KUNGL
TEKNISKA
HÖGSKOLAN

System Modeling and Design Refinement in ForSyDe

Ingo Sander

Stockholm 2003

Thesis submitted to the Royal Institute of Technology in partial fulfillment of the requirements for the degree of Doctor of Technology

Sander, Ingo
System Modeling and Design Refinement in ForSyDe

ISBN 91-7283-501-X
TRITA-IMIT-LECS AVH 03:03
ISSN 1651-4076
ISRN KTH/IMIT/LECS/AVH-03/03--SE

Royal Institute of Technology
Department of Microelectronics and Information Technology
Laboratory of Electronics and Computer Systems
Electrum 229
S-164 40 Kista, Sweden

Abstract

Advances in microelectronics allow the integration of more and more functionality on a single chip. Emerging system-on-a-chip architectures include a large amount of heterogeneous components and are of increasing complexity. Applications using these architectures require many low-level details in order to yield an efficient implementation. On the other hand constant time-to-market pressure on electronic systems demands a short design process that allows to model a system at a high abstraction level, not taking low-level implementation details into account. Clearly there is a significant abstraction gap between an ideal model for specification and another one for implementation. This abstraction gap has to be addressed by methodologies for electronic system design.

This thesis presents the ForSyDe (Formal 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. ForSyDe is based on carefully selected formal foundations. The initial *specification model* uses a *synchronous model of computation*, which separates communication from computation and has an abstract notion of time. ForSyDe uses the concept of *process constructors* to implement the synchronous model, to allow for design transformation and the mapping of a refined model onto the target architecture. The specification model is refined into a detailed *implementation model* by the stepwise application of well-defined *design transformation rules*. These rules are either semantic preserving or they inflict a design decision modifying the semantics. These design decisions are used to introduce the low-level implementation details that are needed for an efficient implementation. The implementation model is mapped onto the components of the target architecture. At present ForSyDe models can be mapped onto VHDL or C/C++ in order to allow commercial tools to generate custom hardware or sequential software. The thesis uses a digital equalizer to illustrate the concepts and potential of ForSyDe.

Acknowledgements

The research was funded by the Royal Institute of Technology, Vinnova and the Swedish Foundation for Strategic Research.

There are many people who in one form or another contributed to the work described in this thesis. I want to take the opportunity to say a big ”Thank you” to all of you.

In particular I would like to thank my formal and informal supervisors over the years. The biggest thank goes to my supervisor, Professor Axel Jantsch, who not only contributed to my work in various ways, but also always encouraged me and took his time for discussions on my research. Thanks a lot for a fantastic supervision! I also want to thank Professor Hannu Tenhunen, who in all years has put an enormous effort on the development of research and education in electronic system design at our department. I also want to thank Professor Ahmed Hemani, who a long time ago accepted me as PhD student and introduced me into research.

I also want to thank all past and current PhD students from ESD/LECS that I have been working together with. It is not possible to mention all of you by name. But I want in particular thank Andreas Jonasson who convinced me that functional languages are fantastic; Wenbiao, Zhonghai, Tarvo and Ashish for their work on ForSyDe; Per, for many discussions, which strangely have never resulted in a joint paper; Abhijit from whom I have learned that subways are dangerous; Johnny, from whom I have learned that almost everything in Sweden is fantastic; Patrik and all others who organized and participated in the soccer and innebandy matches at ESD Lab; Dilian and Dinesh for our chess games and discussions.

A special thanks goes to Hans and all other people from the system group of our department who work day and night to give us a great working environment. And also thanks a lot to Lena for all administrative help.

Also I want to send many thanks to my old department at Campus Haninge. First of all I want to thank Lars Källander and Inge Jovik who gave me the opportunity to combine my teaching with PhD studies. And a special to Eva-Lotta for all discussions about research in general. And also big thanks to all others, who have

not directly contributed to the thesis, but made working more fun.

Thanks to my mother and father who put me on the right track from the beginning. And the biggest thanks of all goes to my wife, Sofi, and to my children, Maria, Ulrika and Tobias. I am such a lucky person to have you!

Stockholm, April 2003
Ingo Sander

Contents

List of Figures

List of Abbreviations

A/D	Analog-to-Digital Converter
ATM	Asynchronous Transfer Mode
CCS	Calculus of Communicating Systems
CFSM	Co-Design Finite State Machine
CIP	Computer-Aided, Intuition-Guided Programming
CSP	Communicating Sequential Processes
D/A	Digital-to-Analog Converter
DFT	Discrete Fourier Transform
DSP	Digital Signal Processor
FFT	Fast Fourier Transform
FIFO	First-In-First-Out (Buffer)
FIR	Finite Impulse Response
ForSyDe	Formal System Design
FPGA	Field Programmable Gate Array
FRP	Functional Reactive Programming
FSM	Finite State Machine
GALS	Globally Asynchronous - Locally Synchronous
HML	Hardware ML (Meta Language)
HW	Hardware
ITRS	International Technology Roadmap for Semiconductors
MASCOT	Matlab and SDL Codesign Techniques
Matlab	Matrix Laboratory
MESCAL	Modern Embedded Systems, Compilers Architectures and Languages
MIMD	Multiple Instruction, Multiple Data
ML	Meta Language
MoC	Model of Computation

OSI	Open Systems Interconnect
RTL	Register Transfer Level
SAFL	Statically Allocated Functional Language
SDF	Synchronous Data Flow
SDL	Specification and Description Language
SIMD	Single Instruction, Single Data
SoC	System on a Chip
SPI	System Property Intervals
SW	Software
VCC	Virtual Component Co-Design
VHDL	Very High Speed Integrated Circuit Hardware Description Language

Chapter 1

Introduction

This chapter presents the motivation for the research on ForSyDe and defines the objectives of this thesis. It states the author's contributions to the ForSyDe methodology and gives an overview of the structure of the thesis.

1.1 Motivation

1.1.1 Embedded Systems

Embedded systems play an increasingly important role in daily life. An embedded system can be defined as a system that uses a computer to perform a specific function, which is neither used nor perceived as a computer [27]. Embedded systems can be found in different application areas, such as train, car and aircraft controllers, consumer electronics, communication systems or medical devices. These embedded systems can include more and more functionality on a single chip due to Moore's law, which predicts that the number of transistors on a given chip can be doubled every 18 months. So far Moore's law, which is based on an observation that was formulated in 1965 [85], has been proven correct for almost 40 years[1]. It works now almost as a requirement specification for the semiconductor industry, which use the International Technology Roadmap for Semiconductors (ITRS) to formulate the challenges implicitly given by Moore's law [4]. It seems that Moore's law will hold at least through the end of this decade, which means that in

[1] Moore had observed that the number of transistors per square inch has been doubled every year since the invention of the integrated circuit. This trend had been slowed down to 18 months for the doubling of transistors, but has since then been stable for many years.

future even more complex functionality can be implemented on a single chip and that new applications using these technology advances will appear.

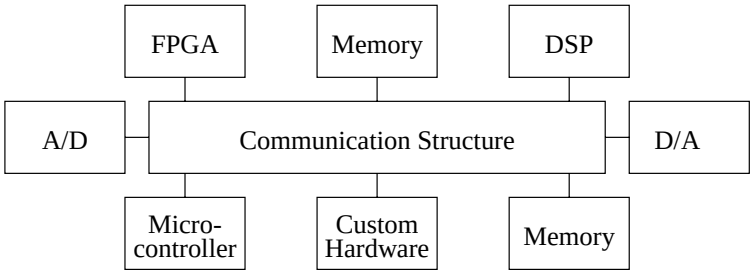


Figure 1.1. A possible system-on-a-chip architecture

Already today it is possible to implement a system on a single chip. As illustrated in Figure 1.1 such a system on a chip (SoC) integrates micro-controllers, digital signal processors (DSPs), memories, custom hardware, and reconfigurable hardware in the form of field programmable gate arrays (FPGAs) together with a communication structure and analog-to-digital (A/D) and digital-to-analog (D/A) converters on a single chip. These architectures offer an enormous potential. However, on the other hand, they are extremely complex and heterogeneous. This does not only apply for the hardware, but also for the software. The communications between all the entities is extremely complex, since these entities are distributed and often share communication channels and memory resources.

Most embedded systems are *reactive systems* [12]. Reactive systems, e.g. a cruise control in a car, react continuously to their environment at the speed of the environment. In contrast *interactive systems*, e.g. an Internet application, interact with the environment at their own speed, while *transformational systems*, e.g. a compiler, transform input data into output data. Very often reactive systems are safety critical. In case of a train control system the train must stop immediately if an emergency condition occurs. The design process must be able to give these guarantees, since there is a risk for human life.

1.1.2 Embedded System Design

The design process of future embedded systems must be able to implement an application with guaranteed properties, as needed by for instance safety critical systems, but has also to cope with the complexity and heterogeneity of the underlying architecture. The present trend in industrial design is that the part of the

verification costs related to the total design costs is continuously increasing. Current industrial designs rely heavily on simulation techniques, which of course will still be important in future, but there is a need for a complement in form of formal verification methods.

Since it in practice is very difficult to capture the functionality of the system and to verify system level properties on the level of architectural components, system design must start on a much higher abstraction level. Unfortunately, the higher the level of abstraction the more implementation details are missing, which are needed to obtain an efficient implementation. The design process must add these implementation details.

System design starts with the development of a *specification model*. In this phase the designer formulates a model according to the requirements given in a *requirement specification*, which usually is written in a natural language, e.g. in English. It is important that the specification model is expressed in a formal language. A formal language has a formal syntax[2] and a formal semantics[3] that allows for formal manipulation and tool support, which can for instance be used to detect inconsistencies, ambiguities or incompleteness in a specification model.

The higher the abstraction level of the specification model, the fewer implementation details are inherent in the model and the larger is the *design space*. The design space is defined as the amount of possible implementations that fulfill a given specification model and is illustrated in Figure 1.2.

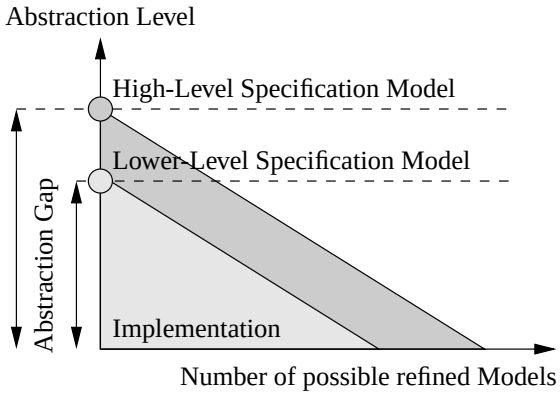


Figure 1.2. A high abstraction level leaves a wider design space

[2] A syntax of a language defines how language symbols are put together to form valid language expressions.

[3] A semantics of a language gives a meaning to language expressions.

Also a high abstraction level usually means that the system can be described in a cleaner and simpler way, since implementation details do not have to be taken into account. Simple models help the designer to understand and formulate the functionality of the system, since there is less distraction from unwanted details. In addition a simpler model also allows for more efficient system model *verification*[4], where it is confirmed that the specification model fulfills all the functional requirements that have been imposed on the system.

On the other hand a high abstraction level of the system model aggravates the *synthesis process*[5], since a model with very few implementation details has to be translated into an implementation on a complex and heterogeneous architecture where a large amount of implementation details are needed. This leads to a large *abstraction gap* (Figure 1.2), which has to be bridged during synthesis.

Thus a system design methodology has to offer the following:

- the possibility to model the system at a high level of abstraction;
- a synthesis process that allows the bridging of the abstraction gap in order to yield an efficient implementation.

These objectives can be summarized as the challenge for a successful system design methodology.

In order to manage the complexity and heterogeneity of SoC applications Edwards et al. [27] believe that the design approach should be based on the use of one or more formal methods to describe the behavior of the system at a high level of abstraction, before a decision on its decomposition into hardware and software is taken. The final implementation of the system should be made by using automatic synthesis from this high level of abstraction to ensure implementations that are ”correct by construction”. Validation through simulation or verification should be done at the higher levels of abstraction.

They advocate a design process that is based on representations with precise mathematical meaning, so that both the verification and the mapping from the initial description to the various intermediate steps can be carried out with tools of guaranteed performance. A formal model of a design should consist of the following components:

1. a *functional specification* given as a set of explicit or implicit relations, which involve inputs, outputs, and possibly internal (state) information;

[4]The term verification is used in this thesis for both system simulation and formal verification.

[5]The term *synthesis* is often used for the *automatic* refinement of a model into a less abstract model due to the addition of details. Here the term is even used for the *manual* refinement of a specification model into an implementation on a given system architecture.

2. a *set of properties* that the design has to satisfy;

3. a *set of performance indexes* that evaluate the design in different aspects (speed, size, reliability, etc.);

4. a *set of constraints* on performance indexes.

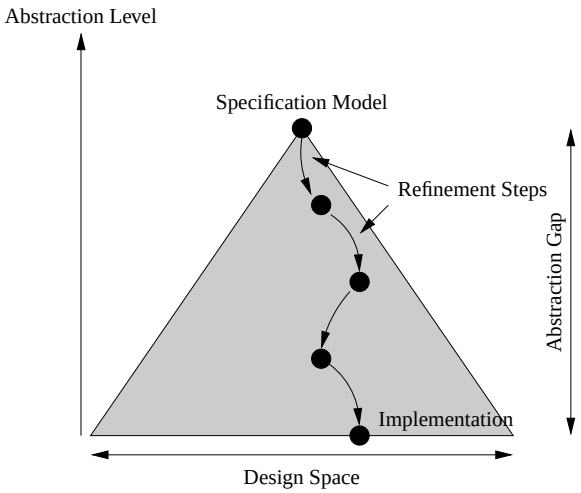


Figure 1.3. The synthesis process is a stepwise refinement from a high-level specification model into a final implementation

The design process takes a model of the design at one level of abstraction and refines it to a lower one. The refinement process involves also the mapping of constraints, performance indexes, and properties to the lower level. Such a design process is illustrated in Figure 1.3.

The view of Edwards et al. should be understood as a goal for system design. In particular a ”correct-by-construction” design process is difficult to achieve. However, a similar view on system design has been formulated by Keutzer et al. [65], who also point out that the orthogonalization of concerns, in particular the separation between (1) computation and communication and (2) function and architecture, is of crucial importance.

1.1.3 The ForSyDe Approach to Embedded System Design

This thesis presents the ForSyDe approach to embedded system design. ForSyDe (Formal System Design) is a design methodology for embedded systems and based

on a similar view on system design as formulated in [27] and [65]. In particular, the starting point for the development of the ForSyDe methodology was the conviction that future system design methodologies have to

- start from a high-level of abstraction in order to be able to cope with the extreme complexity of future applications and architectures;
- give a solid base for the future incorporation of formal methods, since simulation alone is not sufficient to verify future systems at different levels of abstraction.

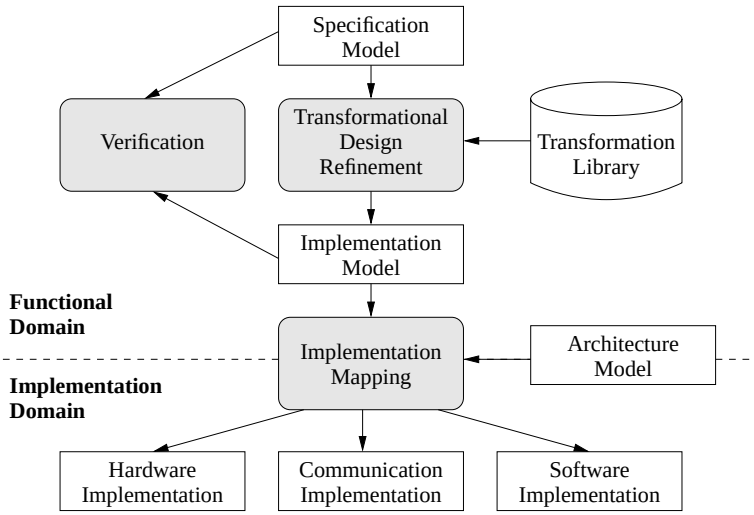


Figure 1.4. The ForSyDe design flow

Thus the ForSyDe design process, which is illustrated in Figure 1.4, starts with the development of a formal and functional *specification model* at a high abstraction level. The model is formal since it has a formal syntax and semantics. It is abstract and functional since a system is modeled as a mathematical function of the input signals. The specification model uses a synchronous model of computation[6], which is based on a clean and simple mathematical formalism and on an abstract communication model. This formal base of ForSyDe gives a good foundation for the integration of formal methods.

The synthesis process is divided into two phases. First the specification model is refined into a more detailed implementation model by the stepwise *application*

[6]Models of computations are discussed in Section 2.1.

of design transformations. Since the semantics of the specification model is a subset of the semantics of the implementation model the same verification techniques[7] can be applied to both models. Design transformation is conducted in the *functional domain*. Inside the functional domain a system model is expressed as a function using the semantics of ForSyDe. The second step in the synthesis phase is the mapping of the implementation model onto a given architecture. This phase comprises activities like partitioning, allocation of resources and code generation. In the implementation mapping phase, the design process leaves the functional domain and enters the *implementation domain*, where the design is described with ”implementation-level languages”, i.e. languages that efficiently express the details of the target architecture, such as synthesizable VHDL for hardware or C for software running on a micro-controller.

The task of the refinement process is to optimize the specification model and to add the necessary implementation details in order to allow for an efficient mapping of the implementation model onto the chosen architecture.

So far ForSyDe is restricted to a small number of transformations (Chapter 5) and mapping rules for hardware and software processes (Chapter 6), which are far from sufficient to target industrial designs, but allow to indicate the potential of ForSyDe.

1.2 Thesis Objectives

The main objective of this thesis is to show the perspectives ForSyDe offers for embedded system design. ForSyDe’s objective is to move system design to a higher level of abstraction and to bridge the abstraction gap by transformational design refinement. The thesis gives a motivation for the basic concepts of ForSyDe, in particular the choice of the synchronous computational model for the specification model, the concept of process constructors, and the transformational design refinement in ForSyDe. The concepts are illustrated by the example of a digital equalizer, which has been modeled, refined and mapped onto an implementation in hardware. The thesis also discusses the potential, limitations and future directions of ForSyDe.

A second objective is to make the thesis a comprehensive documentation of ForSyDe, since so far all information exists distributed in several conference papers.

[7] To date simulation is used for the verification of ForSyDe models, but due to the formal character of ForSyDe there is a good base to incorporate formal methods.

1.3 Author's Contribution

Although the first work on ForSyDe was published not earlier than in 1999, preliminary work on ForSyDe has started about three years earlier, initiated by discussions with another PhD student, Andreas Jonasson. He was inspired by the work of Reekie [92], who used the concept of higher-order functions inside the functional language Haskell to model digital signal processing networks. Unfortunately, Andreas Jonasson left our university shortly after these discussions to start an industrial career. About the same time, in 1997, Axel Jantsch joined our university and has since then participated as informal and formal supervisor in the development of ForSyDe.

The ForSyDe methodology has mainly been developed by the author of the thesis. Initially Andreas Jonasson contributed with many ideas before he joined industry. Wu Wenbiao worked together with the author of the thesis on design transformation [115] for about one and a half years. Recently Zhonghai Lu, Tarvo Raudvere and Ashish Kumar Singh joined the ForSyDe group. Zhonghai Lu's main contribution is a mapping technique for sequential software [73]. Axel Jantsch's main contribution is the work on stochastic processes in ForSyDe [59] and extremely useful feedback on all ForSyDe concepts.

Though the thesis shall be regarded as a comprehensive document on ForSyDe, it does only cover the parts of ForSyDe where the author had an essential contribution. Thus the work on stochastic processes, where the main contributor is Axel Jantsch, and the work on a mapping procedure to sequential software, where the main contributor is Zhonghai Lu, is only mentioned and not given in detail in this thesis. Also each part of the appendix is original work from the author. A large part of the work on this thesis was to integrate the work in earlier papers into a single consistent document with one consistent notation.

The following sub-sections give the contribution of the author to the papers included in this thesis and other papers. All papers given here are peer-reviewed.

1.3.1 Publications included in this Thesis

Modeling The *modeling concepts* of ForSyDe[8] have been introduced in [94]. The paper presents the use of a *synchronous computational model* and the concept of *process constructors*[9]. It exemplifies the modeling technique by a

[8] The name of the methodology, ForSyDe, was introduced in a later paper [115].

[9] Process constructors have been denoted skeletons in earlier papers inspired by the work of Skillicorn [104].

case study of an ATM[10] switch with operation and maintenance (OAM) functionality. The work in this paper was conducted by the author of the thesis and supported by feedback from Axel Jantsch.

Implementation Mapping The papers [95] [96] focus on *hardware synthesis*. They illustrate how a ForSyDe model can be mapped into a hardware description in VHDL. Finally a FIFO-buffer in the ATM switch was manually synthesized into a netlist of standard cell gates. The work in these papers was done by the author of the thesis and supported by feedback from Axel Jantsch.

A case study of hardware and software synthesis using the equalizer model was presented in [73]. The paper suggests a synthesis technique for hardware and software. The basic concepts of this technique are based on earlier work on hardware synthesis [95] [96]. Zhonghai Lu has developed a synthesis technique for sequential software and manually synthesized the equalizer model into behavioral VHDL and sequential C. He was assisted by feedback from the author of the thesis and Axel Jantsch.

Design Refinement *Transformational design refinement* was introduced into ForSyDe in [115]. The paper gave examples for simple *semantic preserving* and *design decision transformations* using the ATM switch model, but did not fully illustrate the potential of transformational refinement. The paper was a joint work of Wu Wenbiao and the author of the thesis, with important feedback from Axel Jantsch.

The paper [97] illustrated the potential of transformational design refinement by *clock domain refinement* and *communication refinement*. The paper introduces the concept of *domain interfaces* and *synchronous sub-domains*, which allows it to establish multiple clock domains by design transformations. The transformations were exemplified by a digital equalizer. The work in this paper was conducted by the author of the thesis and supported by feedback of Axel Jantsch.

The formal format of a transformation rule together with the concept of the *characteristic function*, which allows it to describe the implications of a transformation rule, was presented in [99]. The work in this paper was mainly conducted by the author of the thesis, supported by feedback from Axel Jantsch, and the synthesis of the FIR-filter example by Zhonghai Lu.

[10] Asynchronous Transfer Mode

ForSyDe Methodology The article [98] has been submitted to the journal *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* and is under review at the time of writing this thesis. It describes the ForSyDe methodology in a more concise way than this thesis. The work in the article was conducted by the author of the thesis and supported by feedback from Axel Jantsch.

1.3.2 Publications not included in this Thesis

- The paper [57] compares the languages Erlang [5], C++ [107], Haskell [62], VHDL [54], SDL [29] and ProGram [11] with respect to their capabilities as a system level description for telecommunication systems. The comparison is based on an evaluation method with a large number of criteria. In order to give specific emphasis on particular aspects of the application area, the criteria can be weighted. The evaluation was conducted for pure software, pure hardware and mixed hardware/software systems. The main work of the paper was conducted by Axel Jantsch and Shashi Kumar. The other authors contributed mainly with opinions on the weighting criteria and the evaluation of ”their” languages, in the case of the author of the thesis SDL, Erlang, C++, VHDL and Haskell.

- The paper [58] discusses the role of functions and objects in a system specification. The authors argue that functional models should be used for functional design space exploration and as functional reference model throughout all design and validation activities. In contrast, object oriented models should be used for architectural design space exploration and as a design specification for the following design and implementation phases. The main work of the paper has been done by Axel Jantsch; the contribution of the author of the thesis lies in the example used in the paper and feedback.

- The paper [59] reviews the use of non-determinism and identifies two different purposes. The *descriptive purpose* handles uncertainties in the behavior of existing entities, while the *constraining purpose* is used in specifications to constrain implementations. The authors suggest a *stochastic process* instead of non-determinism, which is used mainly for the descriptive purpose, but can also be used to constrain the system. The paper illustrates the usage of stochastic processes by means of ForSyDe for which a stochastic library has been defined. The main work in this paper has been done by Axel Jantsch; the author of this thesis supported the work with the integration of stochastic processes into ForSyDe and feedback.

1.4 Thesis Layout

The thesis consists of four main parts. The first part contains related work and other background information, which is needed to position ForSyDe and to understand the work described in this thesis (Chapter 2).

The second part presents the modeling concepts of ForSyDe. It is divided into two chapters. Chapter 3 defines the specification and implementation model and introduces the ForSyDe standard library. Chapter 4 illustrates the modeling technique by the example of a digital equalizer.

The third part describes transformational design refinement in ForSyDe (Chapter 5). It gives a format for semantic preserving and design decision transformation rules and shows how the implication of a transformation can be given by means of the characteristic function. It also shows the potential of the transformational approach by the digital equalizer example.

The fourth part of the thesis discusses the mapping to an implementation (Chapter 6). The chapter focuses on hardware synthesis, but also discusses what has to be done in order to target other implementations.

Chapter 7 summarizes the thesis and gives an overview about possible directions for future work on ForSyDe.

In addition the thesis contains several appendix chapters, since the thesis shall serve as a reference document on ForSyDe. Thus the appendix includes the ForSyDe standard library, the full equalizer model expressed in Haskell and mapping templates to VHDL.

Chapter 2

Background

This chapter provides background material in form of an overview of the related work and an introduction into the functional language Haskell. The first section discusses different models of computation. Since ForSyDe uses a synchronous model of computation, a more detailed overview of the family of synchronous languages is given in the second section. The third section gives an overview of other design methodologies. The section covers both "traditional" hardware/software co-design methodologies and other approaches that like ForSyDe use a declarative environment. The fourth section discusses work on design transformation. Finally, the last section gives an introduction into the functional language Haskell, since it is used for the simulation of ForSyDe models.

