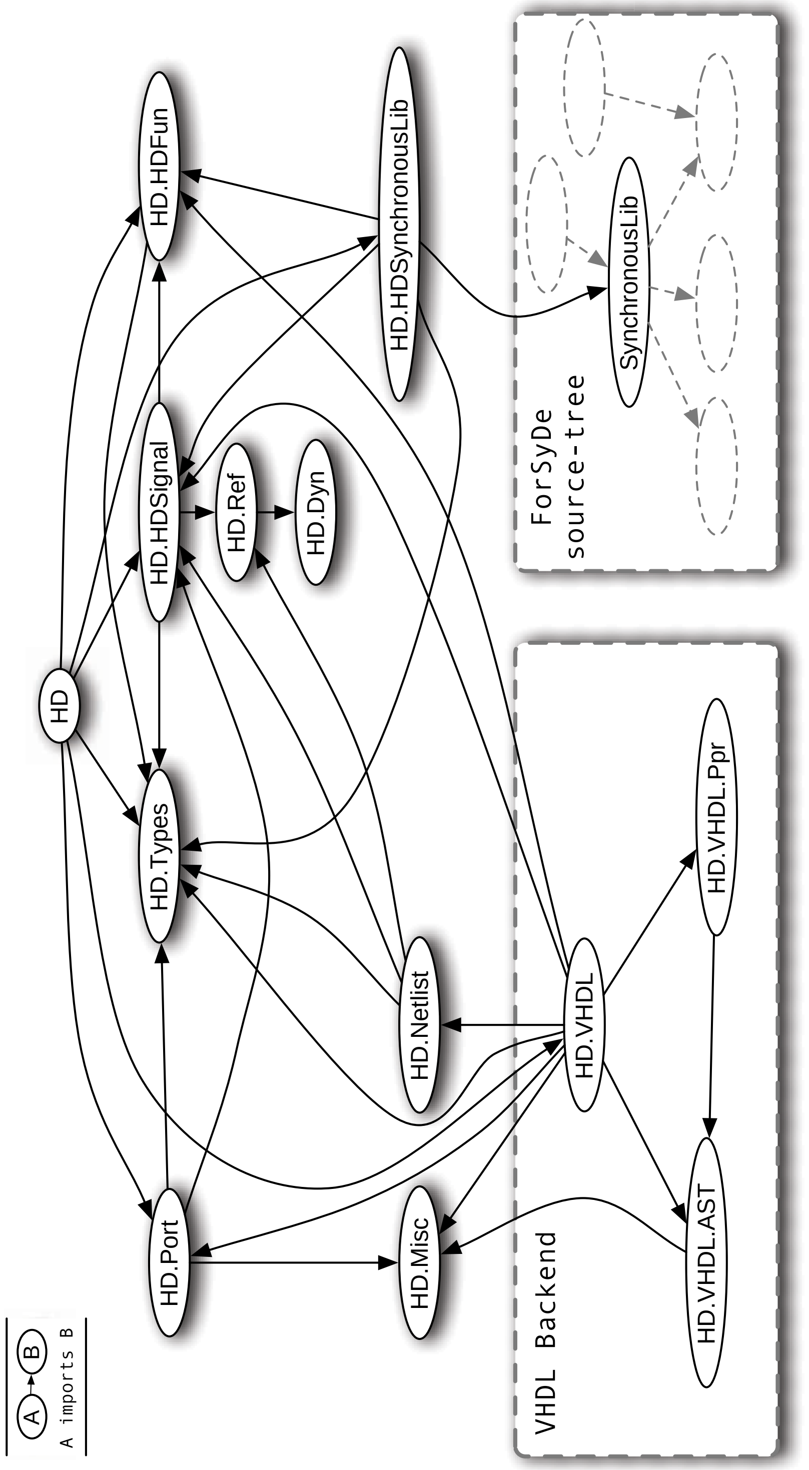


Figure A.1: Compiler modules' dependency graph

- **`HD.HDSignal`**. It is, without doubt, the most important module. It contains the de nition of the user-hidden *Hardware Description Signal.*

  As it was previously mentioned, an `HDSignal` *secretly* stores the structure of the circuit for latter processing, it is the compiler's intermediate representation of a circuit-description, and thus, it is the only usable information which a backend can work with.