2.1 Models of Computation

System models are usually defined by a number of concurrent processes. A *model of computation* (MoC) defines how computation takes place in a structure of concurrent processes, thus giving a semantics to such a structure [28]. This semantics can be used to formulate an abstract machine that is able to execute a model. *Languages* are not computational models, but have underlying computational models. For instance the sequential imperative languages C, Pascal or Fortran share all the same imperative sequential computational model. On the other hand languages can be used to support more than one computational model. In this thesis the functional language Haskell [62] is used to express a synchronous and a multi-rate computational model as discussed in Chapter 3, but preliminary libraries have al-

ready been created for data flow and discrete event models and can be retrieved from the ForSyDe web page [1].

To choose the right model of computation for the modeling of systems is of utmost importance, since each MoC has certain properties. As a trivial case the sequential imperative model is not able to model concurrency and thus not suitable to describe a parallel structure.

The following part presents a number of important models of computations and gives their key properties. The discussion is based on the tagged-signal model introduced by Lee and Sangiovanni-Vincentelli, which is used as a framework to compare models of computation [70]. In the tagged-signal model systems are viewed as concurrent process networks where processes communicate with each other by means of signals. A signal is defined as a set of events where each event has a tag and a value. The tag is used to express an order relation between the events. Based on the tagged-signal model properties of different models of computation are discussed in [70]. This survey has been the major source for the following presentation of some important computational models.

Discrete Event Model A discrete event model is a *timed model*, which means that the events in all signals can be totally ordered by their tags. In contrast to continuous time models, there must be a finite number of tags between any two tags. Discrete event models are often used for the simulation of hardware. Both VHDL [54] and Verilog [53] use a discrete event model for their simulation semantics[1]. A discrete event simulator is usually implemented with a global event queue that automatically sorts the events. Discrete event models may have causality problems due to zero-delay in feedback loops, which are discussed in Section 2.2. A good overview on discrete-event system is given in [23].

Synchronous Model A synchronous model is a special case of a discrete-event system. In a synchronous model all signals have the same set of tags. Tags do not include explicit time information, but are only used to give an order of the events. The synchronous assumption can be formulated according to [9]. The synchronous approach considers "*ideal* reactive systems that produce their outputs *synchronously* with their inputs, their reaction taking no observable time". This implies that the computation of an output event is

[1] Both languages have a different model of computation for synthesis, which is similar to a perfect synchronous model due to the use of synchronous registers with the difference that computation does not have a zero delay.

instantaneous and thus the output event has the same tag as the corresponding input event. The synchronous assumption leads to a clean separation between computation and communication. A global clock triggers computations that are conceptually simultaneous and instantaneous. This assumption frees the designer from the modeling of complex communication mechanisms and provides a solid base for formal methods.

A synchronous design technique has been used in hardware design for clocked synchronous circuits. A circuit behavior can be described determinately independent of the detailed timing of gates by separating combinational blocks from each other with clocked registers. An implementation will have the same behavior as the abstract circuit under the assumption that the combinational blocks are ”fast enough” and that the abstract circuit does not include zero-delay feedback loops.

Feedback loops with zero-delay may cause causality problems in a synchronous model and are discussed together with the family of synchronous languages in Section 2.2.

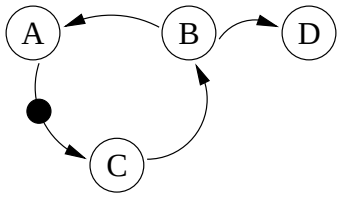


Figure 2.1. A data flow process network

Data Flow Process Networks Data flow process networks [69] are a special variant of Kahn process networks [63] [64]. In a Kahn process network processes communicate with each other via unbounded FIFO channels. Writing to these channels is *non-blocking*, i.e. they always succeed and do not stall the process, while reading from these channels is *blocking*, i.e. a process that reads from an empty channel will stall and can only continue when the channel contains sufficient data items (*tokens*). Processes in a Kahn process network are *monotonic*, which means that they only need partial information of the input stream in order to produce partial information of the output stream. Monotonicity allows parallelism, since a process does not need the whole input signal to start the computation of output events. Processes are not allowed to test an input channel for existence of tokens without consuming them. In a Kahn process network there is a total order of events inside

a signal. However, there is no order relation between events in different signals. Thus Kahn process networks are only partially ordered which classifies them as *untimed model*.

A data flow program is a directed graph consisting of nodes (*actors*) that represent communication and arcs that represent ordered sequences (*streams*) of events (*tokens*) as illustrated in Figure 2.1. Empty circles represent nodes, arrows represent streams and the filled circle represents a token. Data flow networks can be hierarchical since a node can represent a data flow graph.

The behavior of a data flow process is a sequence of *firings*. For each firing tokens are consumed and tokens are produced. The number of tokens consumed and produced may vary for each firing and is defined in the *firing rules* of a data flow actor. Data flow process networks have been shown very valuable in digital signal processing applications. A main goal in the design process is then to find a static firing schedule in order to implement the data flow process network efficiently on shared resources. For general data flow models it is undecidable whether such a schedule exists.

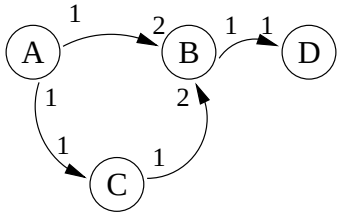


Figure 2.2. A synchronous data flow process network

Synchronous data flow (SDF) [67] [68] puts further restrictions on the data flow model, since it requires that a process consumes and produces a fixed number of tokens for each firing. With this restriction it is guaranteed that a finite static schedule can always be found. Figure 2.2 shows an SDF process network. The numbers on the arcs show how many tokens are produced and consumed during each firing. A possible schedule for the given SDF network is {A,A,C,C,B,D}.

There exists a variety of different data flow models, for an excellent overview see [69].

Rendezvous-based Models A rendezvous-based model consists of concurrent sequential processes. Processes communicate with each other only at synchronization points. In order to exchange information processes must have

reached this synchronization point, otherwise they have to wait for each other. In the tagged signal model each sequential process has its own set of tags. Only at synchronization points processes share the same tag. Thus there is a partial order of events in this model. The process algebra community uses rendezvous-based models. The CSP[2] model of Hoare [47] and the CCS[3] model of Milner [82] [83] are prominent examples.

From the discussion it is evident that different models fundamentally have different strength and weaknesses, and that attempts to combine their common features result in models that are very low level and difficult to use.

ForSyDe has adopted a synchronous computational model for specification due to its simple and clean mathematical formalism and the possibility to model time on an abstract level. This property is the reason why for instance data flow models have been neglected as ForSyDe specification model. Though they are supported by a mathematical framework and are excellently suited for their application areas, e.g. digital signal processing applications, they do not have an abstract notion of time and thus cannot be used to express timing properties and constraints on a level that abstracts from physical time.

Balarin et al. [7] argue, that the synchronous assumption, though very convenient from the analyzing point of view, imposes a too strong restriction on the implementation, as it has to be ”fast enough”. They advocate a GALS (globally asynchronous locally synchronous) approach and implement it in the POLIS methodology [6] as a network of Co-Design FSMs communicating via events. Each CFSM is a synchronous FSM, but the communication is done by the emission of events by the CFSMs, which can happen at any time and independently. The CFSM network is inherently non-deterministic. Balarin et al. argue, that this enables them to easily model the unpredictability of the reaction delay of a CFSM both at the specification and at the implementation level, while they admit, that non-determinism makes the design and verification process more complex.

ForSyDe has avoided a non-deterministic approach, since the advantages of a deterministic synchronous system model may well outweigh the disadvantages. Non-determinism in the system model implies that all possible solutions have to be considered, which makes both the design and the verification process more complex. The task to develop and to verify a system model for a SoC application is already extremely complex and it will be even more complex with a non-deterministic system model. The fact that SoC applications will be implemented on at least partly asynchronous architectures does not justify a non-deterministic

[2] Communicating Sequential Processes

[3] Calculus of Communicating Systems

approach. Instead the ForSyDe methodology is based on the idea that the synthesis of the system model into a partly asynchronous implementation should be part of the synthesis process and not already be decided at the system level.

Hsieh et al. define in [48] another *synchronous assumption*. A cycle consists of an interaction phase (where the environment interacts with the design) followed by a computation phase (where the components in the design perform computation and communicate internally). They do not assume a *zero delay* for the computation phase as in the case of the perfect synchrony hypothesis. Using their definition they define *synchronous equivalence* as: "Two implementations are synchronously equivalent if and only if any two synchronous assumption conforming runs of the two implementations that have the same input traces also have the same output traces". As long as the primary outputs are the same at the end of every cycle, the internal details of the execution do not matter. Thus the concept of synchronous equivalence can be used for design refinement. In ForSyDe *synchronous equivalence* can be shown with the characteristic function as discussed in Chapter 5. The use of the characteristic function is not restricted to synchronous models, but can also be implied on models with synchronous sub-domains.

Skillicorn and Talia discuss models of computation for parallel architectures in [105]. Their community faces similar or even identical problems as are typical for SoC design since a SoC architecture often includes a number of parallel microprocessors or other parallel hardware blocks. In fact all typical parallel computer structures (SIMD, MIMD[4]) can be implemented on a SoC architecture. Recognizing, that programming of a large number of communicating processors is an extremely complex task, they try to define properties for a suitable model of parallel computation. They emphasize that a model should hide most of the details (decomposition, mapping, communication, synchronization) from programmers, if they shall be able to manage intellectually the creation of software. The exact structure of the program should be inserted by the translation process rather than by the programmer. Thus models should be as abstract as possible, which means that the parallelism has not even to be made explicit in the program text. They point out that ad hoc compilation techniques cannot be expected to work on problems of this complexity, but advocate building software, that is correct by construction rather then verifying program properties after construction. Programs should be architecture independent to allow reuse. The model should support cost measures to guide the design process and should have guaranteed performance over a useful variety of architectures.

[4]Flynn has classified typical parallel data structures in [32], where SIMD is an abbreviation for Single Instruction, Multiple Data and MIMD for Multiple Instruction, Multiple Data.

Depending on what information is explicit in a model they distinguish six levels, i.e.

1. nothing explicit
2. parallelism explicit
3. parallelism and decomposition explicit
4. parallelism, decomposition and mapping explicit
5. parallelism, decomposition, mapping and communication explicit
6. parallelism, decomposition, mapping, communication and synchronization explicit

According to this scheme, the ForSyDe modeling approach can both be classified (a) with focus on modeling as 'nothing explicit', and (b) with focus on implementation as 'parallelism' and 'communication explicit'. The motivation for 'nothing explicit' is the use of the functional language Haskell [62] as ForSyDe modeling language, which has no explicit notion of parallelism. The motivation for (b) is that the specification model can be interpreted as a concurrent process model with a synchronous communication. However, neither the process nor the communication structure is fixed, since during the refinement phase processes can be merged and split and synchronous communication channels can be refined into asynchronous channels as elaborated in Chapter 5.

While ForSyDe uses a single unified system model, a lot of work has been done using mixed models of computation. This approach has the advantage, that a suitable model of computation can be used for each part of the system. On the other hand as the system model is based on several computational models, the semantics of the interaction of fundamentally different models has to be defined, which is not a simple task. This even amplifies the verification problem, because the system model is not based on a single semantics. There is little hope that formal verification techniques can help and thus simulation remains the only means of validation. In addition, once a heterogeneous system model is specified, it is very difficult to optimize systems across different models of computation. In summary, cross-domain verification and optimization will remain elusive for many years for any heterogeneous modeling approach. In the following an overview of related work on mixed models of computation is given.

In *charts [36] hierarchical finite state machines are embedded within a variety of concurrent models of computations. The idea is to decouple the concurrency

model from the hierarchical FSM semantics. An advantage is that modular components, e.g. basic FSMs, can be designed separately and composed into a system with the model of computation that best fits to the application domain. It is also possible to express a state in an FSM by a process network of a specific model of computation. *charts has been used to describe hierarchical FSMs that are composed using data flow, discrete event and synchronous models of computations.

The Ptolemy project [28] ”studies heterogeneous modeling, simulation, and design of concurrent systems”. It is implemented in the Ptolemy II software environment [55] that provides support for ”hierarchically combining a large variety of models of computation and allows hierarchical nesting of the models”.

The basic block of a Ptolemy model is the *actor*. Actors are concurrent components and communicate with each other through interfaces (*ports*). Actors can be composed of other actors and are executable. An implementation of a model of computation in Ptolemy associated with a composite actor is called a *domain*. The domain defines the communication semantics and execution order among actors. An actor can be used in different domains. This concept decouples transmission of data, transfer of control and computation (performed by the actor) from each other. To date several domains have been implemented in Ptolemy II, such as communicating sequential processes, continuous time, discrete event, process network and synchronous dataflow.

The MASCOT methodology [20] integrates data and control flow at the system specification level, using the two languages Matlab and SDL. The data flow parts are described in Matlab and the control flow parts in SDL, the system is then co-simulated using a library of wrappers and communication functions. The computational model is elaborated in [56].

Internal representations like the system property intervals (SPI) model [116] and FunState [106] have been developed to integrate a heterogeneous system model into one abstract internal representation. The idea of the SPI model is to allow for ”global system analysis and system optimization across language boundaries, in order to allow reliable and optimized implementations of heterogeneously specified embedded real-time systems”. All synthesis relevant information, such as resource utilization, communication and timing behavior, is extracted from the input languages and transformed into the semantics of the SPI model. An SPI model is a set of parameterized communicating processes, where the parameters are used for the adaptation of different models of computation. SPI allows to model nondeterminism through the use of behavioral intervals. There exists a software environment for SPI that is called the SPI workbench and is developed for the analysis and synthesis of heterogeneous systems.

The FunState representation refines the SPI model by adding the capability of explicitly modeling state information and thus allows the separation of data flow from control flow. The goal of FunState is not to provide a unifying specification, but it focuses only on specific design methods, in particular scheduling and verification. The internal FunState model shall reduce design complexity by representing only the properties of the system model relevant to these design methods.

2.2 Synchronous Languages

The synchronous languages [9] [42] [10] have been successfully used in the area of reactive and safety-critical embedded control systems. These languages are based on the synchronous computational model, which was discussed in Section 2.1. This model gives a solid mathematical foundation for formal reasoning and the application of formal program manipulation techniques. The synchronous languages have the following key properties.

- They support *concurrency*.
- They have a simple and elegant *formal semantics* that allows it to express parallel composition in a clean way.
- They support the concept of *synchrony*, which divides time into discrete instants.

The synchronous assumption implies a simple and formal communication model. Concurrent processes can easily be composed together. However, since the synchronous assumption implies a zero-delay between the output event and the corresponding input event, a feedback loop as illustrated in Figure 2.3 may imply no solution, one solution or many solutions. This problem is also valid in discrete event systems, since they also may have feedback loops with zero-delay.

Figure 2.3a shows a system with zero-delay feedback loop that does not have a stable solution. If the output of the Boolean AND function is True then the output of the NAND function is False. But this means that the output of the AND function has to be False, which is in contradiction to the starting point of the analysis. Starting with the value False on the output of AND does not lead to a stable solution either, since this implies that the output of the NAND function is True and thus the output of the AND function must be True. Clearly there is no solution to this problem.

Figure 2.3b shows a system with feedback loop with multiple solutions. Here the system is stable, if both AND functions have False or if both AND functions have True as their output value. Thus the system has two possible solutions.

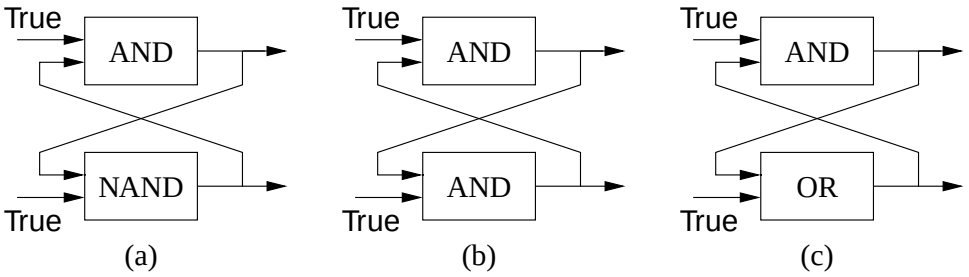


Figure 2.3. A feedback loop in a synchronous system. System a) has no solutions, b) has multiple solutions and c) has a single solution.

Figure 2.3c shows a system with feedback loop with only one solution. Here the only solution is that both outputs are True.

It is crucial for the design of safety-critical systems that feedback loops with no solution as in Figure 2.3a are detected and eliminated, since they result in an oscillator. Also feedback loops with multiple solutions imply a risk for safety-critical systems, since they as in 2.3b lead to non-determinism. Non-determinism may be acceptable, if it is detected and the designer is aware of its implications, but may have serious consequences, if it stays undetected.

Since feedback loops in synchronous models are of such importance there are several approaches which address this problem [28].

Microstep In order to introduce an order between events that are produced and consumed in an event cycle, the concept of microsteps has been introduced into languages like VHDL. VHDL does not belong to the synchronous languages, but has a similar model of time. In order to solve the zero-delay feedback problem, VHDL distinguishes between two dimensions of time. The first one is given by a time unit, e.g. a picosecond, while the second is given by a number of delta-delays. A delta-delay is an infinitesimal small amount of time. Each operation takes zero time units, but one delta-delay. Delta-delays are used to order operations within the same time unit. While this approach partly solves the zero-delay feedback problem, it introduces another problem, since delta delays will never cause the advance of time measured in time units. Thus during an event cycle there may be an infinite amount of delta-delays. This would be the result, if Figure 2.3a would be implemented in VHDL, since each operation causes time to advance with one delta-delay. An advantage of the delta-delay is that simulation will reveal that the composite function oscillates. However, a VHDL simulation would not detect that Figure 2.3b has two solutions, since the simulation seman-

tics of VHDL would assign an initial value for the output of the AND gates (False[5]) and thus would only give one stable solution, concealing the non-determinism from the designer. Another serious drawback of the microstep concept is that it leads to a more complicated semantics, which aggravates the task of formal reasoning.

Forbid zero-delays The easiest way to cope with the zero-delay feedback problem is to forbid them. In case of Figure 2.3a and 2.3b this would mean the insertion of an extra delay function, e.g. after the upper AND function. Since a delay function has an initial value the systems will stabilize. Assuming an initial value of True, Figure 2.3a will stabilize in the current event cycle with the values False for the output of the NAND function and False for the value of the AND function. Figure 2.3b would stabilize with the output value True for both AND functions. A problem with this approach is that a stable system such as 2.3c is rejected, since it contains a zero delay feedback loop. This approach is adopted in the synchronous language Lustre [43].

Unique fixed-point The idea of this approach is that a system is seen as a set of equations for which one solution in form of a fixed-point exists. There is a special value $\perp$ ("bottom") that allows it to give systems with no solution or many solutions a fixed-point solution. The advantage of this method is that the system can be regarded as a functional program, where formal analysis will show, if the system has a unique solution. Also systems that have a stable feedback loop as in Figure 2.3c are accepted, while the systems of Figure 2.3a and b are rejected (the result will be the value $\perp$ as solution for the feedback loops). Naturally, the fixed-point approach demands a more sophisticated semantics, but the theory is well understood [113]. Esterel has adopted this approach and the constructive semantics of Esterel is described in [15].

Relation based This approach allows the specification of systems as relations. Thus a system specification may have zero solutions, one solution or multiple solutions. Though an implementation of a system usually demands a unique solution, other solutions may be interesting for high-level specifications. The relation-based approach has been employed in the synchronous language Signal [39].

[5] VHDL defines the data type `boolean` by means of `type boolean is (false, true)`. At program start variables and signals take the leftmost value of their data type definitions, in case of the boolean data type the value `false`.

ForSyDe defines process networks as a set of equations. To date ForSyDe forbids zero-delays, since this approach leads to mathematically clean designs and does not require a sophisticated semantics. ForSyDe models are expressed and simulated in Haskell, where a zero-delay feedback loop as shown in Figure 2.3a or 2.3b results in a "control stack overflow"[6], while Haskell's lazy evaluation semantics, which is discussed in Section 2.5, in special cases may solve networks of the form of Figure 2.3c.

However, as shown by the case of Esterel [15] it is possible to develop a least fixed-point semantics for ForSyDe. This would also be an option, if in future other languages are incorporated to express ForSyDe models, but most likely zero-delays will be forbidden.

If other modeling languages are incorporated in future versions of ForSyDe, either a fixed-point semantics has to be developed for this new language, or another approach, most likely to forbid zero-delays has to be adopted.

The following part focuses largely on the presentation of the declarative data flow language Lustre since its concepts are closest to ForSyDe. After this presentation follows a short discussion about other synchronous languages. Lustre [43] is a declarative data flow language where systems are composed as sets of equations. Each variable is a function of time and denotes a *flow*. Operators operate not on single values, but on whole flows, i.e. the expression

```
y = a + b;
```

generates a flow y where each value y_k at instance k is the addition of the values a_k and b_k. In order to formulate sequential networks, Lustre defines two temporal operators. The operator $pre(x)$ returns the flow where the first value is undefined (expressed by nil) and each other value is the previous value of x. The operator "$->$" (followed-by) defines the initial value of a flow. If x and y are of the same type, "$x -> y$" is the flow that is equal to x in the first instance and equal to y thereafter. Using these operators a simple counter can be defined as

```
n = 0 -> pre(n) + 1;
```

Here 0 is the constant flow of zeros and 1 the constant flow of ones. The "followed-by" operator generates a flow where 0 is the first value followed by the flow $pre(n) + 1$. This flow is graphically shown in Figure 2.4.

Lustre programs can be recursive. However a variable may only depend on past values of itself in order to ensure that there are no zero delay feedback loops in the model.

[6] The error occurred during simulation with the Haskell interpreter Hugs98.

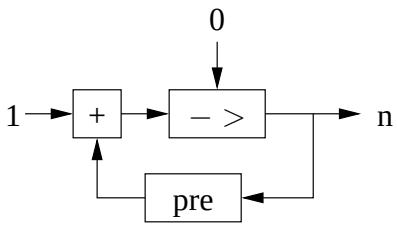


Figure 2.4. A counter in Lustre

Lustre provides the concept of *nodes* in order to structure programs. A node is in itself a flow operator and thus a Lustre program can be hierarchically composed.

A Lustre program has a *basic clock*, which defines the finest notion of time. Each flow is associated with a clock, but flows may have slower clocks than the basic clock. Most operators require that all input flows share the same clock. However, there is the operator $when$ that "samples" a flow to a slower clock and the operator $current$ that interpolates a flow on a clock that is faster than the own clock. Table 2.1 illustrates the use of these operators.

B	False	True	False	True	False	False	True	True
X	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
Y = X when B		x_2		x_4			x_7	x_8
Z = current Y	nil	x_2	x_2	x_4	x_4	x_4	x_7	x_8

Table 2.1. Sample and interpolation operators in Lustre

B expresses a Boolean flow that is associated with the basic clock. Also the flow X is associated with the basic clock. The clock of the flow Y is only defined at the instances where the Boolean flow B has the value True. As shown by the example of Z a flow with a slower clock can be up-sampled by the operator $current$.

Esterel [22] [14] [16] is an imperative language that is very suitable for the description of control. A program consists of a collection of nested, concurrently running threads that are described in an imperative syntax. Threads communicate with each other by means of signals. In addition to usual control signals such as `if-then-else`, Esterel has a large number of preemption statements that allow the termination of statements. There is a formal framework developed for Esterel, which includes *causality analysis* ensuring that causality constraints are never contradictory in any reachable state.

A Signal [39] program is a specification of *constraints* or *relations* on the involved signals. The Signal compiler performs formal calculations on synchronization, logic, and data dependencies to check program correctness and produce executable code. Signal is a declarative data flow language and supports multiple clocks in a similar way as Lustre.

Statecharts [46] and Argos [75] are graphical automata-based languages.

The synchronous languages have been successfully used in industry and there exist industrial tools for Esterel (from Esterel Technology), Lustre (Scade) and Signal (Sildex). Recently Esterel Technology has acquired the Scade environment in order to be able to combine a control-oriented (Esterel) with a data flow oriented (Lustre) synchronous approach.

Synchronous language programs are usually translated to finite state automata in order to implement them as a sequential reactive program on a single processor. However, Esterel [13] and Lustre [93] have been also translated into hardware implementations.

The clean mathematical formalism has led to the development of several verification tools for the synchronous languages. The paper [44] gives an overview over the techniques and tools developed for the validation of reactive systems described in Lustre. However, the authors point out that these techniques can be adapted to any synchronous language.

Since ForSyDe is functional, it has similar characteristics as Lustre. In both environments processes (nodes) communicate with each other by means of synchronous signals (flows). As further discussed in Section 3.3 the ForSyDe specification model uses one basic clock and slower clocks can be defined by absent values. However, ForSyDe operators do not operate on signals, but on values. Thus the basic processes in ForSyDe may be of much higher complexity than the basic operators (which are processes in the ForSyDe sense) of Lustre. Also, all complex operators such as pre, "$->$", $when$ and $current$ exist in similar forms in ForSyDe. ForSyDe uses a fully equipped modeling language with powerful control constructs and data type facilities and thus gives better possibilities to model control or complex data structures than Lustre.

In contrast to Lustre, ForSyDe offers the concept of synchronous sub-domains inside the implementation model. This allows it to define clocks that are faster or slower than the clock of the specification model by the use of domain interfaces (Section 3.4). In Lustre only slower clocks than the basic clock can be defined, since the basic clock has to be the fastest clock.

ForSyDe is based on the same foundation as the synchronous languages, the synchronous assumption, but aims to cover both, control and data flow appli-

cations. In order to achieve this goal it uses a formal system modeling technique where models can be expressed with the purely functional programming language Haskell. While functional languages fit naturally for data flow applications, Haskell provides a rich variety of control constructs, making it more suitable for control-dominated applications. Haskell is purely functional, i.e. a function has no side effects, resulting in a system model, that in itself is a function with no side effects and thus deterministic.

While these properties of Haskell mainly support formal verification methods, other properties support design correctness. According to Lee [66], type systems do more than any other formal method to ensure software correctness. Haskell is not only a strongly typed language, but its type system [37] has also the ability to infer the correct type of a function, which offers another dimension of polymorphism compared to some popular object oriented languages, such as C++ or Java.

2.3 Design Methodologies

This section gives an overview of other design methodologies. The first part of this section discusses design methodologies that have emerged from the research of the hardware/software co-design community. This community has close links to the electronic design automation industry. The second part gives an overview of declarative approaches to system design, which is research from another community. Unfortunately there is not too much interaction between these communities, but ForSyDe uses results of both areas.

2.3.1 Hardware/Software Co-Design

Hardware/software co-design is defined in [78] as the study of the design of embedded computing systems. To date the term *system design* is often used as a synonym for hardware/software co-design. According to [78] hardware/software co-design covers the following problems:

Co-specification The creation of specifications that describe both the hardware and software of a system and their interaction;

Co-synthesis The automatic or semi-automatic design of hardware and software to meet a specification;

Co-simulation The simultaneous simulation of hardware and software elements, often at different levels of abstractions.

The ForSyDe methodology targets system design and can according to the definition above be classified as a hardware/software co-design methodology. However, since ForSyDe is a research project, it does not cover all parts of a hardware/software co-design flow to the same extent. In fact, some parts are so far not covered at all.

ForSyDe addresses the co-specification problem, since it proposes a modeling methodology that describes the functionality of the whole system, at that state not distinguishing between hardware and software, in a single and executable specification (Chapter 3 and 4).

The transformational design refinement approach described in chapter 5 where the specification model is refined into a more detailed implementation model, should belong to the domain of co-synthesis, but is not covered by the definition of Wolf [114] where co-synthesis includes four different tasks:

Partitioning The functionality of the system is divided into smaller, interacting computation units.

Allocation The decision, which computational resources are used to implement the functionality of the system.

Scheduling If several system functions have to share the same resource, the usage of the resource must be scheduled in time.

Mapping The selection of a particular allocated computational resource for each computation unit.

Clearly these tasks are depending on each other. To date ForSyDe does not propose methods for partitioning, allocation, scheduling or mapping. These activities have to be performed manually. However, ForSyDe addresses the mapping from a partitioned model to hardware (Chapter 6) and sequential software [73].

ForSyDe allows the simulation of both the specification model and the more detailed and possibly partitioned implementation model, which may be viewed as co-simulation at a high abstraction level. In the following some hardware/software co-design methodologies are discussed in more detail.

For a more elaborate overview on the hardware/software co-design process see [30] or [79].

Vulcan

Vulcan [41] [40] has mainly been developed for co-synthesis. The functionality is formulated in HardwareC, which is based on C and models a system as concurrent processes communicating with each other. The target architecture consists of

a processor, a memory and a set of ASICs[7]. Co-synthesis starts with a configuration where all functions that can be implemented in hardware are implemented as hardware modules. All other modules are implemented in software. Then Vulcan tries to reduce hardware costs by moving functions from hardware to software as long as the performance constraints can be satisfied. Finally the resulting partition serves as input to high-level synthesis and software compilation tools.

COSYMA

COSYMA (CO-SYnthesis for eMbedded Architectures) [31] was developed about the same time as Vulcan. In contrast to Vulcan, co-synthesis starts from a configuration, where all functions are implemented in software. The advantage with this approach is that the system may include functions that cannot be implemented in hardware, such as dynamic data structures. COSYMA uses C^x as input language, which extends C with the concept of tasks. The target architecture consists of a processor, a memory and custom hardware.

POLIS

The POLIS [6] system is designed for control-dominated systems, where the target architecture consists of a micro-controller and ASICs. POLIS uses a globally asynchronous and locally synchronous internal format that is based on Co-design Finite State Machines (CFSMs). A CFSM is a finite state machine that is extended in order to be able to communicate with other CFSMs asynchronously. The execution delay of each CFSM is unknown, but assumed to be non-zero in order to avoid zero-delay feedback loops. The input specification consists of parallel FSMs expressed in Esterel, which are connected by communication links. This specification can be directly translated to CFSMs. POLIS does not offer any hardware/-software partitioning algorithm and leaves the choice to the designer, but allows to synthesize the CFSM model to hardware and software. POLIS uses Ptolemy as simulation environment. The commercial tool VCC[8] from Cadence is based on the POLIS approach.

SpecC

The SpecC system-level design methodology [35] [26] follows a top-down approach and starts with the development of a *specification model* expressed in the

[7] Application Specific Integrated Circuits
[8] Virtual Component Co-Design

language SpecC. SpecC is an extension of C and provides special language constructs for modeling concurrency, state transitions, structural and behavioral hierarchy, exception handling, timing, communication and synchronization. The specification model defines the granularity of the following exploration phase by the size of the leaf behaviors and the available parallelism. Otherwise it is free from implementation details and a notion of time. The specification model includes also non-functional constraints that are imposed on the design.

During *architecture exploration* the system architecture is derived from the specification model. This phase includes allocation, partitioning and scheduling. The output of this phase is the *architecture model* that reflects the component structure of the system architecture and is annotated with estimated execution delays.

In the *communication synthesis* phase the abstract communications in the architectural model are refined into communication protocols. The refined model, the *communication model* is then translated into an *implementation model* by the usage of compilers and high-level synthesis tools.

The strength of the SpecC methodology is that all models are described in the same language with different amount of details. Thus the same tools can be used for different models of a design.

MESCAL

The MESCAL[9] project [80] was recently formulated in order to ”develop the methodologies, tools, and appropriate algorithms to support the efficient development of fully programmable, platform-based designs for specific application domains” [65]. A goal is to develop a platform that can be used efficiently for various applications inside the same application domain. These domain specific platforms should be highly programmable in order to provide the needed flexibility. Another goal is to develop the MESCAL compiler, which allows to map source applications on a family of programmable platforms and microarchitectures. In order to map the application onto a complex communication structure MESCAL advocates to use the OSI[10] stack model with a set of formal semantics in order to perform a correct-by-construction synthesis from the high-level description to an implementation [101]. The MESCAL methodology reflects the trend in today’s system design. Due to an increasing integration of functionality on a single-chip, the focus is moved to platform-based design, where formal design refinement as required by

[9] Modern Embedded Systems, Compilers, Architectures, and Languages

[10] The OSI (Open Systems Interconnect) reference model defines a protocol stack of seven protocol layers and has been mainly used for the specification of telecommunication systems.

a correct-by-construction approach plays an important role. The concept of networks on a chip [60] is an example for an emerging platform for system design.

2.3.2 Declarative Approaches to System Design

Declarative languages have been used in other research projects in electronic design.

Reekie [92] has used Haskell to model digital signal processing applications. Similar to ForSyDe he modeled streams as infinite lists and used higher-order functions to operate on them. Finally, correctness-preserving methods were applied to transform a model into a more efficient representation. This representation was not synthesized to hardware or software.

Hydra [87] has been developed for educational purpose and is a hardware description language embedded in Haskell. Circuits are modeled as functions and Hydra provides powerful structured composition operators in the form of higher-order functions that allow it to express regular circuit patterns in elegant style.

Ruby [61] is a relational language that has mainly been used for hardware design. In the same way as Hydra, Ruby focuses on the structural composition of the design by providing a variety of efficient composition functions. In [74] a declarative framework for hardware/software co-design based on Ruby has been proposed. Ruby also supports transformations based on equational reasoning and supports data type refinement. There exists a path to an implementation in hardware or software.

Lava [19] [24] is a hardware description language based on Haskell. Similar to Ruby it focuses on the structural representation of hardware and offers a variety of powerful connection patterns. Lava descriptions can be translated into VHDL or mapped to Xilinx FPGAs. There exist interfaces to formal method tools. Recently Singh [103] proposed to formulate system level specifications in Lava and to use Haskell's type classes to describe communication links at different levels of abstraction. However, since Lava focuses on a structural representation of a system, it is less suited to express behavioral specifications.

Mycroft and Sharp have used the languages SAFL (statically allocated functional language) and SAFL+ [102] mainly for hardware design but extended their approach in [86] to hardware/software co-design. They transform SAFL programs by means of meaning preserving transformations and compile the resulting program in a resource-aware manner, i.e. a function that is called more than once will be a shared resource.

The Hawk language [76], which is embedded in Haskell, is used for building executable specifications of microprocessors. The Hawk project addresses the need

for verification of complex modern microprocessors, which is supported by the formal nature of a Hawk specification. Hawk has been used to specify and simulate the integer part of a pipelined DLX processor.

Hardware ML (HML) [71] is a hardware description language that is based on the functional programming language Standard ML [84]. Though HML uses some features of Standard ML, such as polymorphic functions and its type system, it is mainly an improvement of VHDL - there is a direct mapping from HML constructs into corresponding VHDL constructs.

ForSyDe is most similar to Reekie's approach and can be viewed as an extension and further development of Reekie's work. Mycroft and Sharp follow with their SAFL language a similar intention as ForSyDe as they also intend to move refinement to a higher level. However, they restrict themselves to semantic preserving transformations in contrast to ForSyDe, which also allows for design decision transformations as discussed in Chapter 5. Lava, Ruby and HML are different in that they perform hardware modeling and design at a lower level than ForSyDe. While their modeling concepts can be seen as competitors to VHDL and Verilog, in ForSyDe these languages are target languages and hardware synthesis tools are back-end tools. Hawk is different in that it addresses modeling and verification of instruction sets and processor architectures. ForSyDe targets are more general hardware architectures and embedded software running on processors, but not the processor design itself.

Functional reactive programming (FRP) [110] has been used in many reactive programming domains, such as animation, robotics, and graphical user interfaces. The central semantic notions are *behavior*, a function of continuous time, and *event*, a time-ordered sequence of event occurrences. FRP has initially been implemented as an embedded language in Haskell. Since FRP does not guarantee any bounds on execution time or speed it is unsuitable for real-time applications. To address this problem, Real-Time FRP (RT-FRP) [111] has been developed with an operational semantics that gives these guarantees. In contrast to FRP, the operational semantics define a discrete model of time that is used for both events and behaviors. Recently Event-Driven FRP (E-FRP) [112] has been presented which is a variant of RT-FRP. In E-FRP the clock is generalized to a set of events and has no explicit notion of time. Thus behaviors and events in E-FRP correspond to ForSyDe signals, where an E-FRP event can be viewed as a ForSyDe signal with values of an extended data type, i.e. the data type includes the absent value, and E-FRP behaviors correspond to ForSyDe signals with values of non-extended data types. So far E-FRP has only been used to target embedded software for a microcontroller, but no work with a hardware platform as target architecture has been

reported.

2.4 Design Transformation

While the high level of abstraction fits well for system level specification, there is a gap between the system model and a possible implementation on a SoC architecture. ForSyDe tries to bridge this gap by the concept of process constructors. Though the system model is formulated as a function, the use of process constructors implies, that the functional model can be interpreted as a network of synchronously communicating concurrent processes. Such a process structure is almost fixed in other design languages (VHDL, SDL), but in ForSyDe processes can be merged and split during the application of transformation rules during the design transformation phase (Chapter 5). As each process constructor has a hardware and software interpretation, the refined implementation model can be interpreted as a structure with hardware and software components (Chapter 6).

The ForSyDe concept of process constructors is heavily influenced by the work of Skillicorn on *homomorphic skeletons* [104]. The term skeleton, coined by Cole [25], has been used in the parallel programming community to denote an abstract building block that has a predefined implementation on a parallel machine. In order to obtain an implementation the abstract program must be composed of these skeletons. The advantage of such an approach is that it raises the level of abstraction, since programmers program in their language and do not even have to be aware of the underlying parallel architecture. Specialists can be used to design the implementation of these skeletons.

Algorithmic skeletons have been introduced by Cole [25] and are based on algorithms and encapsulate their control structure. Skillicorn [104] uses homomorphic skeletons that are based on composite data types. This approach uses the Bird-Meertens formalism and allows also equational reasoning. Bird demonstrates how to derive programs from specifications using lists [17], arrays and trees [18] as data types. As Skillicorn points out implementations with guaranteed performance can be built for computers that are based on standard topologies. Also cost measures can be provided since the complete schedule of computation and communication is known from the implementation of the skeleton.

The concept of ForSyDe process constructors is based on the work on homomorphic skeletons. The work of Bird on equational reasoning on list homomorphisms has been the starting point for the work on semantic preserving transformations of ForSyDe process networks.

A good overview about program transformation in general is given in [88] and for transformation of functional and logical programs in [89]. One of the most well-known transformation systems is the CIP (computer-aided, intuition-guided programming) project [8]. Inside CIP, program development is viewed as an evolutionary process that usually starts with a formal problem specification and ends with an executable program for the intended target machine. The individual transformations are done by semantic preserving transformation rules, which guarantees that the final version of the program still satisfies the initial specification. Such an approach has the following advantages [8]:

- the final program is correct by construction;
- the transitions can be described by schematic rules and thus be reused for a whole class of problems;
- due to formality the whole process can be supported by the computer;
- the overall structure is no longer fixed throughout the development process, so that the approach is quite flexible.

However, in order to allow for a successful transformation of a specification into an effective implementation, a transformation framework has to provide a sufficient number of transformation rules and there must also exist a transformation strategy in order to choose a suitable order of transformation rules. This strategy ideally interacts with an estimation tool that shows if one implementation is more efficient than another. Since program transformation requires a well-developed framework, it has so far been mainly used for small programs or modules inside a larger program, where software correctness is critical.

Most of the transformational approaches are concerned with software programs where concepts of synchronous sub-domains and resource sharing, as discussed in this thesis, have no relevance. There are also a number of other transformational approaches targeting hardware design, e.g. [90] [100], but none of them explicitly develops the concept of design decisions or addresses the refinement of a synchronous model into multiple synchronous sub-domains as we attempt in this article. In particular the ForSyDe approach allows to use the large amount of work that exists for high-level synthesis [33] [77] by defining design decision transformations for refinement techniques like re-timing or resource sharing.

Voeten points out that each transformational design that is based on a general-purpose language will suffer from fundamental incompleteness problems [109]. This means that the initial model to a large extent determines whether an effective and satisfying implementation can be obtained or not, since only a limited part of

the design space can be explored. The same problem is known in the context of high-level synthesis as syntactic variance problem [34], which in general is unsolvable.

2.5 Introduction to Haskell

Since the functional language Haskell is used to express the system models in ForSyDe, this section gives a small introduction to functional languages and Haskell in particular. Many examples are taken from "A Gentle Introduction to Haskell 98" [51], a good textbook on Haskell is [108].

A functional program is a function that receives the program's input as argument and delivers the program's output as result. Usually the main function is defined in terms of other functions, which can be composed of still other functions until at the bottom of the functional hierarchy the functions are language primitives. Each function is free from side-effects, i.e. they have no internal state. This means that the whole functional program is free from side-effects and thus behaves totally deterministic. Given the same inputs, the functional program will always produce identical outputs. Since all functions are free from side-effects, the order of evaluation is only given by data dependencies. But this means also that there may exist several possible orders for the execution of a functional program. Considering the function

$$f(x, y) \quad = \quad u(h(x), g(y))$$

the data dependencies imply that the functions $h(x)$ and $g(y)$ have to be evaluated before $u((h(x), g(y))$ can be evaluated. However, since there is no data dependency between the functions h and g, there are the following possible orders of execution:

- $h(x)$ is evaluated before $g(y)$;
- $g(y)$ is evaluated before $h(x)$;
- $h(x)$ and $g(y)$ are evaluated in parallel.

Thus functional programs contain implicit parallelism, which is very useful when dealing with embedded system applications, since they typically have a considerable amount of built-in parallelism. Of course it is also possible to parallelize imperative languages like C++, but it is much more difficult to extract parallelism from programs in such languages, since the flow of control is also expressed by the order of statements.

The foundations of functional languages have been discussed in depth in [49], while [52] discusses how functional languages can be used to improve modularity. There is also a study [50] that supports the claimed advantages of functional languages - brevity, rapidity of development and ease of understanding - over conventional imperative languages as C++ and ADA.

The following part introduces the functional programming language Haskell and shows some important features.

To date the ForSyDe methodology uses Haskell 98 which is defined in "Haskell 98 Language and Libraries" [62]. Haskell's standard library, the "standard prelude" defines basic types, operators and functions. Other modules can be developed and imported, such as the ForSyDe Standard Library, which defines data types and functions for ForSyDe.

In addition to common data types, such as `Bool`, `Int` and `Double`, the standard prelude also defines lists and tuples. An example for a list is `[1,2,3,4] :: [Integer]`, which is a list of integers. The notation "`::`" means "has type". An example for a tuple, which is a structure of different types is `('A', 3) :: (Char, Integer)` where the first element is a character and the second one is an integer.

Haskell has adopted the Hindley-Milner type system [81], which was developed for the functional language ML [37]. It has the following significant features [49]:

1. It is strongly and statically typed.
2. It uses type inference to determine the types of every expression, instead of relying on explicit type declarations.
3. It allows polymorphic functions and data structures; that is, functions may take arguments of arbitrary type, if in fact the function does not depend on that type (similarly for data structures).
4. It has user-defined constructs and abstract data types.

The type system is not only strongly typed, like the type system for VHDL or ADA, but it is also capable to infer (calculate) the maximal possible data type for an expression. Given the function

```
fst (x, y) = x
```

which returns the first value of a pair, Haskell's type system will determine the type as

```
fst :: (a,b) -> a
```

This type declaration means that `fst` is a function that has a tuple (a pair) as input parameter, where the first value is of some data type `a` and the second value is of some data type `b`. The output value is of data type `a`. The type declaration does not imply that `a` and `b` have to be of a different type, but since the type declaration gives the maximal type, it is allowed that they are of different types.

The function `fst` can now be used with all kinds of pairs, such as `fst(3, [1,2,3])`, `fst(1,2)` or `fst('A', 3)`.

Haskell is based on the lambda-calculus and allows to write functions in *curried* form, after the mathematician Haskell B. Curry, where the arguments are written by juxtaposition. The function `add` is written in curried form.

```
add :: Num a => a -> a -> a
add x y = x + y
```

Since '`->`' associates from right to left, the "real" type of `add` is `add :: Num a => a -> (a -> a)`. This means that given the first argument, which is of a numeric type `a`, it returns a function from `a` to `a`. This can be used for *partial application* of a curried function. New functions can then be defined by applying the first argument, e.g.

```
inc x = add 1
dec x = dec 1
```

These functions only have one argument and the following type

```
inc :: Num a => a -> a
dec :: Num a => a -> a
```

It is not possible to use partial application, if the *uncurried* form of `add` is used,

```
add :: Num a => (a, a) -> a
add(x, y) = x + y
```

since there is only one argument, which is a tuple of two values and must be supplied as a whole.

Another powerful concept in functional languages is the *higher-order function*. A higher-order function is a function that takes functions as argument and/or produces a function as output. An example of a higher-order function is `map`, which takes a function and a list as argument and applies ("maps") the function `f` on each value in the list. The function is defined as follows

```
map f []     = []            -- Pattern 1 (empty list)
map f (x:xs) = f x : map f xs -- Pattern 2 (all other lists)
```

The higher-order function uses an additional feature of the language, which is called *pattern matching* and is illustrated by the evaluation of `map (+1) [1,2,3]`.

```
    map (+1) [1,2,3]
⇒   map (+1) (1:[2,3])          Pattern 2 matches
⇒   1+1 :  map (+1) [2,3]       Evaluation of Pattern 2
⇒   2 :  map (+1) (2:[3])       Pattern 2 matches
⇒   2 :  2+1 :  map (+1) [3]    Evaluation of Pattern 2
⇒   2 :  3 :  map (+1) (3:[])   Pattern 2 matches
⇒   2 :  3 :  4 :  map (+1) []  Evaluation of Pattern 2
⇒   2 :  3 :  4 :  map (+1) []  Pattern 1 matches
⇒   2 :  3 :  4 :  []           Evaluation of Pattern 2
⇒   [2,3,4]
```

During an evaluation the patterns are tested from the top to the bottom. If a pattern, the left hand side, matches, the corresponding right hand side is evaluated. The expression `map (+1) [1,2,3]` does not match the first pattern since the list is not empty (`[]`). The second pattern matches, since (`x:xs`) matches a list that is constructed of a single value and a list. Since the second pattern matches, the right hand side of this pattern is evaluated. This procedure is repeated recursively until the first pattern matches, where the right hand side does not include a new function call. As this example shows, lists are constructed and processed from head to tail.

The higher-order function `map` can now be used with all functions and lists that fulfill the type declaration for `map`, which Haskell infers as

```
map :: (a -> b) -> [a] -> [b]
```

The type declaration reads as follows. The first argument of `map` is a function that takes a value of some data type `a` and returns a value of another data type `b`. The second argument is a list of some data type `b` and the result is a list of some data type `b`.

Thus functions as `fst` and `map` are polymorph and can be used with several types. However, Haskell has a static type system, that ensures that Haskell programs are *type safe*, i.e. all type errors are detected at compile time. The type system will allow function calls like `map even [1,2,3]`, which will be evaluated to a list of Boolean values `[False, True, False] :: [Bool]`, but reject the list `[1,'A',3]` since this list contains elements of different types.

A very powerful higher-order function is *function composition*, which is expressed by the composition operator '.'.

```
(.)     :: (b -> c) -> (a -> b) -> (a -> c)
```

```
f . g    = \x -> f (g x)
```

This definition uses ”lambda abstractions” and is read as follows. The higher-order function `f . g` produces a function that takes a value x as argument and produces the value $f(g(x))$. The expression `f = (+3) . (*4)` creates a function `f` that performs $f(x) = 4x + 3$. Function composition is extremely useful in ForSyDe since it is allows to merge processes in a structured way.

Haskell allows to define own data types using a `data` declaration. It allows for recursive and polymorph declarations. A data type for a list could be recursively defined as

```
data AList a = Empty
             | Cons a (AList a)
```

The declaration has two *data constructors*. The data constructor `Empty` constructs the empty list and `Cons` constructs a list by adding a value of type `a` to a list. Thus `Cons 1 (Cons 2 (Cons 3 Empty))` constructs a list of numbers. The term *type constructor* denotes a constructor that yields a type. In this case `AList` is a type constructor.

As mentioned before, in Haskell 98 the list data type is predefined. Here `[]` corresponds to `Empty` and `:` to `Cons`. `[a]` corresponds to `AList a`.

An important aspect of Haskell is that it uses *lazy evaluation*, which is in contrast to *eager evaluation*. Here, the value $\perp_T$ is used to denote the *value* for a non-terminating expression or an expression that result in some kind of run-time error, such as `1/0`. A function f is *strict*, if

$$f(\perp_T) = \perp_T$$

In most program languages *all* functions are strict and this means that all arguments to a function must terminate and must not cause a run-time error in order to terminate f. But this is not the case in a lazy language as Haskell. Consider the constant 1 function `const1`, defined by:

```
const1 x = 1
```

The result of `const` $\perp_T$ `1` in Haskell is `1`, since Haskell never attempts to evaluate its argument as the result is always `1`. Functions like `const1` are *non-strict* functions, which are also called “lazy functions”, as they evaluate their arguments “lazily” or “by need”. Thus expressions like `const1 (1/0)` evaluate to `1`. In an eager language like Standard ML [84], where all functions are strict, the result will be a run-time error since the program tries to evaluate `1/0`.

Lazy evaluation is of great use when dealing with possibly infinite data structures such as signals, since the program will only evaluate an expression when needed. Thus, it is possible to define an infinite list of the natural numbers `nat` as

```
  nat = numsFrom 1
  where
    numsFrom n = n : numsFrom (n+1)
```

Of course it is still not possible to evaluate the whole (infinite structure) `nat`, but finite parts of it can be evaluated such as

```
head nat                 =>   1
take 5 nat               =>   [1,2,3,4,5]
(take 5 . map (^2)) nat  =>   [1,4,9,16,25]
```

Abstract data types can be defined using the Haskell module system. A module is a collection of functions. Inside a module all functions and data types are visible for each other. Other modules may only access the functions that are explicitly exported by the module. The following code shows a part of the module `SynchronousLib`, which is part of the ForSyDe Standard Library.

```
module SynchronousLib(
                        module Vector, module Signal, module AbsentExt,
                        mapSY, zipWithSY, zipWith3SY,
                        zipWith4SY, zipWithxSY, scanlSY,
                        scanl2SY, scanl3SY, scanldSY, scanld2SY,
                        scanld3SY, delaySY, delaynSY, whenSY,
                        fillSY, holdSY, zipSY, zip3SY, unzipSY,
                        unzip3SY, zipxSY, unzipxSY, mapxSY, mooreSY,
                        moore2SY, moore3SY, mealySY, mealy2SY,
                        mealy3SY, fstSY, sndSY, groupSY, sourceSY
                      ) where

import Signal
import Vector
import AbsentExt
```

The module imports the libraries `Signal`, `Vector` and `AbstExt`, which are part of the ForSyDe core language (Section 3.5). It exports the modules `Signal`, `Vector`, `AbstExt` and the functions given in the parameter list of the module `SynchronousLib` beginning with `mapSY` until `sourceSY`. Functions that are defined in the module, but not listed in the parameter list are not visible to other modules.