  An `HDSignal` uses the same signal approach as Lava [10], called *Observable Sharing* [31] which was earlier introduced in chapter 2. Actually, both the modules **`HD.Ref`** (the Observable Sharing implementation) and **`HD.Dyn`** (unsafe dynamic types, used internally in `HD.Ref`) were taken and adapted from the Lava distribution.

  It is encouraged to fully understand the details of this module before attempting to extend the compiler or even trying to understand any other part of the source tree.

- **`HD.HDFun`**. Most of the ForSyDe process constructors, such as *mapSY* (see chapter 1) are represented in terms of a higher order function. That is, a function is passed along with the signal in order process it.

  The syntactical information of those functions needs to be internally stored (bundled with an `HDSignal`) for latter translation to the target language. The `HD.HDFun` module makes use of Template Haskell to make it possible: It provides a data structure (`HDFun`) which stores all the function information and a constructor (`mkHDFun`) to create such structure from the Template-Haskell representation (`Q[Dec]`) of a function. Colaterally, `HD.HDFun` is responsible of establishing the Haskell syntax-subset supported by `HDFun`s.

  Familiarity with Template Haskell is obviously a must in order to deal with this module.

A.2.2 Miscellaneous auxiliary functions and types

`HD.Misc` contains miscellaneous helper functions and types which are too small to be splitted in their own modules. The only important part to highlight is the `EProne a` type, which holds an *error-prone* value and uses `mtl`'s `Error` monad class to handle those errors.

Error management within the compiler is handled, to a large extend, through the `Eprone` type.

A.2.3 Ports, Blocks and Block Instances

The concepts of a *Port*,*Block* and *Block Instance* were introduced in chapter 3. The `HD.Port` module implements them and enables their use, providing as well an API to the end-user to deal with them.

The introduction of Ports has implied as well the introduction of typecheking (there is no way to statically assure whether a the type of a port entry matches a signal attached to it).

Moreover, due to the fact that the compiler is Embedded in Haskell, compilation of the guest language (i.e. ForSyDe models) strangely takes place at runtime, and unavoidably so does typechecking.

As a result, the functions which attach or obtain signals to/from a port (namely `plugSig` and `supplySig`) are in charge of doing typechecking.

Moreover, due to the restrictions imposed by the host language, typechecking turns to be a di cult task. A tricky solution using dynamic signals (that is, `forall`[3]`a.HDPrimType a => HDSignal a`) was gured out. It works nicely in practice, but it certainly could (and probably should) be redesigned or improved.

The coexistence of static and dynamic signals requires a way of conversion between them. That is the purpose of the `sigCast` function.

A.2.4 Backends

A backend is in charge of interpreting the intermediate representation of the source code. The most common form of interpretation is a translation into another language (e.g. `VHDL`) but some other possibilities are: simulation, design veri cation, generation of a graphical representation etc ....

In our case, a backend uses a `Block` as input, whose netlist (the internal circuit graph) is traversed to achieve the aforementioned interpretation.

The module `HD.NetList` (which was imported and adapted from Lava) provides the developer with `netlist`, a function which carries out the traversal of the circuit. Due to the use of *Observable Sharing* in the circuit representation, which is right now is restricted to `IO`-references (see module `HD.Ref`) `netlist` is subdued to the `IO` Monad. Furthermore, in order to allow reading and modifying state data during the traversal, the `IO` Monad is wrapped into `mtl`'s `StateT` monad transformer.

There could be interpretations, such as simulation, for which a `ST` version of `netlist` would be preferable to current `IO` version. Adding an alternative

[3]If you are confused with the `forall` keyword, that means you should get familiar with existential types before trying to go through this module.

`ST` version would involve including `ST`-references in `HD.Ref` (maybe from the Lava-project, which already o ers them).

Currently, the only available backend is a translator to VHDL, which is divided in three modules

- **`HD.VHDL`**. It is in charge of the VHDL translation, which is accessible through the `writeVHDL` function.
- **`HD.VHDL.AST`** de nes data structures to hold the AST (*Abstract Syntax Tree*) of the generated VHDL modules.

  It is based in the grammar of the VHDL93 standard but only supports the subset of VHDL's syntax which has been needed. However, due to being based in the standard it could be easily extended in the future if desired.
- **`HD.VHDL.Ppr`**. A `pretty-printer` library for the VHDL AST-structures mentioned above. It is in charge of the transformation and output of the AST into readable VHDL text modules.