The ”Haskell Home Page” [2] is a web page that works as a portal for Haskell related tools and information. Haskell compilers and interpreters are freely available. The ForSyDe project uses mainly the interpreter Hugs98 and the compiler Glasgow Haskell Compiler (GHC). All examples in this thesis have been tested with Hugs98.

Chapter 3

System Models of ForSyDe

This chapter defines the system models of ForSyDe, i.e. the specification model and the implementation model. The specification model is based on a synchronous model of computation. The implementation model extends the specification model through the use of domain interfaces with the possibility to establish synchronous sub-domains. It uses a multi-rate model of computation. The chapter introduces the concept of process constructors and domain interfaces and presents how system models are formulated. In addition an overview of the ForSyDe Standard Library is given, which serves as a basis to express ForSyDe models in the functional language Haskell.

3.1 Formal Models in ForSyDe

The design process in ForSyDe starts with an abstract and high-level specification model. The idea of ForSyDe is to bridge the abstraction gap between the specification model and the implementation by means of transformational refinement inside the functional domain. In order to define design transformations formally it is of utmost importance that the models used in ForSyDe are formal and well-defined. Also formal verification methods, though not incorporated into ForSyDe at present, require a formal semantics.

ForSyDe defines two system models. The specification model is based on a synchronous computational model, since this model offers a clean mathematical formalism and allows for a high level abstraction of physical time, which is needed to formulate timing constraints. The implementation model shall include all necessary and low-level implementation details in order to allow for an efficient mapping

onto a system architecture. So far the implementation model uses a synchronous computational model extended by synchronous sub-domains. This model is called a *multi-rate model* throughout this thesis and allows a direct and efficient mapping to hardware as discussed in Chapter 6. However other computational models may be more suitable for the mapping onto other architectures, for instance a data flow model may be more suitable for software. Thus the intention is to allow additional computational models inside the ForSyDe implementation model. In order to use these models they must be defined formally, and transformation rules and efficient mapping techniques to architectural components must be developed. Also interfaces between all computational models that are allowed in the implementation model have to be developed. Some preliminary libraries have already been developed for ForSyDe and can be downloaded from the ForSyDe web page [1].

The definition of ForSyDe is based on mathematics and independent of a modeling language. Processes are defined as functions and compositions of processes result in another process. As discussed in Section 2.2 ForSyDe does not explicitly define a semantics for zero-delay feedback loops, but simply forbids them. To date the functional language Haskell is used to express a ForSyDe model, since the functional paradigm fits nicely with the formal definition of ForSyDe models.

The next section gives a framework for the definition of computational models in ForSyDe. Section 3.3 defines the specification model and Section 3.4 the implementation model. Finally, Section 3.5 presents how ForSyDe models are expressed in Haskell, the modeling language of ForSyDe.

3.2 Definition of the Computational Models of ForSyDe

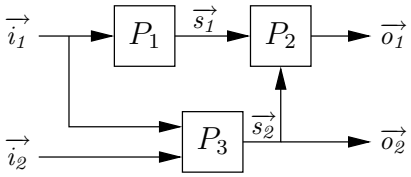


Figure 3.1. Systems are modeled as communicating concurrent processes

In order to compare the properties of different models of computation, Lee and Sangiovanni-Vincentelli have developed a *framework for comparing models of computation* [70] that is based on the view of a system as a *network of concurrent processes* as illustrated in Figure 3.1. The general definition of the ForSyDe

computational models used in this thesis uses a similar notation and is closely related to the basic concepts of this framework.

3.2.1 Signals

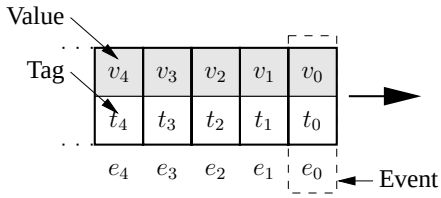


Figure 3.2. A signal is a sequence of events

A signal is a sequence of *events*, where each event has a *tag* and a *value*. Tags can be used to model physical time, the order of events and other key properties of the computational model.

Definition 3.1 (Event) *An event e has a tag $t \in \mathbf{T}$ and a value $v \in \mathbf{V}$.*

$$e = (t, v)$$

The set of all events is denoted by $\mathbf{E}$.

The set of tags $\mathbf{T}$ depends on the computational model. While some models of computation model physical time with the tags, other models use them only to express the order of events. This order can even be a partial order as in the case of data flow process networks.

The set of values $\mathbf{V}$ is not restricted to certain data types. However, some computational models, such as the synchronous computational model, define a special value $\perp$ to model the absence of a value at a given tag. A data type $\mathbf{D}$ can be extended to $\mathbf{D}_\perp$ by adding the value $\perp$.

$$\mathbf{D}_\perp = \mathbf{D} \cup \{\perp\}$$

Such a data type is called an *extended data type*.

The function $T_e(e)$ extracts the tag and the function $V_e(e)$ extracts the value of an event $e = (t, v)$.

$$\begin{aligned} T_e(e) &= t \\ V_e(e) &= v \end{aligned}$$

Definition 3.2 (Signal) *A* signal $\overrightarrow{s}$ *is an infinite sequence of events according to*

$$\begin{array}{l}\overrightarrow{s} =< e_0, e_1, \cdots > \\ \text{where} \\ \forall j \in \mathbb{N}_0 . V_e(e_j) \in \mathbf{V} \wedge T_e(e_j) \in \mathbf{T} \\ \forall j \in \mathbb{N}_0 . T_e(e_j) \leq T_e(e_{j+1})\end{array}$$

The set of all signals is denoted by $\mathbf{S}$.

Definition 3.2 implies that all events in a signal have values of the same data type $\mathbf{V}$ and that these are sorted by the tags in ascending order. The relation $T_e(e_j) \leq T_e(e_{j+1})$ is used to be able to integrate additional computational models, such as data flow models, into ForSyDe. For the existing ForSyDe models the relation $T_e(e_j) < T_e(e_{j+1})$ is sufficient.

A signal with values of an extended data type is called an *extended signal*. Definition 3.2 does not put any restriction on the order relation of the events in different signals. They may not be related to each other at all.

The following functions are defined in order to access events in a signal $\overrightarrow{s} = < e_0, e_1, \cdots >$.

$$\begin{aligned} E(\overrightarrow{s}, j) &= e_j \\ T(\overrightarrow{s}, j) &= T_e(e_j) \\ V(\overrightarrow{s}, j) &= V_e(e_j) \end{aligned}$$

For convenience, the operator $\bigoplus$ is introduced to express a possibly infinite sequence $x =< x_0, x_1, \ldots, x_n >$ in a different way.

$$x = \bigoplus_{j=0}^{n} x_j =< x_0, x_1, \ldots, x_n >$$

Thus a signal $\overrightarrow{s} =< e_0, e_1, \cdots >$ can be expressed as $\overrightarrow{s} = \bigoplus_{j=0}^{\infty} e_j$.

The function $T_S(\overrightarrow{s})$ returns the tags in a signal $\overrightarrow{s}$. The function $V_S(\overrightarrow{s})$ returns the values in a signal $\overrightarrow{s}$.

$$\begin{aligned} T_S(\overrightarrow{s}) &= < T(\overrightarrow{s}, 0), T(\overrightarrow{s}, 1), \cdots > &= \bigoplus_{j=0}^{\infty} T(\overrightarrow{s}, j) \\ V_S(\overrightarrow{s}) &= < V(\overrightarrow{s}, 0), V(\overrightarrow{s}, 1), \cdots > &= \bigoplus_{j=0}^{\infty} V(\overrightarrow{s}, j) \end{aligned}$$

In this thesis input signals are usually denoted by $\overrightarrow{i}$, output signals by $\overrightarrow{o}$ and internal signals by $\overrightarrow{s}$.

3.2.2 Processes

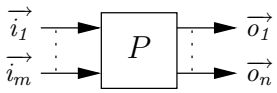


Figure 3.3. Process

A process (Figure 3.3) maps input signals onto output signals. Depending on the computational model that is used, different types of processes can be defined.

Definition 3.3 (Process) *A* process P *is a functional mapping from* $m \in \mathbb{N}_0$ *input signals* $\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}$ *onto* $n \in \mathbb{N}_1$ *output signals* $\overrightarrow{o_1}, \ldots, \overrightarrow{o_n}$.

$$P(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}) \quad = \quad (\overrightarrow{o_1}, \ldots, \overrightarrow{o_n})$$

The set of all processes is denoted by $\mathbf{P}$.

Definition 3.3 does also include processes that do not have any input signals. These processes can be used to model signal *sources*. A useful example is a timer.

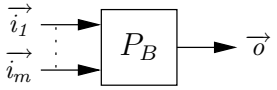


Figure 3.4. A basic process has only one output signal

A *basic process* (Figure 3.4) is a process with only one output signal.

Definition 3.4 (Basic Process) *A* basic process P_B *is a functional mapping from* $m \in \mathbb{N}_0$ *input signals* $\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}$ *onto one output signal* $\overrightarrow{o}$.

$$P_B(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}) \quad = \quad \overrightarrow{o}$$

The set of all basic processes is denoted by $\mathbf{P_B}$.

Processes can be composed of other processes. The term *process network* is used for processes that are composed by other processes, but includes also single

processes. The process network of Figure 3.1 is in itself a process P and can be formulated as

$$
\begin{aligned}
&P(\overrightarrow{i_1}, \overrightarrow{i_2}) = (\overrightarrow{o_1}, \overrightarrow{o_2}) \\
&\text{where} \\
&\quad \overrightarrow{o_1} = P_2(\overrightarrow{s_1}, \overrightarrow{s_2}) \\
&\quad \overrightarrow{o_2} = \overrightarrow{s_2} \\
&\quad \overrightarrow{s_1} = P_1(\overrightarrow{i_1}) \\
&\quad \overrightarrow{s_2} = P_3(\overrightarrow{i_1}, \overrightarrow{i_2})
\end{aligned}
$$

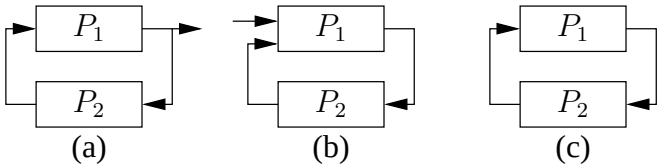


Figure 3.5. Composition of processes can lead to compositions with no input (a), no output (b) or neither input nor output (c)

Using process composition it is possible to produce compositions (Figure 3.5) that have no input (a), no output (b) or neither input nor output (c). However, since process are functional mappings from input to output signals, compositions with no output are not sensible and cannot be regarded as processes. On the other hand compositions with only output signals and no input signal can be viewed as constant functional mappings and are thus valid processes. These processes are called *sources* and can be used in the modeling process as signal generators.

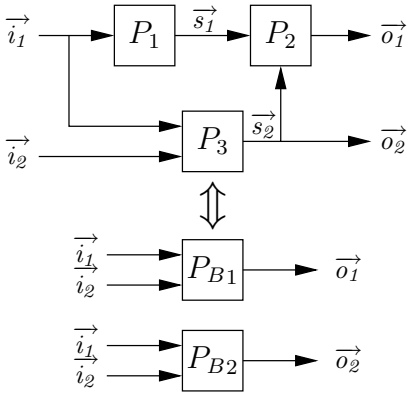
Proposition 1 states that all processes can be composed of basic processes.

Proposition 1 *Any process P with the input signals $\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}$ and the outputs signals $\overrightarrow{o_1}, \ldots, \overrightarrow{o_n}$ can be modeled as a network PN of n basic processes $P_1, \ldots, P_n$ where*

$$
\begin{aligned}
&\exists PN(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}).PN(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}) = P(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}) \\
&\text{where} \\
&\quad P(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}) = (\overrightarrow{o_1}, \ldots, \overrightarrow{o_n}) \\
&\quad PN(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}) = (\overrightarrow{o_1}, \ldots, \overrightarrow{o_n}) \\
&\quad \text{where} \\
&\qquad \overrightarrow{o_1} = P_1(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}) \\
&\qquad \vdots \\
&\qquad \overrightarrow{o_n} = P_n(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m})
\end{aligned}
$$

Since processes are mathematical functions of the input signals, Proposition 1 can be explained as follows. Since the tuple of outputs is a function of all inputs, each single output must also be a function of the inputs and thus can be modeled as a basic process.

Considering again the process network of Figure 3.1, it can be modeled by two basic processes P_{B1} and P_{B2} according to Proposition 1. This equivalence is illustrated in Figure 3.6.



$$
\begin{aligned}
&P(\overrightarrow{i_1}, \overrightarrow{i_2}) = (\overrightarrow{o_1}, \overrightarrow{o_2}) \\
&\text{where} \\
&\quad \overrightarrow{o_1} = P_{B1}(\overrightarrow{i_1}, \overrightarrow{i_2}) \\
&\quad \overrightarrow{o_2} = P_{B2}(\overrightarrow{i_1}, \overrightarrow{i_2}) \\
&\quad \text{where} \\
&\qquad P_{B1}(\overrightarrow{i_1}, \overrightarrow{i_2}) = P_2(P_1(\overrightarrow{i_1}), P_3(\overrightarrow{i_1}, \overrightarrow{i_2})) \\
&\qquad P_{B2}(\overrightarrow{i_1}, \overrightarrow{i_2}) = P_3(\overrightarrow{i_1}, \overrightarrow{i_2})
\end{aligned}
$$

Figure 3.6. Two equivalent process networks

ForSyDe defines two composition operators in order to allow sequential (Definition 3.6) and parallel composition (Definition 3.7). Sequential composition is based on function composition as defined in Definition 3.5.

Definition 3.5 (Function Composition) *Two functions $f : \gamma \rightarrow \beta$ and $g : (\alpha_1, \ldots, \alpha_n) \rightarrow \gamma$ can be composed into a new function $h : (\alpha_1, \ldots, \alpha_n) \rightarrow \beta$ by application of the* function composition operator $\circ_n$.

$$
\begin{aligned}
&h = f \circ_n g \\
&\text{where} \\
&\ h(x_1, \ldots, x_n) = f(g(x_1, \ldots, x_n))
\end{aligned}
$$

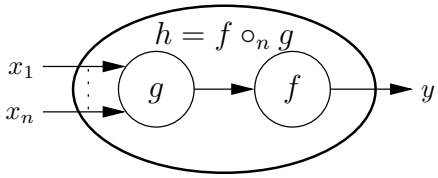


Figure 3.7. Function composition

Function composition is illustrated in Figure 3.7.

Definition 3.6 (Sequential Process Composition) *Two processes* $P_1 : \mathbf{S}_\gamma \rightarrow \mathbf{S}_\beta$ *and* $P_2 : (\mathbf{S}_{\alpha_1}, \ldots, \mathbf{S}_{\alpha_m}) \rightarrow \mathbf{S}_\gamma$ *can be composed sequentially by means of the function composition operator* $\circ_m$ *to yield a basic process* $P_B : (\mathbf{S}_{\alpha_1}, \ldots, \mathbf{S}_{\alpha_m}) \rightarrow \mathbf{S}_\beta$.

$$P = P_1 \circ_m P_2$$
where
$$P(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}) = P_1(P_2(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}))$$

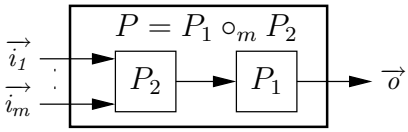


Figure 3.8. Sequential process composition

Sequential process composition is illustrated in Figure 3.8

Definition 3.7 (Parallel Process Composition) *Two processes* $P_1 : \mathbf{S}_{\alpha_1} \rightarrow \mathbf{S}_{\beta_1}$ *and* $P_2 : \mathbf{S}_{\alpha_2} \rightarrow \mathbf{S}_{\beta_2}$ *can be composed by means of the parallel composition operator* $\|$ *to yield a new process* $P : (\mathbf{S}_{\alpha_1}, \mathbf{S}_{\alpha_2}) \rightarrow (\mathbf{S}_{\beta_1}, \mathbf{S}_{\beta_2})$.

$$P = P_1 \parallel P_2$$
where
$$P(\overrightarrow{i_1}, \overrightarrow{i_2}) = (P_1(\overrightarrow{i_1}), P_2(\overrightarrow{i_2}))$$

Parallel process composition is illustrated in Figure 3.9.

Other compositions including non-zero delay feedback loops can be defined as sets of equations.

The computational model of the specification model can be seen as a special case of the implementation model. To date the implementation model is restricted

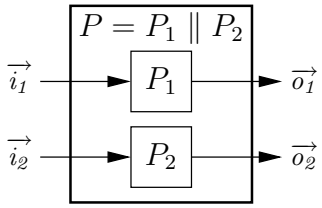


Figure 3.9. Parallel process composition

to multi-rate models and does not allow additional computational models. In order to establish a multi-rate model, domain interfaces are introduced, which allow to establish process networks with a different set of tags inside the implementation model. These sub-networks are internally also based on the synchronous assumption. However they do not share the same set of tags as in the main network. These models are denoted *multi-rate models*, since they have several basic clocks (one for each sub-network) with a different event rate.

To characterize signals in a multi-rate model the term *periodic signal* is defined in Definition 3.8.

Definition 3.8 (Periodic Signal) *A signal* $\overrightarrow{s} =< e_0, e_1, \cdots >$ *is a* periodic signal*, if*

$$\exists c \in \mathbb{Q}. \forall j \in \mathbb{N}_0. T_e(e_{j+1}) = T_e(e_j) + c \quad c > 0$$
$$T(e_0) = 0$$

where c *is the* event cycle *of the signal* $\overrightarrow{s}$*. The function* $C(\overrightarrow{s})$ *returns the event cycle of a periodic signal* $\overrightarrow{s}$*.*

$$C(\overrightarrow{s}) = c$$

The subset of signals that are periodic is denoted by $\mathbf{S_P}$*.*

The term *signal rate* is defined in Definition 3.9.

Definition 3.9 (Signal Rate) *A periodic signal* $\overrightarrow{s} \in \mathbf{S_P}$ *is associated with a* signal rate $R(\overrightarrow{s})$ *according to*

$$R(\overrightarrow{s}) = \frac{1}{C(\overrightarrow{s})}$$

Using Definitions 3.8 and 3.9 the set of tags for a periodic signal can be determined either by the event cycle or the signal rate according to

$$\begin{aligned} T_s(\overrightarrow{s}) &= < 0, C(\overrightarrow{s}), 2C(\overrightarrow{s}), \cdots > \\ &= < 0, 1/R(\overrightarrow{s}), 2/R(\overrightarrow{s}), \cdots > \end{aligned}$$

Since a periodic signal $\overrightarrow{s} \in \mathbf{S_p}$ is completely defined by the set of values $V_s(\overrightarrow{s})$ and the signal rate $R(\overrightarrow{s})$, the short notation

$$\begin{aligned} \overrightarrow{s} &= < (0, v_0), (1/R(\overrightarrow{s}), v_1), (2/R(\overrightarrow{s}), v_2), \cdots > \\ &= \ll v_0, v_1, \ldots \gg_{R(\overrightarrow{s})} \end{aligned}$$

is introduced.

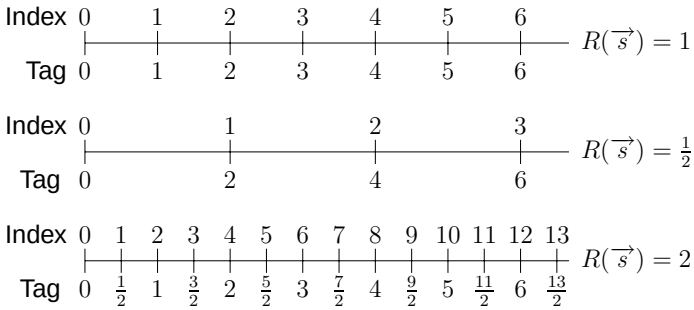


Figure 3.10. Relation of tags in periodic signals

Figure 3.10 visualizes the relation of tags for periodic signals with different signal rates. All signals have their first event at tag 0. The following events appear periodically with an event cycle $C(\overrightarrow{s}) = 1/R(\overrightarrow{s})$. If all signals in a process network are periodic, there is a total order of the tags in all signals.

Note, that the tags do not give an explicit notion of time, but only a total order of the events. Whenever two events have the same tag, they occur at the same instance of time. It does not mean that the event rate implies a periodic clock in the final implementation. In fact a multi-rate system model can be implemented with no clock at all as long as the total order of events is preserved.

It is possible to transform a multi-rate model into a synchronous model by defining a basic clock with an event rate that is calculated by the least common multiplier of the event rates of all sub-networks. However this may lead to a large amount of absent values, since these are needed to synchronize the clocks. If the following event rates exist in a model, $R(\overrightarrow{s_1}) = 1$, $R(\overrightarrow{s_2}) = 2$, $R(\overrightarrow{s_3}) = 3$ and $R(\overrightarrow{s_4}) = 4$, then the least common multiplier is 12 and signal $\overrightarrow{s_1}$ will have 11 absent values for each present value.

The multi-rate model restricts the implementation to a larger extent than synchronous data flow, since it implies a total order between the events of different sub-networks. The multi-rate model was chosen with a synchronous hardware implementation in mind. Thus when mapping to synchronous hardware, this does not restrict the implementation and allows a simple and efficient mapping. However, a

mapping to a distributed and asynchronously communicating implementation will impose unnecessary requirements on the implementation, since the total order of all tags, even those that are not meant for synchronization have to be preserved. Thus the multi-rate model should be restricted to a special class of implementations and in order to allow for a more efficient mapping to other architecture components, other models of computations, e.g. SDF, should be integrated into ForSyDe.

In order to model aperiodic signals the specification and implementation model define the value $\perp$ to denote an absent value. The ForSyDe methodology also defines the function Ψ that extends a function $f : D \rightarrow R$ into an extended function $f_\perp : D_\perp \rightarrow R_\perp$ that is also defined for the absent value.

$$
\begin{array}{l}
\Psi(f(x)) = f_\perp(x) \\
\text{where} \\
\begin{array}{lcl}
f & : & D \rightarrow R \\
f_\perp & : & D_\perp \rightarrow R_\perp \\
f_\perp(x) & = & \begin{cases} \perp & \text{if } x = \perp \\ f(x) & \text{otherwise} \end{cases}
\end{array}
\end{array}
\tag{3.1}
$$

To date the ForSyDe methodology is restricted to periodic processes (Definition 3.10), since ForSyDe only allows a multi-rate model in the implementation model.

Definition 3.10 (Periodic Process) *A* periodic process P_P *is a functional mapping from* m *input signals* $\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}$ *onto* n *output signals* $\overrightarrow{o_1}, \ldots, \overrightarrow{o_n}$*, where all input signals and output signals are periodic signals.*

$$
\begin{array}{l}
P(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}) = (\overrightarrow{o_1}, \ldots, \overrightarrow{o_n}) \\
\text{where} \\
\overrightarrow{i_j} \in \mathbf{S_P} \quad \forall j.1 \leq j \leq m \\
\overrightarrow{o_k} \in \mathbf{S_P} \quad \forall k.1 \leq k \leq n
\end{array}
$$

The set of periodic processes is denoted by $\mathbf{P_P}$.

In ForSyDe most processes are constructed by *process constructors*[1]. A process constructor takes functions and values as argument and generates a process as output. Functions are used to model the computation behavior inside a process, while values are used to define parameters or the value of an initial state of a process. Examples for process constructors are given later in this chapter.

[1]There is the exception of *combinator processes* that do not perform any computation. They transform signals of tuples into tuples of signals and vice versa. The combinator processes $zipSY_m$ and $unzipSY_n$ are defined in the next section.

Definition 3.11 (Process Constructor) *A* process constructor PC *is a function that takes* 0 *to* m *combinational functions* f_i *and* 0 *to* n *values* v_i *and produces a process* P *as output.*

$$PC(f_1, \ldots, f_m, v_1, \ldots, v_n) = P \in \mathbf{P}$$

The set of all process constructors is denoted by $\mathbf{PC}$.

3.3 The Specification Model

A ForSyDe specification model is a network of concurrent synchronous processes. These processes communicate with each other synchronously by means of signals, which are normalized to a signal rate $r_S = 1$. Thus, all signals have the same set of tags, i.e.

$$T_s(\overrightarrow{s}) =< 0, 1, 2, \cdots >$$

Since all signals in the specification model have the same signal rate $R(\overrightarrow{s}) = 1$, the following short notation for a signal $\overrightarrow{s}$ with a signal rate $R(\overrightarrow{s}) = 1$ is introduced:

$$\begin{aligned} \overrightarrow{s} &= \ll v_0, v_1, v_2, \ldots \gg_1 \\ &= \ll v_0, v_1, v_2, \ldots \gg \end{aligned}$$

The specification model is based on the synchronous assumption, which also forms the base for the family of the synchronous languages (Section 2.2). The synchronous abstraction leads to a clean separation between computation and communication. Below follows the formal definition of a synchronous process and of the specification model.

Definition 3.12 (Synchronous Process) *A* synchronous process P_S *is a periodic process where all input signals* $\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}$ *and all output signals* $\overrightarrow{o_1}, \ldots, \overrightarrow{o_n}$ *have the same signal rate* r.

$$\begin{aligned} &P(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}) = (\overrightarrow{o_1}, \ldots, \overrightarrow{o_n}) \\ &\text{where} \\ &\quad P_S \in \mathbf{P_P} \\ &\quad R(\overrightarrow{i_j}) = r \quad \forall i.1 \leq j \leq m \\ &\quad R(\overrightarrow{o_k}) = r \quad \forall j.1 \leq k \leq n \end{aligned}$$

The set of all synchronous processes is denoted by $\mathbf{P_S}$.

Definition 3.13 (Specification Model) *A specification model M_S is a process network that is composed of synchronous processes and models a system. All signals in the process network have the signal rate $r_S = 1$.*

$$M_S \in \mathbf{P_S}$$

The set of all specification models is denoted by $\mathbf{M_S}$.

The specification model reflects the design principles of the ForSyDe methodology. In order to allow for formal design on a high abstraction level, the specification model has the following characteristics:

- It is based on a *synchronous computational model*, which cleanly separates computation from communication.
- It is *purely functional* and *deterministic*.
- It uses *ideal data types* such as lists with infinite size.
- It uses the concept of well defined *process constructors* which implement the synchronous computational model.
- It is formally defined and can be expressed in the functional language Haskell [62].

The specification model abstracts from implementation details, such as buffer sizes and low-level communication mechanisms. This enables the designer to focus on the functional behavior on the system rather than structure and architecture. This abstract nature leaves a wide design space for further design exploration and design refinement, which is supported by the transformational refinement techniques of ForSyDe (Chapter 5).

In order to be able to model signals with aperiodic behavior or lower data rates, a data type $\mathbf{V}$ can be extended to an extended data type $\mathbf{V}_\perp = \mathbf{V} \cup \{\perp\}$ by adding the absent value $\perp$. Thus a signal

$$\overrightarrow{s} = \ll \perp, \perp, \perp, \perp, \perp, Reset, \perp, \perp \gg$$

models a reset signal with the value $Reset$ at tag 5, and no value at all other tags.

To model a synchronous process the concept of *synchronous process constructors* is used. A synchronous process constructor is a higher-order function that may take combinational functions, i.e a function that has no internal state, and variables as argument and produces a synchronous process as output. For the specification

model only synchronous process constructors are allowed. However, it is possible to formulate process constructors for other computational models and these may be integrated in future into the implementation model.

Definition 3.14 (Synchronous Process Constructor) *A* synchronous process constructor PC_S *is a function that takes* 0 *to* m *combinational functions* f_i *and* 0 *to* n *values* v_i *and has a basic synchronous process* P_S *as output.*

$$PC_S(f_1, \ldots, f_m, v_1, \ldots, v_n) \quad = \quad P_S \in \mathbf{P_S}$$

The set of synchronous process constructors is denoted by $\mathbf{PC_S}$.

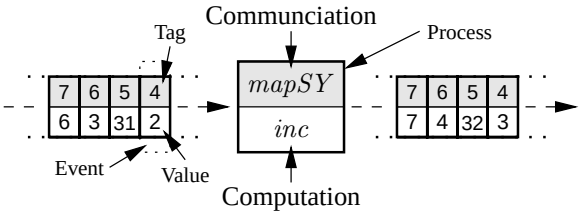


Figure 3.11. Synchronous process constructors separate timing from communication

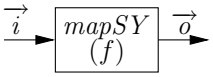
The concept of synchronous process constructors is illustrated by means of Figure 3.11. Here the synchronous process constructor $mapSY$, which is formally defined in Figure 3.12, takes the increment function inc to construct a process that maps the function inc on each value of the input signal.

Since the computational model is synchronous each produced output event has the same tag as the corresponding consumed input event. Figure 3.11 shows clearly the separation between timing (gray shaded), performed by the process constructor $mapSY$, and computation (white shaded) which is performed by the supplied function inc. The process is formally defined as

$$\begin{aligned} &P_S = mapSY\,(inc) \\ &\text{where} \\ &\quad inc(x) \quad = \quad x + 1 \end{aligned}$$

The process constructor $mapSY$ is used to model combinational processes with one input and one output signal. The supplied function defines the computational behavior of the process.

The ForSyDe methodology obliges the designer to use synchronous process constructors for the modeling of processes in the specification model. This leads to



$$
\begin{aligned}
&mapSY(f) = P_S \in \mathbf{P_S} \\
&\text{where} \\
&\overrightarrow{o} \quad = \quad P_S(\overrightarrow{i}) \\
&T(\overrightarrow{o}, j) \quad = \quad T(\overrightarrow{i}, j) \qquad \forall j \in \mathbb{N}_0 \\
&V(\overrightarrow{o}, j) \quad = \quad f(V(\overrightarrow{i}, j)) \quad \forall j \in \mathbb{N}_0
\end{aligned}
$$

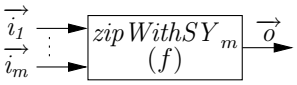
Figure 3.12. The process constructor $mapSY$

a well-defined specification model where almost all processes are constructed by synchronous process constructors. Please remember that definition 3.11 disallows the use of non-combinational functions as argument for process constructors.

The concept of synchronous process constructors gives the following benefits:

- Synchronous process constructors implement the *synchronous computational model*.
- Due to the construction of processes with process constructors and combinational functions, there is a *clean separation between synchronization and computation*. Synchronization is implemented inside the process constructors and computation is achieved by the supplied combinational functions.
- There is a family of process constructors that generate a process with a *local state*. However, there is no explicit global state in the model, which would make it more difficult to reason formally about the system model.
- Process constructors have a *structural hardware and software interpretation*. This means, that a system model, which is composed of process constructors also has an interpretation in hardware, software or a mixture of both. The hardware semantics is described in Chapter 6, for the software semantics please refer to [73].
- As process constructors are higher-order functions, the work on *correctness-preserving transformations* [89] can be used for the development of semantic preserving transformations (Chapter 5)
- A *characteristic function*, which is a useful tool for design transformations, can be determined for all process constructors (Section 5.2).

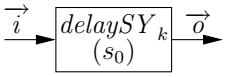
In the following we define additional synchronous process constructors for combinational processes, i.e. processes that have no internal state, and for sequential process, i.e. processes that have an internal state.



$$
\begin{aligned}
&zipWithSY_m(f) = P_S \in \mathbf{P_S} \\
&\text{where} \\
&\overrightarrow{o} \quad = \quad P_S(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}) \\
&T(\overrightarrow{o}, j) \quad = \quad T(\overrightarrow{i_1}, j) = \cdots = T(\overrightarrow{i_m}, j) \quad \forall j \in \mathbb{N}_0 \\
&V(\overrightarrow{o}, j) \quad = \quad f(V(\overrightarrow{i_1}, j), \ldots, V(\overrightarrow{i_m}, j)) \quad \forall j \in \mathbb{N}_0
\end{aligned}
$$

Figure 3.13. The process constructor $zipWithSY$

The synchronous process constructor $zipWithSY_m$ applies a function f to the current values of m input signals $\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}$ (Figure 3.13). It is used to model combinational processes with more than one inputs.



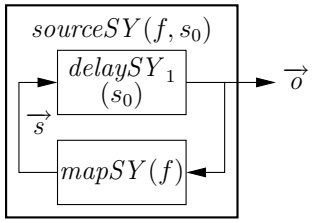
$$
\begin{aligned}
&delaySY_k(s_0) = P_S \in \mathbf{P_S} \\
&\text{where} \\
&\overrightarrow{o} \quad = \quad P_S(\overrightarrow{i}) \\
&T(\overrightarrow{o}, j) \quad = \quad T(\overrightarrow{i}, j) \qquad\qquad\qquad\qquad \forall j \in \mathbb{N}_0 \\
&V(\overrightarrow{o}, j) \quad = \quad \begin{cases} s_0 & \text{if } j < k \\ V(\overrightarrow{i}, j-k) & \text{otherwise} \end{cases} \quad \forall j \in \mathbb{N}_0
\end{aligned}
$$

Figure 3.14. The process constructor $delaySY$

The basic sequential synchronous process constructor is $delaySY_k$ (Figure 3.14). It takes an initial value s_0 and delays a signal $\overrightarrow{i}$ by k event cycles as illustrated by the following example.

$$delaySY_2(\perp) \ll 1, 2, 3, 4, \ldots \gg = \ll \perp, \perp, 1, 2, 3, 4, \ldots \gg$$

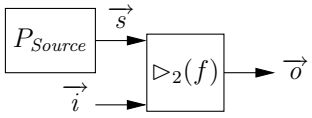
In the following we also use the short notations $\diamond$ for $mapSY$, $\triangleright_m$ for $zipWithSY_m$ and $\triangle_k$ for $delaySY_k$. More complex process constructors are defined by a composition of the basic constructors $mapSY$, $zipWithSY_m$ and $delaySY_k$.



$$sourceSY(f, s_0) = P_S \in \mathbf{P_S}$$
where
$$\begin{array}{rcl} \overrightarrow{o} & = & P_S \\ \overrightarrow{o} & = & delaySY_1(s_0)(\overrightarrow{s}) \\ \overrightarrow{s} & = & mapSY(f)(\overrightarrow{o}) \end{array}$$

Figure 3.15. The process constructor $sourceSY$

The process constructor $sourceSY(f, s_0)$ (Figure 3.15) is used to define a source process. The process will generate an infinite series of output values. Since there is no input signal, the tags of the output signal of the source process are implicitly defined by other signals and processes that are part of the system model as indicated in Figure 3.16.



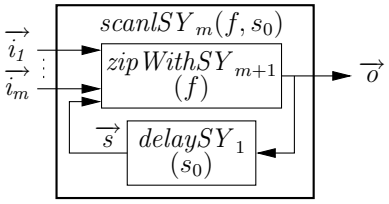
$$T_S(\overrightarrow{s}) = T_S(\overrightarrow{i}) = T_S(\overrightarrow{o})$$

Figure 3.16. The tags in an output signal of a source process are implicitly defined by other signals and processes in the system model

A useful example for a process based on $sourceSY$ is a counter that counts from m to n and is expressed as follows:

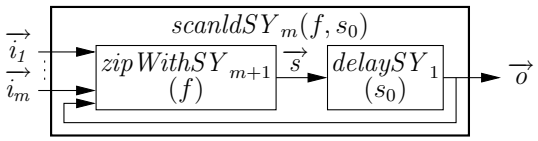
$$counterSY(m, n) = sourceSY(f, m)$$
where
$$f(x) = \begin{cases} m & \text{if } x \geq n \\ x + 1 & \text{otherwise} \end{cases}$$

The process constructors $scanlSY_m(f, s_0)$ (Figure 3.17) and $scanldSY_m(f, s_0)$ (Figure 3.18) model finite state machines without an output decoder. The difference is the location of the delay process. This means that the output of a process



$$
\begin{aligned}
&scanlSY_m(f, s_0) = P_S \in \mathbf{P_S} \\
&\text{where} \\
&\overrightarrow{o} \;=\; P_S(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}) \\
&\overrightarrow{o} \;=\; zipWithSY_{m+1}(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}, \overrightarrow{s}) \\
&\overrightarrow{s} \;=\; delaySY_1(s_0)(\overrightarrow{o})
\end{aligned}
$$

Figure 3.17. The process constructor $scanlSY$



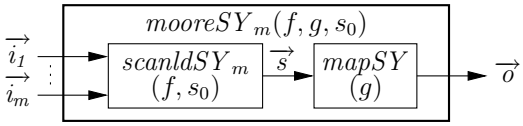
$$
\begin{aligned}
&scanldSY_m(f, s_0) = P_S \in \mathbf{P_S} \\
&\text{where} \\
&\overrightarrow{o} \;=\; P_S(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}) \\
&\overrightarrow{o} \;=\; delaySY_1(s_0)(\overrightarrow{s}) \\
&\overrightarrow{s} \;=\; zipWithSY_{m+1}(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}, \overrightarrow{o})
\end{aligned}
$$

Figure 3.18. The process constructor $scanldSY$

$scanldSY(f, s_0)$ is one event cycle delayed compared to a process $scanlSY(f, s_0)$ as shown in the following example.

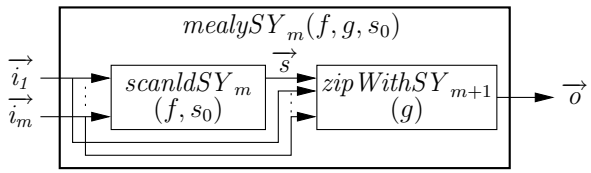
$$
\begin{aligned}
scanlSY((+), 0) \ll 1, 2, 3, 4, \ldots \gg &= \ll 1, 3, 6, 10, \ldots \gg \\
scanldSY((+), 0) \ll 1, 2, 3, 4, \ldots \gg &= \ll 0, 1, 3, 6, 10, \ldots \gg
\end{aligned}
$$

The synchronous process constructors $mooreSY_m$ (Figure 3.19) and $mealySY_m$ (Figure 3.20) model finite state machines of Moore and Mealy type. The difference between these FSMs is the following. In a Moore machine the result of an output decoder (represented by the process $mapSY(g)$) does only depend on the state (represented by $\overrightarrow{s}$), but not on the input signals. In a Mealy machine, the result of the output decoder ($zipWithSY_{m+1}(g)$) depends not only on the state, but also on the input signals. Both process constructors use $scanldSY(f, s_0)$ in their definition



$$mooreSY_m(f, g, s_0) = P_S \in \mathbf{P_S}$$
where
$$\begin{aligned} \overrightarrow{o} &= P_S(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}) \\ \overrightarrow{o} &= mapSY(g)(\overrightarrow{s}) \\ \overrightarrow{s} &= scanldSY_m(f, s_0)(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}) \end{aligned}$$

Figure 3.19. The process constructor $mooreSY$



$$mealySY_k(f, g, s_0) = P_S \in \mathbf{P_S}$$
where
$$\begin{aligned} \overrightarrow{o} &= P_S(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}) \\ \overrightarrow{o} &= zipWithSY_{m+1}(g)(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}, \overrightarrow{s}) \\ \overrightarrow{s} &= scanldSY_m(f, s_0)(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}) \end{aligned}$$

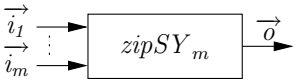
Figure 3.20. The process constructor $mealySY$

since this follows traditional synchronous hardware design style where a register is located after the next state decoder in order to make the design more reliable.

ForSyDe defines the *combinator processes* $zipSY_m$ (Figure 3.21) and $unzipSY_n$ (Figure 3.22). These processes do not perform any computation but convert a tuple of signals into signals of tuples ($zipSY_m$) and vice versa ($unzipSY_n$) as shown in the following example.

$$\begin{aligned} \overrightarrow{s_1} &= \ll 0, 1, 2, \ldots \gg \\ \overrightarrow{s_2} &= \ll a, b, c, \ldots \gg \\ zipSY_2(\overrightarrow{s_1}, \overrightarrow{s_2}) &= \ll (0, a), (1, b), (2, c), \ldots \gg \\ unzipSY_2 \ll (0, a), (1, b), (2, c), \ldots \gg &= (\ll 0, 1, 2, \ldots \gg, \ll a, b, c, \ldots \gg) \end{aligned}$$

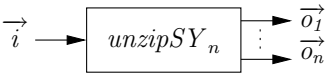
The specification model is a hierarchy of processes and can be viewed as a layered model as illustrated in Figure 3.23.



$zipSY_m(f) = P_S \in \mathbf{P_S}$
where

$$\begin{array}{lcll} \overrightarrow{o} & = & P_S(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}) & \\ T(\overrightarrow{o}, j) & = & T(\overrightarrow{i_1}, j) = \cdots = T(\overrightarrow{i_m}, j) & \forall j \in \mathbb{N}_0 \\ V(\overrightarrow{o}, j) & = & (V(\overrightarrow{i_1}, j), \ldots, V(\overrightarrow{i_m}, j)) & \forall j \in \mathbb{N}_0 \end{array}$$

Figure 3.21. The combinator process $zipSY$



$unzipSY_n(f) = P_S \in \mathbf{P_S}$
where

$$\begin{array}{lcll} (\overrightarrow{o_1}, \ldots, \overrightarrow{o_n}) & = & P_S(\overrightarrow{i}) & \\ V(\overrightarrow{i}, j) & = & (v_j(1), \ldots, v_j(n)) & \forall j \in \mathbb{N}_0 \\ T(\overrightarrow{o_1}, j) & = & \cdots = T(\overrightarrow{o_n}, j) = T(\overrightarrow{i}, j) & \forall j \in \mathbb{N}_0 \\ V(\overrightarrow{o_k}, j) & = & (v_j(k)) & \forall j \in \mathbb{N}_0; \forall k.1 \leq k \leq n \end{array}$$

Figure 3.22. The combinator process $unzipSY$

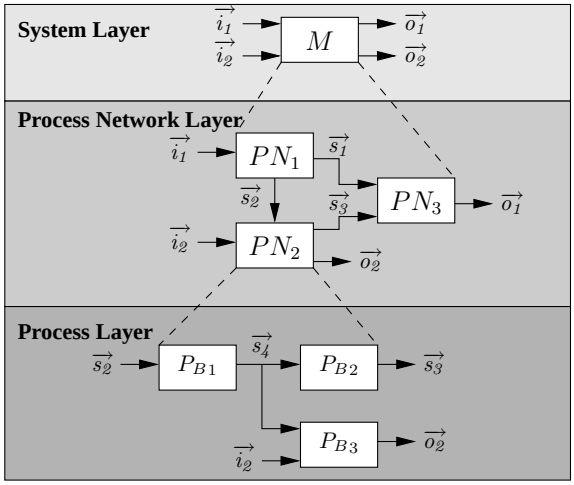


Figure 3.23. The specification model can be viewed as a layered model

The *system layer* contains the top-level process that is a composition of all other processes and defines by its signals the interface to the environment. The system model may, but does not need to have, a number of *process network layers*, where each layer consists of one or more process networks. In Figure 3.23 there is only one process network layer. The *process layer* is the bottom layer and consists of networks of basic processes, which are constructed by process constructors together with their supplied functions and values, and combinator processes. Figure 3.23 shows only the processes for the process network PN_2, the processes for the process networks PN_1 and PN_3 are not part of this figure.

The system layer and the structure of the process network layer of Figure 3.23 are expressed by the following set of equations:

$$
\begin{aligned}
& M(\overrightarrow{i_1}, \overrightarrow{i_2}) = (\overrightarrow{o_1}, \overrightarrow{o_2}) \\
& \text{where} \\
& (\overrightarrow{s_1}, \overrightarrow{s_2}) \;=\; PN_1(\overrightarrow{i_1}) \\
& (\overrightarrow{s_3}, \overrightarrow{o_2}) \;=\; PN_2(\overrightarrow{i_2}) \\
& o_1 \;=\; PN_3(\overrightarrow{s_1}, \overrightarrow{s_3})
\end{aligned}
$$

In the same manner the process PN_2 is expressed by another set of equations:

$$
\begin{aligned}
& PN_2(\overrightarrow{s_2}, \overrightarrow{i_2}) = (\overrightarrow{s_3}, \overrightarrow{o_2}) \\
& \text{where} \\
& \overrightarrow{s_4} \;=\; P_{B1}(\overrightarrow{s_1}) \\
& \overrightarrow{o_2} \;=\; P_{B2}(\overrightarrow{s_4}, \overrightarrow{i_2}) \\
& \overrightarrow{s_3} \;=\; P_{B3}(\overrightarrow{s_4})
\end{aligned}
$$

The processes P_{B1}, P_{B2} and P_{B3} are not explicitly given here but they are defined by process constructors and supplied combinational functions and values.

3.4 The Implementation Model

The implementation model is a product of the refinement process (Chapter 5). In contrast to the specification model, which is a network of concurrent synchronous processes, it may also include processes of other computational models. However, to date the ForSyDe methodology allows in addition to synchronous processes only *domain interfaces*[2] in the implementation model. Domain interfaces are defined in Definition 3.15.

[2] In principle these interfaces should be called multi-rate domain interfaces, but for shortness they are only called domain interfaces in this thesis.

Definition 3.15 (Domain Interface) *A* domain interface P_D *is a periodic process where all input signals* $\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}$ *have a signal rate* r_i*, which is different from the signal rate* r_o *of the output signals* $\overrightarrow{o_1}, \ldots, \overrightarrow{o_n}$*.*

$$
\begin{aligned}
&P(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}) = (\overrightarrow{o_1}, \ldots, \overrightarrow{o_n}) \\
&\text{where} \\
&\quad P_D \in \mathbf{P_D} \\
&\quad R(\overrightarrow{i_j}) = r_i \quad \forall j.1 \leq j \leq m \\
&\quad R(\overrightarrow{o_k}) = r_o \quad \forall k.1 \leq k \leq n \\
&\quad r_i \neq r_o
\end{aligned}
$$

The set of all domain interfaces is denoted as $\mathbf{P_D}$*.*

Domain interfaces are used to establish *synchronous sub-domains*, which comprise a local synchronous process network with a different signal rate. This is illustrated in Figure 3.24.

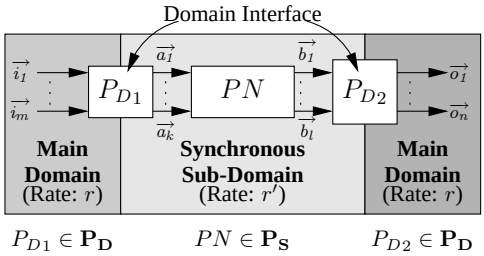


Figure 3.24. Synchronous sub-domains are introduced by domain interfaces

The domain interfaces P_{D1} and P_{D2} establish a synchronous sub-domain. In the main domain all signals have the event rate r, but in the synchronous sub-domain the event rate is r'. Thus domain interfaces introduce an interface between two synchronous process networks with different event rates and establish a multi-rate process network.

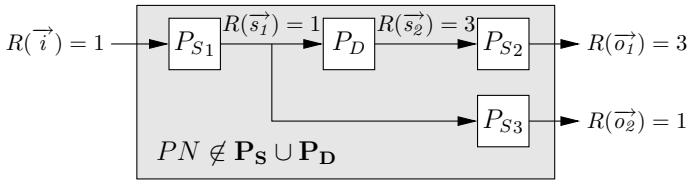


Figure 3.25. A process network with several input and output signal rates

The implementation model allows both synchronous processes and domain interfaces. However the composition of these processes may generate periodic processes that cannot be classified as synchronous process or domain interface as illustrated in Figure 3.25, where the process network PN has two output signals with a different event rate.

This leads to the following definition of an implementation model[3].

Definition 3.16 (Implementation Model) *An implementation model M_I is a process network that is composed of synchronous processes and domain interfaces and models a system.*

$$M_I \in \mathbf{P_P}$$

The set of all implementation models is denoted by $\mathbf{M_I}$.

Synchronous processes and domain interfaces belong to the class of *homogeneous periodic processes* that is defined in Definition 3.17.

Definition 3.17 (Homogeneous Periodic Process) *A* homogeneous periodic process P_H *is a periodic process, where all input signals* $\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}$ *have the rate* r_i *and all output signals* $\overrightarrow{o_1}, \ldots, \overrightarrow{o_n}$ *have the rate* r_o.

$$\begin{aligned} &P_H(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}) = (\overrightarrow{o_1}, \ldots, \overrightarrow{o_n}) \\ &\text{where} \\ &\quad P_H \in \mathbf{P_P} \\ &\quad R(\overrightarrow{i_j}) = r_i \quad \forall j.1 \leq j \leq m \\ &\quad R(\overrightarrow{o_k}) = r_o \quad \forall k.1 \leq k \leq n \end{aligned}$$

The set of all homogeneous periodic processes is denoted by $\mathbf{P_H}$.

Proposition 2 *The set of all homogeneous periodic processes is the union of all possible synchronous processes and domain interfaces.*

$$\mathbf{P_H} = \mathbf{P_S} \cup \mathbf{P_D}$$

Definition 3.18 (System Model) *A system model M is either a specification model or an implementation model.*

$$M \in \mathbf{M_S} \cup \mathbf{M_I}$$

The set of system models is denoted as $\mathbf{M}$.

[3]This definition has to be extended, if additional models of computation are integrated into the ForSyDe methodology

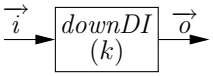
Since synchronous sub-domains violate the synchronous assumption they are not allowed in the specification model, but are introduced by well-defined transformations during the refinement process. Inside a synchronous sub-domain the synchronous assumption is still valid and the same formal techniques can be used as for the initial system model. However, since also domain interfaces have a formal definition, the whole implementation model can be formally analyzed and modified as further elaborated in Chapter 5.

ForSyDe defines process constructors for domain interfaces. These process constructors are called *domain interface constructors* (Definition 3.19).

Definition 3.19 (Domain Interface Constructor) *A* domain interface constructor PC_D *is a function that takes* 0 *to* m *combinational functions* f_i *and* 0 *to* n *values* v_i *and has a domain interface* P_D *as output.*

$$PC_D(f_1, \ldots, f_m, v_1, \ldots, v_n) \quad = \quad P_D \in \mathbf{P_D}$$

The set of domain interface constructors is denoted by $\mathbf{PC_D}$.