A.2.5 Integration with ForSyDe

The development of the compiler did not entail big changes in the ForSyDe library. A new alternative signal type (`HDSignal`) was developed to avoid modifying ForSyDe's original stream-based `Signal` type.

However, one of the main compiler design-requirements was to be able to reuse the original name and semantics of ForSyDe's signal-processing functions with the new signal type. It is a certainly nice feature, since it allows all earlier documentation to still apply for both signal types. In order to make it possible, ForSyDe's original `SynchronousLib` module had to be modi ed. Its `mapSY`, `zipWithSY` and `delaySY` functions (the only ForSyDe processes currently supported natively by the compiler) were transformed into type classes[4] and instances were created for both `Signal` and `HDSignal`.

The new type classes and `Signal` instances remain in `SynchronousLib` while the `HDSignal` ones were included in a new module: `HDSynchronousLib`.

It is worth to note that, even if the compiler only currently supports *mapSY*, *zipWithSY* and *delaySY* natively as primitives[5], due to the type class scheme described above, all their derived process constructors (e.g. *sourceSY*),

[4]Due to the nature of the the signal processing functions, the type classes required various parameters, and thus *Multiparameter Type Classes* (a common Haskell extension) where used for that purpose.

[5]It can be said that they constitute the primitive processes of the compiler.

are automatically supported. However they are treated as compound processes and thus, there is currently no way to internally identify them as individual entities. Treating those derived processes as primitives would be desirable for a more precise control of the way in which the design is interpreted, allowing optimizations and improving code generation in the backends.

A.3 Recipes to extend the compiler

The following sections are targeted at suggesting the steps to follow when developing common extensions for the compiler.

The main purpose of these recipes is to help new developers, only interested in certain extensions, to get familiar with the compiler. Depending on the extension, some other changes might be required. As a result, it is encouraged to take the recipes with a grain of salt and expect further modi cations to be needed in the process.

A.3.1 Adding support for new types of signal values

`Bool` and `Int` `HDSignal`s might be enough for many designs, but it can be a good idea to support other signal values such as tuples, enumerates etc ...

To add a new type, say `a`

1) Add a constructor for `a` in `HD.HDTypes.HDType`

2) Make `a` an instance of `HD.HDTypes.HDPrimType`

3) Add a constructor for `a` in `HD.HDTypes.HDPrimConst`

4) Adjust the backends to handle the new type. In the VHDL backend that roughly means changing `HD.VHDL.hDST2TM` and `HD.VHDL.hDPC2Expr`.

5) The new type might require extending the Haskell subset supported by `HDFuns`. For instance, tuples would be di cult to use without pattern matching. Read on for details about how to perform this extension.

6) Consider adding support for new functions within an `HDFun`. As an example, in the case of choosing to add support for tuples, it would be desirable to get support for well-known `fst` and `snd` functions. Details about how to do this are covered in a following section.

A.3.2 Treating more ForSyDe processes as primitives

As it was previously stated, the only process constructors treated natively as primitives are *mapSY*, *zipWithSY* and *delaySY*. *sourceSY*, for example, is handled as a compound of *mapSY* and *delaySY*.

In order to treat a process, say *sourceSY* from ForSyDe's original `SynchronousLib`, as a primitive:

1) Add `sourceSY` to `HD.HDSignal.HDSignal`

2) Transform `SynchronousLib.sourceSY` into a type class and use its previous de nition to make `Signal` an instance of it.

3) Modify `HD.HDSynchronousLib` by making `HDSignal` an instance of the new `SourceSY` class.

4) Adjust the backends to handle the new primitive. Basically, that entails changing the `new` and `define` functions passed to `netlist`, see `HD.VHDL` for more details.

A.3.3 Extending `HDFun`'s Haskell-syntax subset

The Haskell syntax admited by `HDFuns` (more precisely by `HD.HDFun.mkHDFun`) is quite limited. For example, neither constructor pattern-matching nor *where-clauses* are allowed (chapter 3 gives a precise description of the supported subset).

To broaden the supported Haskell-subset.

1) Locate what type in the the Template Haskell AST (`Language.Haskell.TH.Syntax`) holds the desirable syntax extension.

2) Modify the corresponding `Lift` instantiate for that type in `HD.HDFun` in order to give support to the new feature.

3) Adapt the compiler backends to the change.

A.3.4 Adding support for new functions within `HDFuns`

The functions which a designer can use within a `HDFun` (e.g. `(&&)`, `(==)`, `(-)`, `(+)` ...) are limited. That limitation, however, is not intrinsic to `HDFuns` themselves, which can indeed make use of any valid Haskell function as long as it is done in a correct manner. The problem is that those functions have to be interpreted at some point by the translation backends.

Thus, in order to support new functions within an `HDFun`, the backend has to be aware of them and know how to handle them. For that reason, to add in a new function simply ...

- Adapt the backends to support the new function.

The way to dispatch those functions depends highly on the translation backend, for example, in the `VHDL` backend, translation tables are kept to help mapping them from Haskell to VHDL.

A simulation backend on the other hand does not involve a translation and could make use of the original Haskell function (which is stored in the `HDFun` algebraic type). That makes possible to forget about the `HDFun` AST and translation tables.

A.3.5 Adding a new backend

To add a new backend such as simulation, translation to another target language or code veri cation, the intermediate representation the ForSyDe model has to be traversed and processed. A good way to start up would be looking at the sources of current `VHDL` backend to understand how the `netlist` function works in practice.

A simulation backend could be coded as follows.

1) Verify whether the `HD.NetList.netlist` function ful ls the needs of the backend and otherwise create an alternative version.

   Simulation could serve as a means of translating between `HDSignal`s and `Signal`s but unfortunately `netlist` depends on `IO` from which no one can safely escape (transforming an `IO a` value into `a` causes side e ects). Thus, a `ST-netlist`[6] version would be necessary to bypass `IO`. See section A.2.4 for more details.

2) Check if the intermediate representation of the circuit (contained in `HD.-HDSignal.HDSignal`) needs to be extended.

   A simulation backend would need to make use of the original Haskell function stored in `HDFun`s to avoid having to traverse its AST. However, the function itself is not stored in an `HDSignal`

3) Lastly, code the backend.

[6]The `a` value of `ST a` can be extracted through `runST`

A.4 Improving the source code

Every piece of software is an evolving entity which can always be improved. The compiler produced during this thesis is no exception and is far for being perfect in many ways.

Due to the time-limitation of the thesis, I was not able to make the code be organized and work as good I would have liked to, but at least I was careful enough to anotate and coment all the potential improvements I could notice. I used the traditional `FIXME` and `TODO` tags for that purpose.

Reading the output produced by `grep -ri 'TODO\|FIXME' *` in the `HD` directory gives a good hint about where to start working.

Appendix B

Examples

B.1 *plus1*

B.1.1 `Plus1.hs`

ForSyDe speci cation model of a simple adder system.

```haskell
{-# OPTIONS_GHC -fth #-} -- The use of -fth is due to splicing mkHDFun
module Plus1 where
-- A simple addaing system

import HD
import SynchronousLib (mapSY)

-- hdPlus1 input port
plus1In :: InPort
plus1In = mkInPort 1 [("plus1Input", Int)]

-- hdPlus1 output port
plus1Out :: OutPort
plus1Out = mkOutPort 1 [("plus1Output", Int)]


hdPlus1 :: HDSignal Int -> HDSignal Int
hdPlus1 = mapSY doPlus1
 where doPlus1 = $(mkHDFun [d| doPlus1 :: Int -> Int
                               doPlus1 a = a + 1         |])


plus1Circ :: InPort -> OutPort
plus1Circ ip = supplySig  plus1Sig "plus1Output" plus1Out
  where plus1Sig = plugSig "plus1Input" ip hdPlus1

plus1Block :: Block
plus1Block = mkBlock "plus1_block" plus1In plus1Circ
```