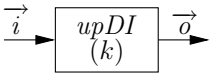
$$
\begin{aligned}
&downDI(k) = P_D \in \mathbf{P_D} \\
&\text{where} \\
&\overrightarrow{o} \quad = \quad P_D(k)(\overrightarrow{i}) \\
&T(\overrightarrow{o}, j) \quad = \quad T(\overrightarrow{i}, kj) \qquad \forall j \in \mathbb{N}_0 \\
&V(\overrightarrow{o}, j) \quad = \quad V(\overrightarrow{i}, kj) \qquad \forall j \in \mathbb{N}_0 \\
&R(\overrightarrow{o}) \quad = \quad R(\overrightarrow{i})/k
\end{aligned}
$$

Figure 3.26. The domain interface constructor $downDI$

The domain interface constructors $downDI$ (Figure 3.26) and $upDI$ (Figure 3.27) generate processes for down- and up-sampling[4]. Figure 3.28 illustrates the functionality of $downDI$ and $upDI$ by a small example.

Note that $upDI(k)$ introduces an output signal with an event rate that is k times higher than the event rate of the input signal by insertion of $k-1$ absent events and not by outputting each value k times. This may look strange at first sight, but has a reason. Assume that the output of $upDI(k)$ is the input of an accumulation

[4]For the definition of domain interface constructors the functions div(m,n) and rem(m,n) are used. The function div(m,n) performs integer division and truncates the result, while rem(m,n) returns the remainder of an integer division. Thus the result of $\text{div}(5,3)$ is 1 and the result of $\text{rem}(5,3)$ is 2.



$upDI(k) = P_D \in \mathbf{P_D}$
where

$$
\begin{array}{lcll}
\overrightarrow{o} & = & P_D(k)(\overrightarrow{i}) & \\
T(\overrightarrow{o}, j) & = & \frac{j}{k} C(\overrightarrow{i}) & \forall j \in \mathbb{N}_0 \\
V(\overrightarrow{o}, j) & = & \begin{cases} V(\overrightarrow{i}, \mathrm{div}(j,k)) & \text{if } \exists u \in \mathbb{N}_0 . j = uk \\ \bot & \text{otherwise} \end{cases} & \forall j \in \mathbb{N}_0 \\
R(\overrightarrow{o}) & = & kR(\overrightarrow{i}) &
\end{array}
$$

Figure 3.27. The domain interface constructor $upDI$

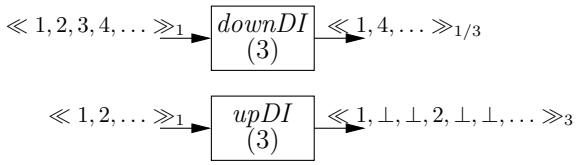


Figure 3.28. Example for down- and up-sampling

process. Then a wrong result will be produced if each value is accumulated k-times. Thus $upDI(k)$ is just up-sampling the event rate, but does not introduce new values. If this is needed a process $holdSY(s_0)$ can be introduced as shown in Figure 3.29. The process $holdSY(s_0)$ is based on $scanlSY$ and outputs for each absent input value the last present input value. In case that the input signal starts with an absent value, the output value is s_0. The usage of $holdSY$ is shown in Figure 3.29.

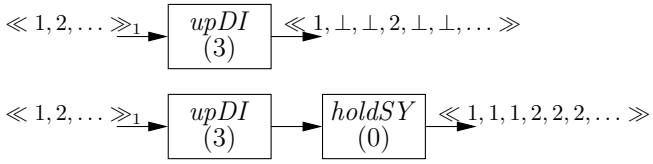
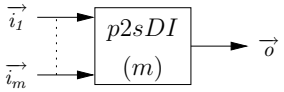


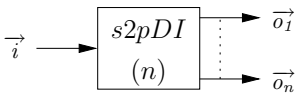
Figure 3.29. Usage of $holdSY$

The domain interface constructor $p2sDI(m)$ (Figure 3.30) generates a process that transforms parallel input signals into a serial signal with a signal rate that is m times higher than the signal rate of the inputs. The domain interface constructor $s2pDI(n)$ (Figure 3.31) generates a process that performs the opposite operation,



$$
\begin{array}{l}
p2sDI(m) = P_D \in \mathbf{P_D} \\
\text{where} \\
\begin{array}{lcll}
\overrightarrow{o} & = & P_D(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}) & \\
T(\overrightarrow{o}, j) & = & \frac{j}{m} C(\overrightarrow{i_1}) = \cdots = \frac{j}{m} C(\overrightarrow{i_m}) & \forall j \in \mathbb{N}_0 \\
V(\overrightarrow{o}, j) & = & V(\overrightarrow{i_u}, w) & \forall j \in \mathbb{N}_0 \\
 & & \text{where } u = \text{rem}(j, m) + 1 & \\
 & & \quad\; w = \text{div}(j, m) & \\
R(\overrightarrow{o}) & = & mR(\overrightarrow{i_1}) = \cdots = mR(\overrightarrow{i_m}) &
\end{array}
\end{array}
$$

Figure 3.30. The domain interface constructor $p2sDI$



$$
\begin{array}{l}
s2pDI(n) = P_D \in \mathbf{P_D} \\
\text{where} \\
\begin{array}{lcll}
(\overrightarrow{o_1}, \ldots, \overrightarrow{o_n}) & = & P_D(\overrightarrow{i}) & \\
T(\overrightarrow{o_k}, j) & = & T(\overrightarrow{i}, nj) & \forall j, k.j \in \mathbb{N}_0 \wedge 1 \leq k \leq n \\
V(\overrightarrow{o_k}, j) & = & \begin{cases} \perp & \text{if } j = 0 \\ V(\overrightarrow{i}, (j-1)n + k - 1) & \text{otherwise} \end{cases} & \\
 & & \forall j, k.j \in \mathbb{N}_0 \wedge 1 \leq k \leq n & \\
R(\overrightarrow{o_k}) & = & R(\overrightarrow{i})/n & \forall k.1 \leq k \leq n
\end{array}
\end{array}
$$

Figure 3.31. The domain interface constructor $s2pDI$

i.e. the transformation of a serial signal into n parallel signals.

Figure 3.32 illustrates that the sequential composition of $p2sDI(k)$ and $s2pDI(k)$ introduces an extra delay. This is because of the fact that all signals have their first value at tag 0 and that the process $s2pDI(n)$ needs n event cycles (of the input signal) to determine all values of all output signals during an output event cycle.

3.5 The ForSyDe Modeling Language

In principle all languages that are able to express ForSyDe's computational models may be used as modeling language for ForSyDe. Since ForSyDe has started

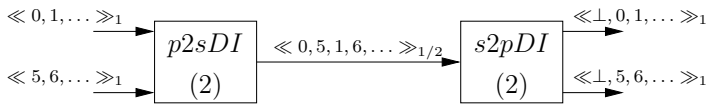


Figure 3.32. The composition of $p2sDI(k)$ and $s2pDI(k)$ introduces an extra delay

from a research perspective, the functional language Haskell [62], which was introduced in Section 2.5 has been selected as modeling language, because it is free from side-effects and supports many of the key concepts of ForSyDe due to concepts like higher-order functions and lazy evaluation. Thus the implementation of signals, process constructors, domain interfaces and the absent extension function Ψ is straight forward and allows to express ForSyDe models in a clean way with minimal effort.

In contrast to Haskell, imperative languages, such as C++, Java or VHDL, do not directly support all concepts of ForSyDe, in particular not the concept of higher-order functions. Hence, a system model expressed in these languages will be not as elegant as a Haskell model.

On the other hand, industrial system designers are used to imperative languages and may have difficulties to accept Haskell as their modeling language. However, for an eventual future industrialization of ForSyDe, there are at least two possible approaches in order to make it more appealing for designers.

- An incorporation of a more accepted modeling language would enable the designer to use ForSyDe without learning a new language paradigm. However the use of that language should be restricted in accordance to the ForSyDe principles. A similar idea has been used successfully in using VHDL for logic synthesis, where the synthesis semantics differ from the simulation semantics. Such an approach would allow to gradually introduce new concepts into industrial design practice.

  It seems that C++ together with System C [38] is a good candidate, since it allows the development of a class library for process constructors. However, in contrast to Haskell, it is weakly typed and not free from side effects. Thus C++ will not aid the designer to the same extent.

- The development of a graphical user interface for ForSyDe would allow to ”hide” Haskell from the designer. The designer would be able to pick process constructors, which may have more intuitive names like $comb_1$ or $comb_2$ instead of $mapSY$ and $zipWithSY$, and only need to formulate the corresponding combinational functions and initial values in order to specify a

process. The specification model can then be developed by drawing signals between processes. Such a GUI could also assist the designer during design refinement, where the tool ideally would highlight possible transformations together with estimation data and the designer selects one of the proposed transformations.

Naturally, both approaches can be combined.

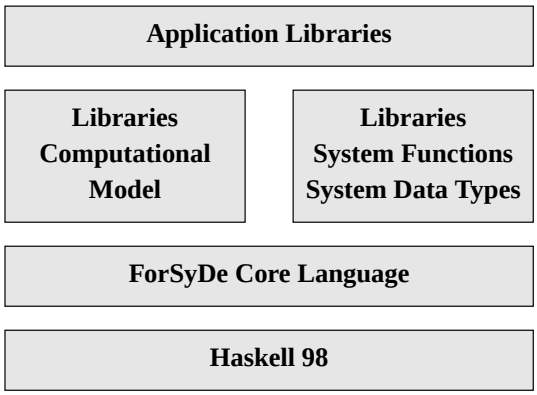


Figure 3.33. The ForSyDe Standard Library

The ForSyDe Standard Library[5] consists of several layers as illustrated in Figure 3.33. The bottom layer is the Haskell 98 language [62]. The layer above Haskell 98 defines the *ForSyDe Core Language*. Here the fundamental data types, such as signal and vector, and the corresponding functions are defined. Computational models are defined in a *Computational Model Libraries* and are located on top of the core language. Also on top of the core language there are the *Libraries of System Functions and Data Types*, which contain functions and data types that are typical for system applications and are independent of the computational model. Examples for such functions are the DFT[6] and FFT[7] (Appendix A.3.3). The top layer of the ForSyDe Standard Library consists of *Application Libraries*. These libraries include components and functions that are modeled for specific computational models, such as a FIR[8]-filter that is modeled for the synchronous computational model (Appendix A.5.2).

[5] Appendix A shows a large part of the ForSyDe Standard Library. This part contains the library for the synchronous and the multi-rate model, which is sufficient to model the examples given in this thesis.

[6] Discrete Fourier Transform

[7] Fast Fourier Transform

[8] Finite Impulse Response

The ForSyDe Standard library is under continuous development. The idea is to keep the ForSyDe core library and the existing computational model libraries stable. The extension of the library is planned to be done by the incorporation of new computational models and new system functions and data types. The version of the ForSyDe Standard Library that is discussed in this thesis is version 2.3, which can be downloaded from the ForSyDe web page [1].

To date, all signals in ForSyDe are modeled with the same data type `Signal`. This data type models signals as a sequence (list) of events, with the exception that there is no more information about the tags of a signal than the order of the tags, which is given by the position in the sequence. However, this is sufficient for the modeling purpose, since information about signal rates is implicitly given by the chosen process constructors and domain interfaces.

The data type `Signal` is defined as

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

This reads as follows. A signal with values of a data type `a` is either empty (`NullS`) or recursively composed by a value of type `a` and a signal of type `a`, e.g.

```
1 :- 2:- 3 :- 4 :- 5 :- NullS
```

is a signal of integers. The output from the Haskell program is given in the short notation

```
{1, 2, 3, 4, 5}
```

for this signal.

In order to be able to model the absent value $\perp$, ForSyDe defines the data type `AbstExt a` that extends a data type A into the extended data type $A_\perp$.

```
data AbstExt a =  Abst
              | Prst a
```

`Abst` denotes the absent value $\perp$ and `Prst a` denotes a value of type A. The extended data type is used for aperiodic signals, e.g. reset signals and signals with a low data rate. The data type

```
Signal (AbstExt Int)
```

models a signal that in addition to integer values also contains absent values. Signals, that have a defined value at each tag, are modeled with regular data types.

The ForSyDe Standard Library defines the function `psi` that implements the higher-order function Ψ ((3.1)) that extends a function of type `a -> b` into a function of data type `AbstExt a -> AbstExt b`.

```
psi f                = f'
  where f' Abst      = Abst
        f' (Prst x)  = Prst (f x)
```

Process constructors are implemented with higher-order functions.

```
mapSY f NullS   = NullS
mapSY f (x:-xs) = f x :- mapSY f xs
```

The higher-order function `mapSY` is defined in the same way as the function `map` of the Haskell "standard prelude". It has two arguments. The first argument is a function f and the the second argument is a signal of any type. The definition of `mapSY` uses pattern matching. The first pattern matches, if the signal is empty (`NullS`). The second pattern matches, if the signal has at least one value, i.e. it is constructed by a head value `x` and a signal tail `xs`. In this case `f` is applied to `x` and the result of this function will be the first value of the output signal. The rest of the output signal is calculated recursively by the function call `mapSY f xs`.

Infinite signals can be generated using the implementation of the process constructor $sourceSY$. The m-to-n counter is part of the application library for the synchronous computational model inside the ForSyDe Standard Library.

```
counterSY m n = sourceSY f m
  where
    f x | x >= n    = m
        | otherwise = succ x
```

The signal definition

```
countSignal = counterSY 0 4
```

defines an infinite signal that repetitively counts from 0 to 4. Although the signal can never be evaluated in total, the definition is very useful, since the laziness of Haskell allows to evaluate parts of the infinite signal as discussed in Section 2.5. To allow for this, the ForSyDe Standard Library defines the function `takeS n` that returns the first n values of a signal and the function `atS` that returns the n-th value (counted from 0) of a signal. In the following a session with the Haskell interpreter Hugs98 is shown to illustrate how possibly infinite signals can be accessed.

```
Hugs98Prompt> takeS 10 countSignal
{0,1,2,3,4,0,1,2,3,4} :: Signal Integer
Hugs98Prompt> atS 4 countSignal
4 :: Integer
```

The ForSyDe Standard Library implements the domain interface constructors $downDI$, $upDI$, $p2sDI$ and $s2pDI$. Their use is illustrated in the following Hugs98 session, where the last example illustrates the composition of $p2sDI(2)$

and $s2pDI(2)$[9] as shown in Figure 3.32. In this session the signals $\overrightarrow{s_1}$ and $\overrightarrow{s_2}$ are defined as

$$\overrightarrow{s_1} = \ll 0, 1, 2, 3, 4 \gg$$
$$\overrightarrow{s_2} = \ll 5, 6, 7, 8, 9 \gg$$

```
Hugs98Prompt> downDI 2 (takeS 10 (counterSY 0 4))
{0,2,4,1,3}
Hugs98Prompt> upDI 2 (takeS 5 (counterSY 0 4))
{0,_,1,_,2,_,3,_,4,_}
Hugs98Prompt> par2ser2DI s1 s2
{0,5,1,6,2,7,3,8,4,9}
Hugs98Prompt> ser2par2DI (par2ser2DI s1 s2)
{(_,_),(0,5),(1,6),(2,7),(3,8),(4,9)}
```

Process networks are modeled either with composition operators of as a set of equations.

The functions for sequential and parallel function composition (Definition 3.5-3.7) are implemented as `funComb1`, `funComb2`, ... and `parComb`. The operation `funComb1` is identical to the Haskell function composition operator '.', which instead for `funComb1` is used throughout this thesis. The process network of Figure 3.1 is expressed as a set of equations.

```
system i1 i2 = (o1, o2)
  where
    o1 = p2 s1 s2
    o2 = s2
    s1 = p1 i1
    s2 = p3 i1 i2
```

3.6 Summary

This chapter defines the specification and implementation model of ForSyDe. The specification model is based on a synchronous computational model and models a system as a hierarchical process network. Processes are constructed by process constructors, which among other benefits implement the synchronous computational model, allow for design transformation (Chapter 5) and which can be given an implementation semantics (Chapter 6).

The implementation model uses domain interfaces together with synchronous processes in order to establish synchronous sub-domains, i.e. a process network

[9] The domain interface constructors $p2sDI(2)$ and $s2pDI(2)$ are implemented as `par2ser2DI` and `ser2par2DI` respectively.

with a different signal rate. Such model is called a multi-rate model. So far ForSyDe uses only a multi-rate model inside the implementation model, but other computational models can be integrated in future.

The ForSyDe modeling language is embedded into the functional language Haskell. The ForSyDe Standard Library provides among others data types, process constructors and domain interfaces, which allow it to express and simulate specification and implementation models using Haskell. Chapter 4 illustrates the development of a ForSyDe specification model by means of a digital equalizer. The full specification model expressed in Haskell is given in Appendix B.

Chapter 4

Development of the Specification Model

The topic of this chapter is the development of a specification model in ForSyDe. The first part discusses the importance of modeling rules. Since ForSyDe models are expressed with the functional language Haskell, modeling rules are needed to ensure that the model complies with the formal definition of the ForSyDe specification model, in particular with the synchronous computational model. The second part illustrates modeling in ForSyDe by means of the specification model of a digital equalizer.

4.1 Modeling in ForSyDe

System design in ForSyDe starts with the development of an abstract and functional system specification model that is based on a synchronous computational model. As discussed in Section 3.5 at present the ForSyDe modeling language is embedded in Haskell. This means that the designer is restricted to a subset of possible Haskell programs. This subset is implicitly defined by the *modeling rules* of ForSyDe. These rules guarantee that a ForSyDe specification model complies to the synchronous computational model and thus can be interpreted as a network of concurrent processes. The most important modeling rule is that the specification model must be expressed as a composition of processes. The composition is either done by composition operators or as a set of equations as discussed in Chapter 3. Since the semantics of ForSyDe define a system model as a function of the

input signals, the functional language paradigm is well suited to express ForSyDe models.

A modeling rule states that processes are either constructed by process constructors or are combinator processes or a composition of other processes. As discussed in Chapter 3 the use of process constructors has a lot of benefits. In particular they implement the computational model and also allow to give an implementation semantics to a process as illustrated in Chapter 6 for the mapping of ForSyDe processes to hardware.

Another important modeling rule defines that process constructors may only be used together with combinational, i.e. stateless, functions, in order to ensure a clear semantics of the model. There are only local states in a ForSyDe model and these states reside inside the processes that are constructed by sequential process constructors.

ForSyDe allows both a top-down and a bottom-up design process. Components in the form of process networks or even system models can be stored in a design library and reused in a bottom-up approach. However, due to the functional characteristic of ForSyDe, where the model is a hierarchy of functions, a top-down process is very natural and will be used for the development of the specification model in the next section.

It should be pointed out that the designer models in the ForSyDe modeling language, which to date is embedded in Haskell. The designer is never confronted directly with the formal definition of ForSyDe (as given in the previous section). However, since ForSyDe is not restricted to Haskell, the following specification model is given using the mathematical notation of ForSyDe. In order to show that there is almost a one-to-one mapping between the formal notation and Haskell, parts of the model are also given in Haskell.

4.2 The Equalizer Specification Model

The development of a specification model is illustrated by means of a digital equalizer. The equalizer has originally been used in [20] to illustrate the MASCOT[1] methodology. MASCOT [21] uses SDL[2] [29] to model control and Matlab[3] [45] to model data flow parts. SDL has been used heavily, in particular by the telecommunication industry, for the specification of control-intensive applications. It is

[1] Matlab and SDL Codesign Techniques
[2] Specification and Description Language
[3] Matrix Laboratory

based on concurrent finite state machines. In contrast Matlab contains very powerful mathematical operations and has been mainly used for the modeling of signal processing algorithms.

This section presents the main parts of the ForSyDe equalizer specification model and uses mainly the formal notation of Chapter 3. The complete executable equalizer specification model, expressed in Haskell, is given in Appendix B.

The discussion of the specification model starts from the top. In general the interfaces to the environment are given by a requirement specification written in a natural language. The task of the equalizer is to modify an audio input signal according to the position of the buttons for the bass and treble levels and to output the modified signal. In addition the equalizer also monitors the output signal in order to prevent damage to the speakers in case of a too high bass level. Figure 4.1 shows the equalizer as a black box together with its environment.

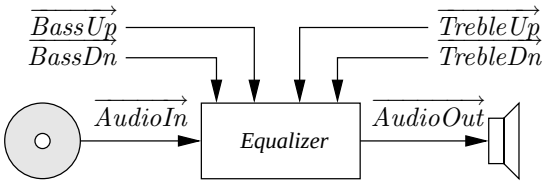


Figure 4.1. The equalizer and its environment

This specification can be naturally decomposed into four functions and a delay process as shown in Figure 4.2.

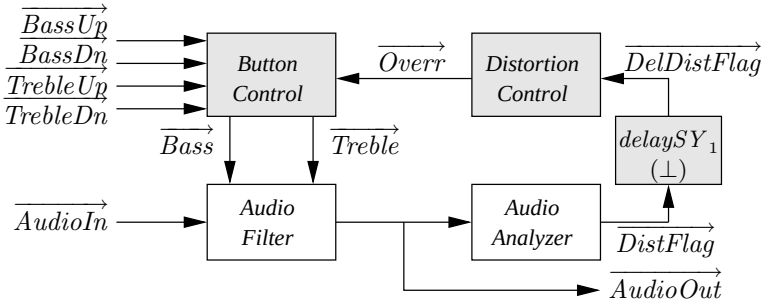


Figure 4.2. Subsystems of the equalizer

The $ButtonControl$ subsystem monitors the position of the button inputs and the override signal from the subsystem $DistortionControl$ and adjusts the current bass and treble levels. This information is then passed to the subsystem $AudioFilter$, which receives the audio input, and filters and amplifies the audio

signal according to the current bass and treble levels. This signal, the output signal of the equalizer, is analyzed by the $AudioAnalyzer$ subsystem, which determines, whether the bass exceeds a predefined threshold. The result of this analysis is passed to the subsystem $DistortionControl$, which decides, if a minor or major violation is encountered and issues the necessary commands to the $ButtonControl$ subsystem. Since the equalizer has an internal feedback loop a delay process has been introduced between the $AudioAnalyzer$ and the $DistortionControl$ in order to give the system a defined start value.

As indicated in Figure 4.2 the equalizer model can be divided into two parts. The data flow part (white boxes) is responsible for the processing of the audio signal. The control part (gray shaded) controls the signal processing according to the input buttons and the output from the $AudioAnalyzer$ and sets the current bass and treble levels. While data flow signals such as $\overrightarrow{AudioIn}$ and $\overrightarrow{AudioOut}$ have a defined value for each event cycle, the control signals $\overrightarrow{BassUp}$, $\overrightarrow{BassDn}$, $\overrightarrow{TrebleUp}$, $\overrightarrow{TrebleDn}$ are aperiodic and not asserted for most of the time. The output of the $AudioAnalyzer$, the signal $DistFlag$, outputs only one valid value for a certain number of input values (depending on the size of the DFT inside the $AudioAnalyzer$) and thus has to be modeled with an extended data type.

Since the specification model is a synchronous model, the absence of valid values is modeled with the special value $\perp$. Thus, there are two types of signals in the equalizer model. The signals $\overrightarrow{AudioIn}$, $\overrightarrow{AudioOut}$, $\overrightarrow{Bass}$ and $\overrightarrow{Treble}$ always have valid values and are modeled as signals with a non-extended data type, in this case $\mathbb{R}$. All other signals will at certain tags not have a valid value and are modeled with extended data types, which include the value $\perp$. A signal with extended data types does not have a present value during all event cycles. This information is very useful for design refinement as elaborated in Chapter 5. Table 4.1 summarizes the data types used for the signals.

Signal	Data Type of Signal Values
$\overrightarrow{AudioIn}$, $\overrightarrow{AudioOut}$, $\overrightarrow{Bass}$, $\overrightarrow{Treble}$	$\mathbb{R}$
$\overrightarrow{BassUp}$, $\overrightarrow{BassDn}$, $\overrightarrow{TrebleUp}$, $\overrightarrow{TrebleDn}$	Extended enumeration type
$\overrightarrow{Overr}$, $\overrightarrow{DistFlag}$, $\overrightarrow{DelDistFlag}$	Extended enumeration type

Table 4.1. Data types for signals

The interaction of the subsystems is modeled by means of a set of equations, where each equation specifies the input and output signals of a subsystem. The formal ForSyDe description is given below.

$$
\begin{aligned}
&Equalizer(\overrightarrow{BassDn}, \overrightarrow{BassUp}, \overrightarrow{TrebleDn}, \overrightarrow{TrebleUp}, \overrightarrow{AudioIn}) \\
&\quad = \overrightarrow{AudioOut} \\
&\text{where} \\
&(\overrightarrow{Bass}, \overrightarrow{Treble}) = ButtonControl(\overrightarrow{BassDn}, \overrightarrow{BassUp}, \overrightarrow{TrebleDn}, \\
&\qquad\qquad \overrightarrow{TrebleUp}, \overrightarrow{Overr}) \\
&\overrightarrow{AudioOut} = AudioFilter(\overrightarrow{Bass}, \overrightarrow{Treble}, \overrightarrow{AudioIn}) \\
&\overrightarrow{Overr} = DistortionControl(\overrightarrow{DelDistFlag}) \\
&\overrightarrow{DistFlag} = AudioAnalyzer(\overrightarrow{AudioOut}) \\
&\overrightarrow{DelDistFlag} = delaySY_1(\perp)(\overrightarrow{DistFlag})
\end{aligned}
$$

Since the equalizer contains a feedback loop there is a need for an initial value in order to stabilize the system. This is also required by the ForSyDe modeling rules, since zero-delay feedback loops are forbidden. This is done by the insertion of a process $delaySY_1(\perp)$, which delays the signal $\overrightarrow{DistFlag}$ one event cycle and produces the value $\perp$ as its first value.

Below follows the corresponding Haskell code for this level of the equalizer model. It is a direct mapping of the formal ForSyDe description. This code does also include the four parameters of the equalizer model, which have been abstracted away in the formal description. The parameters `lpCoeff`, `bpCoeff` and `hpCoeff` give the coefficients for the three FIR-filters that are used to define the characteristics of the low pass, the band pass and the high pass filters in the $AudioFilter$ subsystem. The parameter `dftPts` gives the number of points for the DFT that is used in the $AudioFilter$ subsystem.

```
equalizer :: Vector Double -> Vector Double -> Vector Double -> Integer
          -> Signal (AbstExt Sensor) -> Signal (AbstExt Sensor)
          -> Signal (AbstExt Sensor) -> Signal (AbstExt Sensor)
          -> Signal Double -> Signal Double
equalizer lpCoeff bpCoeff hpCoeff dftPts
          bassUp bassDn trebleUp trebleDn input = output
  where
    (bass, treble)  = buttonControl overrides bassUp bassDn
                                    trebleUp trebleDn
    output          = audioFilter lpCoeff bpCoeff hpCoeff
                                  bass treble input
    distFlag        = audioAnalyzer dftPts output
    overrides       = distortionControl delayedDistFlag
    delayedDistFlag = delaySY Abst distFlag
```

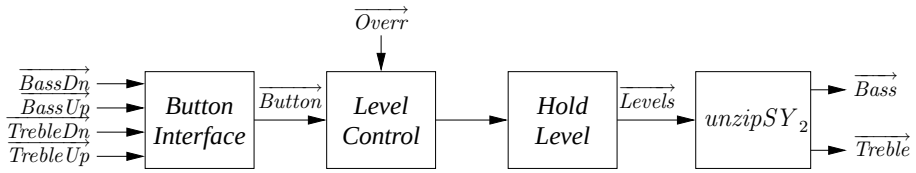


Figure 4.3. The subsystem $ButtonControl$

4.2.1 The Subsystem *ButtonControl*

The subsystem $ButtonControl$ (Figure 4.3) works as a user interface in the equalizer system. It receives the four input signals $\overrightarrow{BassDn}$, $\overrightarrow{BassUp}$, $\overrightarrow{TrebleDn}$, $\overrightarrow{TrebleUp}$ and the override signal $\overrightarrow{Overr}$ from the $DistortionControl$ and calculates the new bass and treble values for the output signals $\overrightarrow{Bass}$ and $\overrightarrow{Treble}$. The subsystem contains the main processes $ButtonInterface$ and $LevelControl$. The process $ButtonInterface$ monitors the four input button signals and outputs the value of the pressed button or the absent value, if no button is pressed during an event cycle. The process $LevelControl$ keeps track of the current bass and treble levels and adjusts them, if either an event from the signal $\overrightarrow{Button}$ or $\overrightarrow{Overr}$ is present. In this case the process outputs the current levels, otherwise the output value is absent. The process $HoldLevel$ holds the value of the current bass and treble levels until a new value is computed. It is modeled by means of the process $holdSY(0.0, 0.0)$ (Figure 3.29) that outputs the last present value, if the input value is absent. The process $unzipSY_2$ transforms a signal of tuples (the current bass and treble level) into a tuple of signals (a bass and a treble signal). The process network from Figure 4.3 is expressed as a set of equations in the formal notation

$$
\begin{aligned}
&ButtonControl(\overrightarrow{Overr}, \overrightarrow{BassDn}, \overrightarrow{BassUp}, \overrightarrow{TrebleDn}, \overrightarrow{TrebleUp}) \\
&\quad = (\overrightarrow{Bass}, \overrightarrow{Treble}) \\
&\text{where} \\
&\quad (\overrightarrow{Bass}, \overrightarrow{Treble}) = unzipSY_2(\overrightarrow{Levels}) \\
&\quad \overrightarrow{Levels} = (HoldLevel \circ_2 LevelControl)(\overrightarrow{Button}, \overrightarrow{Overr}) \\
&\quad \overrightarrow{Button} = ButtonInterface(\overrightarrow{BassDn}, \overrightarrow{BassUp}, \overrightarrow{TrebleDn}, \overrightarrow{TrebleUp})
\end{aligned}
$$

and in Haskell:

```
buttonControl :: Signal (AbstExt OverrideMsg) -> Signal (AbstExt Sensor)
              -> Signal (AbstExt Sensor) -> Signal (AbstExt Sensor)
              -> Signal (AbstExt Sensor) -> (Signal Bass,Signal Treble)
buttonControl overrides bassDn bassUp trebleDn trebleUp
    = (bass, treble)
```

```
where (bass, treble) = unzipSY levels
      levels = ((holdSY (0.0, 0.0)) `funComb2` levelControl)
                    button overrides
      button = buttonInterface bassDn bassUp trebleDn trebleUp
```

The Process $ButtonInterface$

The process $ButtonInterface$ monitors the four input buttons $\overrightarrow{BassDn}$, $\overrightarrow{BassUp}$, $\overrightarrow{TrebleDn}$, $\overrightarrow{TrebleUp}$ and outputs the value for the pressed button $\overrightarrow{Button}$. If two or more buttons are pressed the conflict is resolved by the priority order of the buttons. If no button is pressed the output is absent. Since the process is purely combinational and has four inputs, it is modeled by means of a process that is based on the process constructor $zipWithSY_4$.

$$
\begin{aligned}
&ButtonInterface(\overrightarrow{BassUp}, \overrightarrow{BassDn}, \overrightarrow{TrebleUp}, \overrightarrow{TrebleDn}) \\
&\quad = (zipWithSY_4(f))(\overrightarrow{BassUp}, \overrightarrow{BassDn}, \overrightarrow{TrebleUp}, \overrightarrow{TrebleDn}) \\
&\text{where} \\
&f(\overrightarrow{BassUp}, \overrightarrow{BassDn}, \overrightarrow{TrebleUp}, \overrightarrow{TrebleDn}) \\
&\quad = \begin{cases}
\mathsf{BassUp} & \text{if } bassUp = \mathsf{Active} \\
\mathsf{BassDn} & \text{if } bassUp = \bot \ \wedge \\
& \quad\ bassDn = \mathsf{Active} \\
\mathsf{TrebleUp} & \text{if } bassUp = \bot \ \wedge \\
& \quad\ bassDn = \bot \ \wedge \\
& \quad\ trebleUp = \mathsf{Active} \\
\mathsf{TrebleDn} & \text{if } bassUp = \bot \ \wedge \\
& \quad\ bassDn = \bot \ \wedge \\
& \quad\ trebleUp = \bot \ \wedge \\
& \quad\ trebleDn = \mathsf{Active} \\
\bot & \text{otherwise}
\end{cases}
\end{aligned}
$$

Observe that the use of process constructors simplifies the task for the designer. Since the process constructor (here $zipWithSY$) implements the synchronous computational model, the designer has only to formulate the combinational function (f), which will be applied by the process constructor to all values of the incoming signal.

The $ButtonInterface$ is modeled with help of pattern matching. Instead of writing a large number of `if-then-else` clauses, pattern matching allows to write conditional patterns in a readable and concise way. The value '_' works as a wild card.

```
buttonInterface :: Signal (AbstExt Sensor) -> Signal (AbstExt Sensor)
```

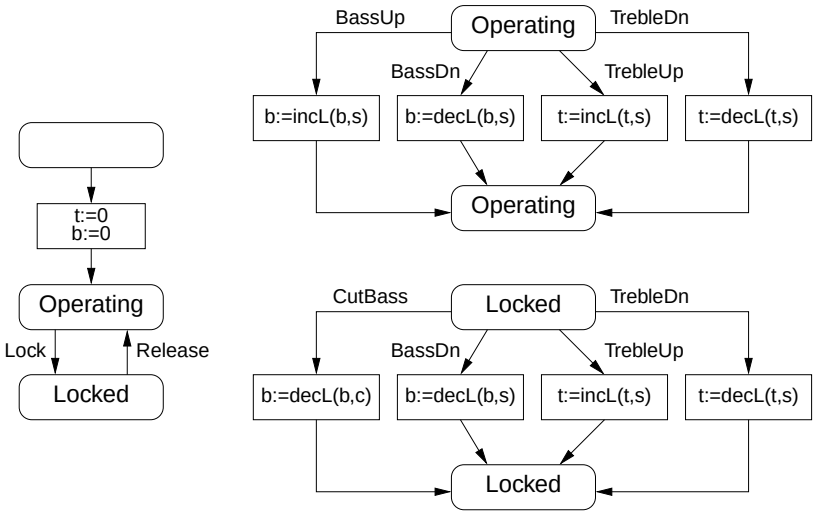


Figure 4.4. The State diagram of the process $LevelControl$

```
                -> Signal (AbstExt Sensor) -> Signal (AbstExt Sensor)
                -> Signal (AbstExt Button)
buttonInterface bassUp bassDn trebleUp trebleDn
   = zipWith4SY f bassUp bassDn trebleUp trebleDn
     where f (Prst Active) _ _ _ = Prst BassUp
           f _ (Prst Active) _ _ = Prst BassDn
           f _ _ (Prst Active) _ = Prst TrebleUp
           f _ _ _ (Prst Active) = Prst TrebleDn
           f _ _ _ _             = Abst
```

The Process *LevelControl*

The state diagram of the process $LevelControl$ is shown in Figure 4.2.1. The functionality has been extended compared to the original MASCOT model. The state diagram is inspired by the original SDL diagram.

The process has a local state expressing the current values for the bass b and treble t. The $LevelControl$ has two modes, in the mode Operating the bass and treble values are stepwise changed in steps of 0.2. However, there exists a maximum and a minimum value of +5.0 and -5.0 respectively. The process enters the mode Locked when the $\overrightarrow{Overr}$ has the value Lock. In this mode an additional increase of the bass level is prohibited and even decreased by 1.0 in case the signal $\overrightarrow{Overr}$ has the value CutBass. The subsystem returns to the mode Operating on the $\overrightarrow{Overr}$ value

Release. The output of the process is an absent extended signal of tuples with the current bass and treble levels.

$$
\begin{array}{l}
LevelControl(\overrightarrow{Button}, \overrightarrow{Overr}) \\
\quad = mealySY_2(nextState, output, (\mathsf{s}_0, \mathsf{b}_0, \mathsf{t}_0))(\overrightarrow{Button}, \overrightarrow{Overr}) \\
\text{where} \\
\quad \mathsf{s}_0 = \mathsf{Operating} \\
\quad \mathsf{b}_0 = 0 \\
\quad \mathsf{t}_0 = 0 \\
\quad nextState((s,b,t), btn, ov) = (s^+, b^+, t^+) \\
\quad \text{where} \\
\qquad s^+ = \begin{cases} \mathsf{Locked} & \text{if } (s = \mathsf{Operating} \wedge ov = \mathsf{Lock}) \vee \\ & \quad (s = \mathsf{Locked} \wedge ov \neq \mathsf{Release}) \\ \mathsf{Operating} & \text{if } (s = \mathsf{Operating} \wedge ov \neq \mathsf{Lock}) \vee \\ & \quad (s = \mathsf{Locked} \wedge ov = \mathsf{Release}) \end{cases} \\
\qquad b^+ = \begin{cases} incL(b, \mathsf{Step}) & \text{if } (s = \mathsf{Operating} \wedge btn = \mathsf{BassUp}) \\ & \quad \wedge (ov \neq \mathsf{CutBass}) \\ decL(b, \mathsf{Step}) & \text{if } (btn = \mathsf{BassDn}) \wedge \\ & \quad (ov \neq \mathsf{CutBass}) \\ decL(b, cutStep) & \text{if } (ov = \mathsf{CutBass}) \end{cases} \\
\qquad t^+ = \begin{cases} incL(t, \mathsf{Step}) & \text{if } btn = \mathsf{TrebleUp} \\ decL(t, \mathsf{Step}) & \text{if } btn = \mathsf{TrebleDn} \end{cases} \\
\qquad \mathsf{Step} = 0.2 \\
\qquad \mathsf{CutStep} = 1.0 \\
\qquad \mathsf{MaxLevel} = 5.0 \\
\qquad \mathsf{MinLevel} = -5.0 \\
\quad incL(x, step) = \begin{cases} \mathsf{MaxLevel} & \text{if } (x + step) \geq \mathsf{MaxLevel} \\ x + step & \text{otherwise} \end{cases} \\
\quad decL(x, step) = \begin{cases} \mathsf{MinLevel} & \text{if } (x - step) \geq \mathsf{MinLevel} \\ x - step & \text{otherwise} \end{cases} \\
output((s,b,t), btn, ov) = \begin{cases} \bot & \text{if } btn = \bot \wedge ov = \bot \\ (b,t) & \text{otherwise} \end{cases}
\end{array}
$$

The process is modeled by means of the process constructor $mealySY_2$, which has a next-state function $nextState$, an output function $output$ and an initial state as arguments. Since the process constructor implements the computational model, the designer has only to formulate the initial state, the next-state and the output function. The state is divided into a mode (with the initial value $s_0 = \mathsf{Operating}$), a bass value ($b_0 = 0$) and a treble value ($t_0 = 0$). The next-state function can be extracted from the state diagram. The output function selects the bass and treble

values from the state in case of a present value of either $\overrightarrow{Button}$ or $\overrightarrow{Overr}$. Otherwise the output event has an absent value.

The Haskell code follows the formal ForSyDe description and can be found in Appendix B.

4.2.2 The Subsystem *DistortionControl*

The *Distortion Control* is directly developed from the SDL-specification that has been used for the MASCOT-model [20]. The specification is shown in Figure 4.5.

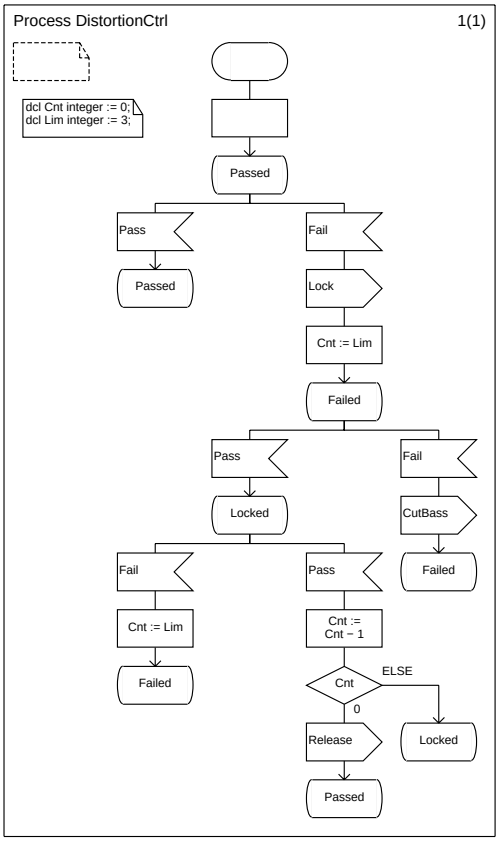


Figure 4.5. SDL-description of the *DistortionControl*

The *DistortionControl* is a single FSM, which can be modeled by means of the process constructor *mealySY*. The global state is not only expressed by the modes - Passed, Failed and Locked -, but also by means of the variable *cnt*. The state machine has two possible input values, Pass and Fail, and three output values,

Lock, Release and CutBass. It takes two functions, ns to calculate the next state and out to calculate the output. The state is represented by a pair of the mode and the variable cnt. The initial state is the same as in the SDL-model, given by the tuple $(\mathsf{Passed}, 0)$. Whenever an input value matches a pattern of the ns function the corresponding right hand side is evaluated, giving the next state. An event with an absent value leaves the state unchanged. The output function is modeled in a similar way. The output is absent, when there is no output message as indicated in the SDL-model.

The ForSyDe model for the $DistortionControl$ is given below.

$$
\begin{aligned}
&DistortionControl = mealySY_1(ns, out, (\mathsf{Passed}, lim))\\
&\text{where}\\
&\quad ns((st, cnt), inp)\\
&\quad = \begin{cases}
(st, cnt) & \text{if } inp = \perp\\
(\mathsf{Passed}, cnt) & \text{if } st = \mathsf{Passed} \wedge inp = \mathsf{Pass}\\
(\mathsf{Failed}, lim) & \text{if } st = \mathsf{Passed} \wedge inp = \mathsf{Fail}\\
(\mathsf{Locked}, cnt) & \text{if } st = \mathsf{Failed} \wedge inp = \mathsf{Pass}\\
(\mathsf{Failed}, cnt) & \text{if } st = \mathsf{Failed} \wedge inp = \mathsf{Fail}\\
(\mathsf{Failed}, lim) & \text{if } st = \mathsf{Locked} \wedge inp = \mathsf{Fail}\\
(\mathsf{Passed}, cnt - 1) & \text{if } st = \mathsf{Locked} \wedge inp = \mathsf{Pass} \wedge cnt = 1\\
(\mathsf{Locked}, cnt - 1) & \text{if } st = \mathsf{Locked} \wedge inp = \mathsf{Pass} \wedge cnt \neq 1
\end{cases}\\
&\quad out((st, cnt), inp)\\
&\quad = \begin{cases}
\mathsf{Lock} & \text{if } st = \mathsf{Passed} \wedge inp = \mathsf{Fail}\\
\mathsf{CutBass} & \text{if } st = \mathsf{Failed} \wedge inp = \mathsf{Fail}\\
\mathsf{Release} & \text{if } st = \mathsf{Locked} \wedge inp = \mathsf{Pass} \wedge cnt = 1\\
\perp & \text{otherwise}
\end{cases}\\
&lim = 3
\end{aligned}
$$

4.2.3 The Subsystem $AudioFilter$

The task of the $AudioFilter$ (Figure 4.6) is to amplify different frequencies of the audio signal independently according to the current bass and treble levels. The audio signal is split into three identical signals, one for each frequency region. The signals are filtered and then amplified according to the assigned amplification level. As the equalizer in this design only has a bass and treble control, the middle frequencies are not amplified. The output signal from the $AudioFilter$ is the addition of the three filtered and amplified signals. This level is also modeled as set of equations.

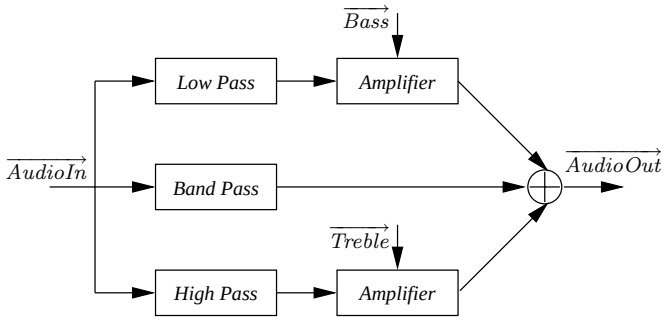


Figure 4.6. Subsystems of the $AudioFilter$

$$AudioFilter(\overrightarrow{Bass}, \overrightarrow{Treble}, \overrightarrow{AudioIn}) = \overrightarrow{AudioOut}$$

where

$$\begin{aligned}
\overrightarrow{AudioOut} &= Sum(\overrightarrow{Low}, \overrightarrow{Middle}, \overrightarrow{High}) \\
\overrightarrow{Low} &= (Amplifer(\overrightarrow{Bass}) \circ LowPass)(\overrightarrow{AudioIn}) \\
\overrightarrow{Middle} &= BandPass(\overrightarrow{AudioIn}) \\
\overrightarrow{High} &= (Amplifer(\overrightarrow{Treble}) \circ BandPass)(\overrightarrow{AudioIn})
\end{aligned}$$

The subsystems of the $AudioFilter$ are implemented as processes. A parametric process FIR that models a FIR-filter, is used to implement all filter functions, i.e. for the low pass, band pass and high pass filter. A FIR-filter is described by the equation

$$y_n = \sum_{m=0}^{k} x_{n-m} h_m$$

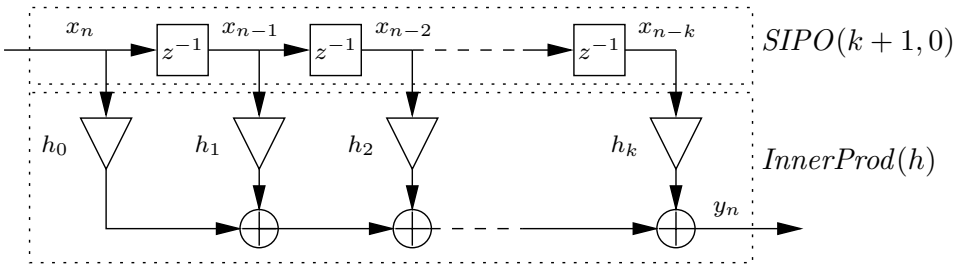


Figure 4.7. FIR-filter

The FIR-filter is modeled as shown in Figure 4.7 as a composition of a shift register with parallel outputs ($SIPO(k+1,0)$) which captures the current state of the filter and a combinational process $InnerProd(h)$ that calculates the inner product of the outputs of the shift register and the coefficient vector h.

$$\begin{aligned} FIR(h) &= InnerProd(h) \circ SIPO(k+1,0) \\ &\text{where } h = \langle h_0, \ldots, h_k \rangle \end{aligned}$$

The shift register $SIPO$ has two parameters for the size n and the initial values s_0 and consists of two parts.

$$SIPO(n, s_0) = unzipSY_n \circ scandSY_1(shiftr, \langle \underbrace{s_0, \ldots, s_0}_{n} \rangle)$$

The sequential process constructor $scandSY_1(shiftr, \langle v, \ldots, v \rangle)$ creates a process, which models a shift register with a vector of size n, where all elements have the initial value s_0. However, according to the definition of the process constructor $scandSY_1$ the output of the shift register is a signal of a vector with n elements, which has to be converted into n parallel signals. This conversion is done be the combinator process $unzipSY_n$. The process $InnerProd(h)$ has the coefficient vector h as parameter. It is modeled with the process constructor $zipWithSY_{k+1}$. The supplied parametric function $ipV(h)$ calculates the inner product of the coefficient vector and the given state vector.

$$\begin{aligned} InnerProd(h) &= zipWithSY_{k+1}(ipV(h)) \\ &\text{where } (ipV(h))(x_0, \ldots, x_n) = h_0 x_0 + \cdots + h_k x_n \end{aligned}$$

The formal FIR-filter description is directly translated to Haskell. Here the functions `zipWithxSY` and `unzipxSY` are Haskell implementations of the process constructor $zipWithSY_m$ and the combinator process $unzipSY_n$ that use vectors of signals of equal type.

```
fir h = innerProd h . sipo k 0.0
    where k = lengthV h

sipo n s0 = unzipxSY . scanldSY shiftrV initState
    where initState = copyV n s0

innerProd h = zipWithxSY (ipV h)
   where ipV NullV   NullV   = 0
         ipV (h:>hv) (x:>xv) = h*x + ipV hv xv
```

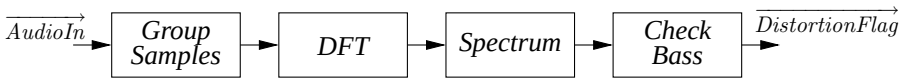


Figure 4.8. The $AudioAnalyzer$ Subsystem

The parametric FIR filter can now be used to express various FIR-filters of different order. The band pass filter in the equalizer is expressed as

$$\begin{aligned} BandPass = FIR([0.063, \quad 0.081, \quad 0.095, \quad 0.104, \quad 0.107 \\ 0.104, \quad 0.095, \quad 0.081, \quad 0.063]) \end{aligned}$$

For other processes in the $AudioFilter$ see Appendix B.4.

4.2.4 The Subsystem $AudioAnalyzer$

The $AudioAnalyzer$ analyzes the current bass level and raises a flag when the bass level exceeds a limit.

As illustrated in Figure 4.8 the $AudioAnalyzer$ is divided into four blocks. The input signal is first grouped into samples of size k by the process $GroupSamples$, since the process DFT operates on a vector of k samples. The process DFT calculates the frequency spectrum of the signal. Then the power spectrum is calculated in $Spectrum$. In $CheckBass$ the lowest frequencies are compared with a threshold value. If they exceed this value, the output $\overrightarrow{DistFlag}$ will have the value Fail.

Since $GroupSamples$ needs k cycles for the grouping, it produces $k-1$ absent values $\perp$ for each grouped sample. Thus the following processes DFT, $Spectrum$ and $CheckBass$ are all Ψ-extended (3.1) in order to be able to process the absent value $\perp$. This means that also the output signal $\overrightarrow{DistFlag}$ has $k-1$ absent values for each present value.

$$\begin{aligned} &AudioAnalyzer(\overrightarrow{AudioOut}) = \overrightarrow{DistFlag} \\ &\text{where} \\ &\quad AudioAnalyzer = \quad CheckBass \circ Spectrum \circ \\ &\qquad\qquad\qquad\qquad DFT \circ GroupSamples \\ &\quad \text{where} \\ &\qquad CheckBass \quad = \quad mapSY(\Psi(checkBass)) \\ &\qquad Spectrum \quad = \quad mapSY(\Psi(spectrum)) \\ &\qquad DFT \quad = \quad mapSY(\Psi(dft(k))) \\ &\qquad GroupSamples \quad = \quad groupSY(k) \end{aligned}$$

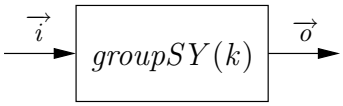
The process $GroupSamples$ is expressed by the library process $groupSY(k)$. The process is based on $mooreSY$ and defined as follows.

$$
\begin{array}{l}
groupSY(k) = P_S \in \mathbf{P_S} \\
\text{where} \\
\quad \overrightarrow{o} = P_S(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}) \\
\quad P_S = mooreSY_1(f, g, s_0) \\
\qquad \text{where} \\
\qquad s_0 = \langle\rangle \\
\qquad f(x,s) = \begin{cases} \langle x \rangle & \text{if } \#s = k \vee \#s = 0 \\ s \oplus x & \text{otherwise} \end{cases} \\
\qquad g(s) = \begin{cases} s & \text{if } \#s = k \\ \bot & \text{otherwise} \end{cases}
\end{array}
\tag{4.1}
$$

The concatination operator $\oplus$ is defined as

$$
\begin{array}{rcl}
\langle\rangle \oplus x & = & \langle x \rangle \\
\langle v_1, \ldots, v_n \rangle \oplus x & = & \langle v_1, \ldots, v_n, x \rangle
\end{array}
$$

and $\#v$ returns the number of elements in the vector v.



$$\overrightarrow{i} =< v_0, v_1, \cdots >$$

$$\overrightarrow{o} =< \langle\rangle, \underbrace{\bot, \ldots, \bot}_{k-1}, \langle v_0, \ldots, v_{k-1} \rangle, \bot, \cdots >$$

Figure 4.9. The process $groupSY$

The process $groupSY(k)$ is illustrated in Figure 4.9. Each input value is read and stored in the internal state vector of $groupSY$ until the state vector has k values. At that point the state vector is written to the output and afterwards reset to an empty vector. Since $groupSY$ is a synchronous process, absent values have to be produced for each input value as long as the grouping is not completed. The process starts with the empty vector as initial state, which is also sent to the output.