B.1.2 plus1.vhd

VHDL code generated from its Haskell model. Note that the code depends on ForSyDe's VHDL Library, which is included in appendix C.

```
-- Generated with ForSyDe
library forsyde;
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;


entity plus1 is
     port (plus1Input : in forsyde.types.int32;
           plus1Output : out forsyde.types.int32;
           clk : in std_logic;
           resetn : in std_logic);
end entity plus1;


architecture Structural of plus1  is
     signal mapSY_0 : forsyde.types.int32;
begin
     plus1Output <= mapSY_0;

     mapSY_0_block : block
          port (mapSY_input : in forsyde.types.int32;
                mapSY_output : out forsyde.types.int32);
          port map (mapSY_input => plus1Input,
                    mapSY_output => mapSY_0);
          function doPlus1 (a : forsyde.types.int32)
                            return forsyde.types.int32 is
          begin
               return a + TO_SIGNED (1, 32);
          end;
     begin
          mapSY_output <= doPlus1 (a => mapSY_input);
     end block mapSY_0_block;
end architecture Structural;
```

B.1.3 RTL model of *plus1*

RTL model obtained with Altera's *Quartus II* software after compiling the VHDL model for a Stratix FPGA.

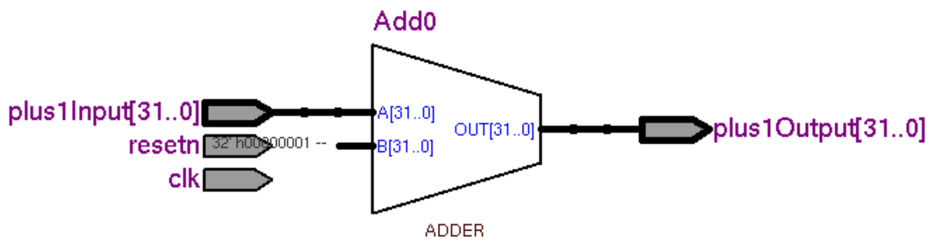


Figure B.1: RTL model of *plus1*

Appendix C

ForSyDe's VHDL Library

C.1 `forsyde.vhd`

The following listing conatins the, still very simple, source code of `forsyde.vhd`, a VHDL library used by the target VHDL models.

```
library ieee;
use ieee.numeric_std.all;

package types is
  subtype int32 is signed(31 downto 0);  -- 32 bit integers
end types;
```

Bibliography

[1] Ingo Sander. *System Modeling and Design Refinement in ForSyDe.* PhD thesis, Royal Institute of Technology, Sweden, 2003.

[2] ANSI/IEEE. *IEEE standard VHDL language reference manual.*, 1993. Std 1076-1993.

[3] *Haskell 98 Language and Libraries: the revised report.* Cambridge University Press, 2003.

[4] Paul Hudak, John Peterson, and Joseph Fasel. A gentle introduction to Haskell 98. `http://www.haskell.org/tutorial/`, 1999.

[5] Simon Thompson. *Haskell: The Craft of Functional Programming.* Addison Wesley, July 1996.

[6] Don Stewart Bryan O'Sullivan and John Goerzen. *Real-World Haskell.* O'Reilly, Upcoming publication. `http://www.realworldhaskell.org`.

[7] Ingo Sander. The ForSyDe methodology. In *Swedish System-on-Chip Conference*, 2002.

[8] M. Sheeran. Hardware design and functional programming: a perfect match. *Journal of Universal Computer Science*, 11(7):1135 1158, 2005.

[9] Tom Hawkins. Declarative programming language simpli es hardware design, September 2003. `EEdesign.com`, Design news.

[10] Mary Sheeran Per Bjesse, Koen Claessen and Satnam Singh. Lava: Hardware design in haskell. In *ICFP '98*, 1998.

[11] Koen Claessen. An embedded language approach to hardware description and veri cation, August 2000. Dept. of Computer Science and Engineering, Chalmers University of Technology. Lic. thesis.

[12] Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of haskell programs. In *International Conference on Functional Programming*, pages 268 279. ACM, 2000.

[13] R. R. Collins. The pentium f00f bug. *Doctor Dobbs Journal*, 23(5):62 69, 1998.

[14] V. Pratt. Anatomy of the Pentium Bug. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, *TAPSOFT'95: Theory and Practice of Software Development*, number 915, pages 97 107. Springer Verlag, 1995.

[15] Carl-Johan H. Seger, Robert B. Jones, John W. O'Leary, Tom Melham, Mark D. Aagaard, Clark Barrett, and Don Syme. An industrially e ective environment for formal hardware veri cation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(9):1381 1405, September 2005.

[16] Jim Grundy, Tom Melham, and John O'Leary. A reflective functional language for hardware design and theorem proving. *J. Funct. Program.*, 16(2):157 196, 2006.

[17] Alexander Aiken, John H. Williams, and Wimmers. The FL project: The design of a functional language.

[18] John Launchbury, Je rey R. Lewis, and Byron Cook. On embedding a microarchitectural design language within Haskell. In *ICFP '99: Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, pages 60 69, New York, NY, USA, 1999. ACM Press.

[19] K. Claessen and G. Pace. An embedded language framework for hardware compilation. In *DCC'02, ETAPS, Grenoble, France.*, 2002.

[20] Paul Hudak. Building domain-speci c embedded languages. *ACM Computing Surveys*, 28(4es):196 196, 1996.

[21] Daan Le en and Erik Me er. Domain speci c embedded compilers. In *2nd USENIX Conference on Domain Specific Languages (DSL'99)*, pages 109 122, Austin, Texas, October 1999. Also appeared in ACM SIGPLAN Notices 35, 1, (Jan. 2000).

[22] Paul Hudak, Tom Makucevich, Syam Gadde, and Bo Whong. Haskore music notation - an algebra of music. *Journal of Functional Programming*, 6(3):465 483, 1996.

[23] Conal Elliott, Sigbjørn Finne, and Oege de Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(2), 2003. Updated version of paper by the same name that appeared in SAIG '00 proceedings.

[24] Axel Jantsch. Models of embedded computation. In Richard Zurawski, editor, *Embedded Systems Handbook*, chapter 4. CRC Press, 2005. Invited contribution.

[25] S. Singh and M. Sheeran. Designing FPGA circuits in lava.

[26] Koen Claessen, Mary Sheeran, and Satnam Singh. The design and veri cation of a sorter core. *Lecture Notes in Computer Science*, 2144:355+, 2001.

[27] Joseph Cordina and Gordon J. Pace. Functional HDLs: A Historical Overview. In *Computer Science Annual Workshop 2006 (CSAW'06)*. University of Malta, December 2006.

[28] Steven D. Johnson. *Synthesis of Digital Design from Recursive Equations*. MIT Press, Cambridge, MA, USA, 1984.

[29] J. O'Donnell. Generating netlists from executable circuit speci cations in a pure functional language, 1992.

[30] J. O'Donnell. Hardware description with recursion equations, 1987.

[31] Koen Claessen and David Sands. Observable sharing for functional circuit description. In *Asian Computing Science Conference*, pages 62 73, 1999.

[32] N.M. Mendes Alves and S. de Mello Schneider. Implementation of an embedded hardware description language using haskell. *Journal of Universal Computer Science*, 9(8):795 812, 2003.

[33] John T. O'Donnell. Embedding a hardware description language in template haskell. In *Domain-Specific Program Generation*, pages 143 164, 2003.

[34] S. Klinger. The Haskell programmer's guide to the IO monad don't panic. Technical Report TR-CTIT-05-54, Enschede, November 2005.

[35] John O'Donnell. Overview of Hydra: A concurrent language for synchronous digital circuit design. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 240, Washington, DC, USA, 2002. IEEE Computer Society.

[36] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, (37):67 111, May 2000.

[37] Ross Paterson. A new notation for arrows. In *International Conference on Functional Programming*, pages 229 240. ACM Press, September 2001.

[38] John Hughes. Programming with arrows. In *Advanced Functional Programming*, pages 73 129, 2004.

[39] T. Sheard. Another look at hardware design languages: freeing ourselves from the tyranny of the host language, 2005.

[40] Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In Manuel M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1 16. ACM Press, October 2002.

[41] K. Claessen and M. Sheeran. A tutorial on Lava: A hardware description and veri cation system, 2000.

[42] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 96 107, New York, NY, USA, 2004. ACM Press.

[43] Mark P. Jones. Type classes with functional dependencies. In *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 230 244, London, UK, 2000. Springer-Verlag.

[44] Andrew K. Martin and Ahmed Gheith. A framework for designing hardware in Ocaml. In *Hardware Design and Functional Languages (HFL'07), Braga, Portugal*, March 2007.

[45] The GHC Team. The glorious glasgow haskell compilation system user's guide, version 6.6. `http://www.haskell.org/ghc/docs/6.6/html/users_guide/`.