For other processes in the $AudioAnalyzer$ see Appendix B.5.

4.3 Discussion

This chapter illustrated modeling in ForSyDe by the example of a specification model of a digital equalizer. The ForSyDe modeling technique leads to very compact[4], well-structured models, which may have many levels of hierarchy. Process networks are described as sets of equations, which capture the structure of a system in a much cleaner way than netlists in VHDL. Control is modeled by means of sequential process constructors that result in processes that can be directly implemented as finite state machines. The functional character of ForSyDe allows an efficient modeling of data flow parts.

The specification model shows how important absent values are in modeling. Absent values are used in several signals to indicate that there is not a present value in all event cycles, which is also reflected in the data type of the signal. This information can be used during design refinement in order to obtain a more efficient implementation as elaborated in Chapter 5.

The example showed also, that there is a direct correspondence between the formal notation of ForSyDe and Haskell. Haskell's powerful type system gives additional benefits to system design. Haskell programs are type-safe and since the type system allows type inference, functions, process constructors and processes can be formulated in a flexible way that facilitates the development of a design library. A good example for the potential of ForSyDe is the process FIR that is modeled with very few lines of code, is parametric for any number of coefficients and has a direct interpretation in hardware. Thus ForSyDe gives support for a combined top-down and bottom-up modeling process, which leads to abstract models that can be refined during transformational design refinement as discussed in Chapter 5.

In principle there is also the possibility to formulate ForSyDe models on a less abstract level, since the implementation model also belongs to the functional domain. Such a low-level model could then without transformation directly be mapped into an implementation. This may be useful for application areas for which not enough transformations have been developed. These low-level models could then be combined with abstract models of other application areas, for which a sufficient amount of transformations exists. However, the intention of ForSyDe is to start with an abstract specification model with a large design space, but ForSyDe also allows to start from lower abstraction levels.

[4] A Haskell program is considerably shorter than a corresponding VHDL description [50].

Chapter 5

Design Refinement

This chapter presents transformational design refinement, which is one of the key concepts in ForSyDe. The objective is to refine an abstract specification model by the application of formally defined transformation rules into a detailed implementation model. The transformational approach is based on the formal definition of ForSyDe and in particular the concept of process constructors. Transformation rules are classified as semantic preserving and design decision. Each rule is accompanied with an implication that indicates the semantical changes caused by a transformation. The first part of this chapter gives the foundations for transformational design refinement in ForSyDe, in particular it is shown how a characteristic function can be derived for all process networks in ForSyDe. In addition semantic preserving transformations and design decisions are introduced by illustrative examples. The second part uses the model of the digital equalizer to illustrate some powerful design transformations, such as clock domain and communication refinement.

5.1 Transformational Design Refinement

One of the key ideas of the ForSyDe methodology is to move large parts of the synthesis, which traditionally are part of the implementation domain, into the functional domain. This is done in the refinement phase where the specification model M_0 is stepwise refined by well defined design transformations T_i into a final implementation model M_n (Figure 5.1). Only at this late stage of the design process the implementation model is translated using the ForSyDe hardware and software semantics into a synthesizable implementation description.

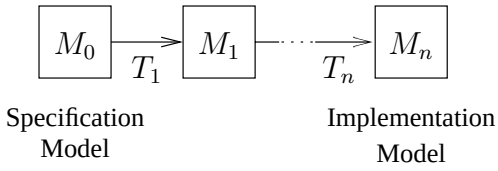


Figure 5.1. Transformational design refinement

A *transformation* (Definition 5.2) is the application of a *transformation rule* (Definition 5.1) to a process network that is a part of a system model as illustrated in Figure 5.2.

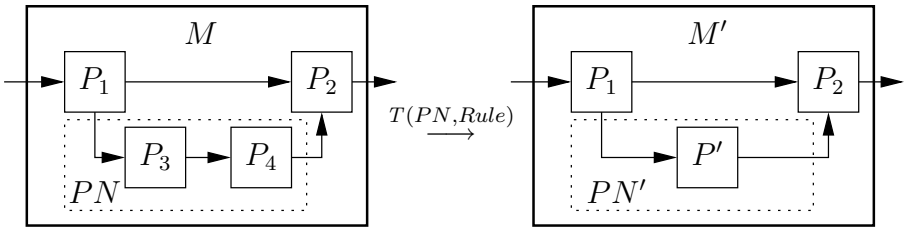


Figure 5.2. Design transformation

Here, the transformation rule *Rule* is applied to the process network PN inside the system model M. The result of the transformation is the system model M'. The only difference between M and M' is the replacement of PN by PN'.

Definition 5.1 (Transformation Rule) *A* transformation rule $R : \mathbf{P} \rightarrow \mathbf{P}$ *is a functional mapping of a process network PN onto another process network PN' with the same number of input signals and the same number of output signals. A transformation rule is denoted by $R(PN) = PN'$ or $PN \xrightarrow{R} PN'$.*

The set of all transformation rules is denoted by $\mathbf{R}$.

Definition 5.2 (Transformation) *A* transformation $T : (\mathbf{M}, \mathbf{P}, \mathbf{R}) \rightarrow \mathbf{M}$ *is a functional mapping of a system model M onto another system model M' with the same input signals and the same number of output signals. Using the transformation rule R the internal process network PN in M is replaced by $R(PN)$ to yield M'. A transformation is denoted by $T(M, PN, R) = M' = M[R(PN)/PN]$ or $M \xrightarrow{T(PN,R)} M' = M[R(PN)/PN]$, where $[x/y]$ reads as y is replaced by x.*

The set of all transformations is denoted by $\mathbf{T}$.

In order to be able to compare process networks and to develop and classify transformation rules, ForSyDe defines the characteristic function for basic processes in Definition 5.3 and extends it to processes in general in Definition 5.4.

Definition 5.3 (Characteristic Function of a Basic Process) *The* characteristic function $\mathcal{E}_{P_B}(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}, j)$ *of a basic process* P_B *with the input signals* $\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}$ *and the output signal* $\overrightarrow{o}$ *expresses the dependence of any output event with index* $j \in \mathbb{N}_0$ *on the input signals.*

$$\begin{aligned}\mathcal{E}_{P_B}(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}, j) &= E(\overrightarrow{o}, j) = E(P_B(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}), j) \\ &= (T(P_B(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}), j), V(P_B(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}), j)) \\ &= (\mathcal{T}_{P_B}(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}, j), \mathcal{V}_{P_B}(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}, j))\end{aligned}$$

$\mathcal{T}_{P_B}$ *is the* characteristic tag function *and* $\mathcal{V}_{P_B}$ *is the* characteristic value function *of a process* P_B.

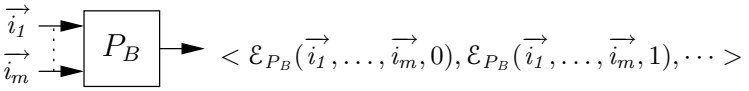


Figure 5.3. A process is completely defined by its characteristic function

The characteristic function $\mathcal{E}_{P_B}$ is sufficient to specify the behavior of the basic process P_B with the input signals $\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}$ by means of

$$\begin{aligned}\overrightarrow{o} &= P_B(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}) \\ &= < \mathcal{E}_{P_B}(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}, 0), \mathcal{E}_{P_B}(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}, 1), \cdots >\end{aligned}$$

as illustrated in Figure 5.3. Since any process can be expressed as a network of basic processes as stated in Proposition 1, all processes can be completely defined by their characteristic function, if there exists a characteristic function for each basic process.

Definition 5.4 (Characteristic Function of a Process) *The* characteristic function $\mathcal{E}_P(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}, j)$ *of a process* P *with the input signals* $\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}$ *and the output signals* $\overrightarrow{o_1}, \ldots, \overrightarrow{o_n}$ *expresses the dependence of the output events at index* $j \in \mathbb{N}_0$ *on the input signals.*

$$\mathcal{E}_P(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j) = (\mathcal{E}_{P_{B\,1}}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j),\ldots,\mathcal{E}_{P_{B\,n}}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j))$$

where

$$\begin{aligned}
(\overrightarrow{o_1},\ldots,\overrightarrow{o_n}) &= P(\overrightarrow{i_1},\ldots,\overrightarrow{i_m}) \\
\overrightarrow{o_1} &= P_{B1}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m}) \\
\overrightarrow{o_2} &= P_{B2}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m}) \\
&\vdots \\
\overrightarrow{o_n} &= P_{Bn}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m})
\end{aligned}$$

As discussed later in this chapter, a characteristic function can be derived for each process constructed by a synchronous process constructor or domain interface constructor and for each combinator process. It is also possible to give a characteristic function for compositions of these processes. The characteristic function can be given for all process networks and system models in ForSyDe that do not include a zero-delay feedback loop and have at least one input signal. Thus a characteristic function for a pure source process cannot be determined.

The characteristic function is sufficient to fully describe the behavior of a process. In the following basic processes are classified as causal, synchronous, combinational or sequential. These classifications can be extended to cover processes with more than one output signal.

- A basic process P_B with the input signals $\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}$ is *causal*, if and only if there exists a function $f(\overrightarrow{i_1},\ldots,\overrightarrow{i_m})$, such that

$$\begin{aligned}
\mathcal{V}_{P_B}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j) = f(V(\overrightarrow{i_1},0),\ldots,V(\overrightarrow{i_1},k_1),\ldots, \\
V(\overrightarrow{i_m},0),\ldots,V(\overrightarrow{i_m},k_m))
\end{aligned}$$

  where $\forall l.1 \leq l \leq m$ and $\forall j, k_l \in \mathbb{N}_0$

$$T(\overrightarrow{i_l},k_l) \leq \mathcal{T}_{P_B}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j)$$

  Thus a process is causal, if all output events only depend on input events that do not have a larger tag than the produced output event. ForSyDe considers only causal systems.

- A basic process P_B with the input signals $\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}$ is *synchronous*, if and only if $\forall j \in \mathbb{N}_0$

$$\mathcal{T}_{P_B}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j) = T(\overrightarrow{i_1},j) = \cdots = T(\overrightarrow{i_m},j)$$

- A basic process P_B with the input signals $\overrightarrow{i_1}$, ..., $\overrightarrow{i_m}$ is *combinational*, if it is synchronous and $\forall j \in \mathbb{N}_0$

$$\mathcal{V}_{P_B}(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}, j) = f(V(\overrightarrow{i_1}, j), \ldots, V(\overrightarrow{i_m}, j))$$

- A basic process P_B with the input signals $\overrightarrow{i_1}$, ..., $\overrightarrow{i_m}$ is *sequential*, if and only if it is synchronous and $\forall j \in \mathbb{N}_0$

$$\exists k \in \mathbb{N}_1 . \mathcal{V}_{P_B}(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}, j) = f(\ldots, V(\overrightarrow{i_1}, j-k), \ldots)$$

In contrast to other transformational approaches for the design of software [88] [89], which only allow the application of *semantic preserving* transformation rules, ForSyDe also defines *design decision* transformation rules, which change the semantics of the model in a defined manner. Design decisions are mainly motivated by the application domain of ForSyDe. ForSyDe targets embedded systems, which often have to fulfill hard requirements on performance or power dissipation. To fulfill these requirements the resources of the system have to be used efficiently. The purpose of the specification model is to model a system on a level that abstracts from implementation details. Thus the ForSyDe specification model uses ideal (infinite) buffers, floating-point numbers and a simple synchronous communication scheme. On the other hand an efficient implementation has to use finite buffers and fixed-point numbers and may have to use versatile communication mechanisms for the synchronization of different components in a distributed implementation. Clearly it is not possible to transform a high-level specification model into such an implementation without changing the semantics. Thus design decisions are needed to bridge the abstraction gap.

The application of a semantic preserving transformation rule does not change the meaning of the system model and is mainly used to optimize the model for synthesis. In contrast, the application of a design decision rule changes the meaning of a system model. A typical design decision is the refinement of an infinite buffer into a fixed-size buffer with a size of n elements. While such a design decision clearly modifies the semantics of the system model, the transformed model may still behave in the same way as the original model under a certain precondition. For instance, if it is possible to prove, that the ideal buffer will never contain more than n elements, it can be replaced by a finite buffer of size n and will behave as the ideal buffer for the given precondition.

Design transformation rules can be classified as *semantic preserving* (Definition 5.5) or *design decision* (Definition 5.6).

Definition 5.5 (Semantic Preserving Transformation Rule) *A transformation rule* $PN \xrightarrow{R} PN'$ *is* semantic preserving rule, *if and only if*

$$\mathcal{E}_{PN}(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}, j) = \mathcal{E}_{PN'}(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}, j).$$

A design decision changes the semantics of the design only to a small extent, i.e. there is a close relation between the original design and the transformed design. This is reflected in Definition 5.6, where such a relation is required.

Definition 5.6 (Design Decision Rule) *A transformation rule* $PN \xrightarrow{R} PN'$ *is a* design decision rule, *if it is not semantic preserving*

$$\mathcal{E}_{PN}(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}, j) \neq \mathcal{E}_{PN'}(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}, j).$$

and there exists a relation rel *between* PN *and* PN'.

$$PN \overset{rel}{\leftrightarrow} PN'$$

These relations are in practice often difficult to express and it is up to the creator of a transformation rule to ensure that there exists a meaningful relation. Examples for close relations between an original and a transformed process network are functional equivalence (Definition 5.7) and tail equivalence (Definition 5.8), which are discussed in Section 5.3.

The rest of this chapter is structured as follows. Section 5.2 gives the characteristic functions for processes constructed by process constructors and process networks. Section 5.3 shows how the characteristic function is used for the development and classification of transformation rules. Finally, Section 5.4 illustrates the potential of transformational design refinement by the example of the digital equalizer, where powerful design transformations are applied.

5.2 Characteristic Functions for Processes and Process Networks

In this section the characteristic function is given for processes based on synchronous process constructors, domain interface constructors, combinator processes and process networks.

5.2.1 Processes based on Synchronous Process Constructors

Synchronous processes output the events at the same tag as the corresponding input event. Thus for a synchronous process, the characteristic tag function yields the tag of the corresponding input event for a given index j.

Combinational processes, such as processes based on $mapSY$ and $zipWithSY$ have a characteristic function that only depends on current input events.

The characteristic function of a process based on $mapSY$ (Figure3.12) is

$$\begin{array}{l} \mathcal{E}_{\diamond(f)}(\overrightarrow{i},j) = (\mathcal{T}_{\diamond(f)}(\overrightarrow{i},j), \mathcal{V}_{\diamond(f)}(\overrightarrow{i},j)) \\ \text{where} \\ \quad \begin{array}{lcl} \mathcal{T}_{\diamond(f)}(\overrightarrow{i},j) & = & T(\overrightarrow{i},j) \\ \mathcal{V}_{\diamond(f)}(\overrightarrow{i},j) & = & f(V(\overrightarrow{i},j)) \end{array} \end{array} \tag{5.1}$$

The characteristic function of a process based on $zipWithSY_m$ (Figure 3.13) is

$$\begin{array}{l} \mathcal{E}_{\triangleright_m(f)}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j) = (\mathcal{T}_{\triangleright_m(f)}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j), \mathcal{V}_{\triangleright_m(f)}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j)) \\ \text{where} \\ \quad \begin{array}{lcl} \mathcal{T}_{\triangleright_m(f)}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j) & = & T(\overrightarrow{i_1},j) = \cdots = T(\overrightarrow{i_m},j) \\ \mathcal{V}_{\diamond(f)}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j) & = & f(V(\overrightarrow{i_1},j),\ldots,V(\overrightarrow{i_m},j)) \end{array} \end{array}$$

Sequential processes, such as processes based on $delaySY$, depend also on past input events.

The characteristic function of a process based on $delaySY_k(s_0)$ (Figure 3.14) is

$$\begin{array}{l} \mathcal{E}_{\triangle_k(s_0)}(\overrightarrow{i},j) = (\mathcal{T}_{\triangle_k(s_0)}(\overrightarrow{i},j), \mathcal{V}_{\triangle_k(s_0)}(\overrightarrow{i},j)) \\ \text{where} \\ \quad \begin{array}{lcl} \mathcal{T}_{\triangle_k(s_0)}(\overrightarrow{i},j) & = & T(\overrightarrow{i},j) \\ \mathcal{V}_{\triangle_k(s_0)}(\overrightarrow{i},j) & = & \left\{ \begin{array}{ll} s_0 & \text{if } j < k \\ V(\overrightarrow{i},j-k) & \text{otherwise} \end{array} \right. \end{array} \end{array}$$

The characteristic function of a process based on $scanlSY_m(f,s_0)$ (Figure 3.14) is recursively expressed by

$$\mathcal{E}_{scanlSY_m(f,s_0)}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j) = (\mathcal{T}_{scanlSY_m(f,s_0)}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j), \mathcal{V}_{scanlSY_m(f,s_0)}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j))$$

where

$$\mathcal{T}_{scanlSY_m(f,s_0)}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j) = T(\overrightarrow{i_1},j) = \cdots = T(\overrightarrow{i_m},j)$$

$$\mathcal{V}_{scanlSY_m(f,s_0)}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j) = f(V(\overrightarrow{i_1},j),\ldots,V(\overrightarrow{i_m},j),s_j)$$

where

$$s_j = \begin{cases} s_0 & \text{if } j = 0 \\ f(V(\overrightarrow{i_1},j-1),\ldots,V(\overrightarrow{i_m},j-1),s_{j-1}) & \text{if } j > 0 \end{cases}$$

The first elements of the characteristic value function of $scanlSY_m(f,s_0)$ can be explicitly given by

$$\mathcal{V}_{scanlSY_m(f,s_0)}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j) = \begin{cases} f(V(\overrightarrow{i_1},0),\ldots,V(\overrightarrow{i_m},0),s_0) & \text{if } j = 0 \\ f(V(\overrightarrow{i_1},1),\ldots,V(\overrightarrow{i_m},1),f(V(\overrightarrow{i_1},0),\ldots,V(\overrightarrow{i_m},0),s_0)) & \text{if } j = 1 \\ \vdots & \vdots \end{cases}$$

This example shows that a characteristic function has increasing complexity with increasing sequential depth.

The characteristic function of a process based on $scanldSY_m(f,s_0)$ (Figure 3.18) is recursively expressed by

$$\mathcal{E}_{scanldSY_m(f,s_0)}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j) = (\mathcal{T}_{scanldSY_m(f,s_0)}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j), \mathcal{V}_{scanldSY_m(f,s_0)}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j))$$

where

$$\mathcal{T}_{scanldSY_m(f,s_0)}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j) = T(\overrightarrow{i_1},j) = \cdots = T(\overrightarrow{i_m},j)$$

$$\mathcal{V}_{scanldSY_m(f,s_0)}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j) = s_j$$

where

$$s_j = \begin{cases} s_0 & \text{if } j = 0 \\ f(V(\overrightarrow{i_1},j-1),\ldots,V(\overrightarrow{i_m},j-1),s_{j-1}) & \text{if } j > 0 \end{cases}$$

The characteristic function of a process based on $mooreSY_m(f,g,s_0)$ (Figure 3.19) is recursively expressed by

$$\mathcal{E}_{mooreSY_m(f,g,s_0)}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j) = (\mathcal{T}_{mooreSY_m(f,g,s_0)}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j), \mathcal{V}_{mooreSY_m(f,g,s_0)}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j))$$

where

$$\mathcal{T}_{mooreSY_m(f,g,s_0)}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j) = T(\overrightarrow{i_1},j) = \cdots = T(\overrightarrow{i_m},j)$$
$$\mathcal{V}_{mooreSY_m(f,g,s_0)}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j) = g(s_0)$$

where

$$s_j = \begin{cases} s_0 & \text{if } j = 0 \\ f(V(\overrightarrow{i_1},j-1),\ldots,V(\overrightarrow{i_m},j-1),s_{j-1}) & \text{if } j > 0 \end{cases}$$

The characteristic function of processes based on $mealySY_m(f,g,s_0)$ (Figure 3.20) is recursively expressed by

$$\mathcal{E}_{mealySY_m(f,g,s_0)}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j) = (\mathcal{T}_{mealySY_m(f,g,s_0)}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j), \mathcal{V}_{mealySY_m(f,g,s_0)}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j))$$

where

$$\mathcal{T}_{mealySY_m(f,g,s_0)}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j) = T(\overrightarrow{i_1},j) = \cdots = T(\overrightarrow{i_m},j)$$
$$\mathcal{V}_{mealySY_m(f,g,s_0)}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j) = g(V(\overrightarrow{i_1},j),\ldots,V(\overrightarrow{i_m},j),s_j)$$

where

$$s_j = \begin{cases} s_0 & \text{if } j = 0 \\ f(V(\overrightarrow{i_1},j-1),\ldots,V(\overrightarrow{i_m},j-1),s_{j-1}) & \text{if } j > 0 \end{cases}$$

5.2.2 Processes based on Domain Interface Constructors

The characteristic function of the domain interface constructors is given below, where 'div' is the integer division function and 'rem' the remainder function. Remember that $C(\overrightarrow{s})$ is the event cycle of a periodic signal $\overrightarrow{s}$ (Definition 3.8).

The characteristic function of the domain interface constructor $downDI$ (Figure 3.26) is

$$\mathcal{E}_{downDI(k)}(\overrightarrow{i},j) = (\mathcal{T}_{downDI(k)}(\overrightarrow{i},j), \mathcal{V}_{downDI(k)}(\overrightarrow{i},j))$$

where

$$\mathcal{T}_{downDI(k)}(\overrightarrow{i},j) = T(\overrightarrow{i},kj)$$
$$\mathcal{V}_{downDI(k)}(\overrightarrow{i},j) = V(\overrightarrow{i},kj)$$

The characteristic function of the domain interface constructor $upDI$ (Figure 3.27) is

$$\mathcal{E}_{upDI(k)}(\overrightarrow{i}, j) = (\mathcal{T}_{upDI(k)}(\overrightarrow{i}, j), \mathcal{V}_{upDI(k)}(\overrightarrow{i}, j))$$
where
$$\begin{aligned} \mathcal{T}_{upDI(k)}(\overrightarrow{i}, j) &= \tfrac{j}{k} C(\overrightarrow{i}) \\ \mathcal{V}_{upDI(k)}(\overrightarrow{i}, j) &= V(\overrightarrow{i}, \text{div}(j,k)) \end{aligned}$$

The characteristic function of the domain interface constructor $p2sDI$ (Figure 3.30) is

$$\mathcal{E}_{p2sDI(m)}(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}, j) = (\mathcal{T}_{p2sDI(m)}(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}, j), \mathcal{V}_{p2sDI(m)}(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}, j))$$
where
$$\begin{aligned} \mathcal{T}_{p2sDI(m)}(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}, j) &= \tfrac{j}{m} C(\overrightarrow{i_1}) = \cdots = \tfrac{j}{m} C(\overrightarrow{i_m}) \\ \mathcal{V}_{p2sDI(m)}(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}, j) &= V(\overrightarrow{i_u}, w) \\ &\quad \text{where} \\ &\quad\quad u = \text{rem}(j,m) + 1 \\ &\quad\quad w = \text{div}(j,m) \end{aligned}$$

The characteristic function of the domain interface constructor $s2pDI$ (Figure 3.31) is

$$\mathcal{E}_{s2pDI(n)}(\overrightarrow{i}, j) = (\mathcal{E}_{P_1}(\overrightarrow{i}, j), \ldots, \mathcal{E}_{P_n}(\overrightarrow{i}, j))$$
where $\forall k. 1 \leq k \leq n$
$$\begin{aligned} \mathcal{E}_{P_k}(\overrightarrow{i}, j) &= (\mathcal{T}_{P_k}(\overrightarrow{i}, j), \mathcal{V}_{P_k}(\overrightarrow{i}, j)) \\ \mathcal{T}_{P_k}(\overrightarrow{i}, j) &= T(\overrightarrow{i}, jn) \\ \mathcal{V}_{P_k}(\overrightarrow{i}, j) &= \begin{cases} \perp & \text{if } j = 0 \\ V(\overrightarrow{i}, (j-1)n + k - 1) & \text{otherwise} \end{cases} \end{aligned}$$

5.2.3 Combinator Processes

The characteristic function of the combinator processes $zipSY_m$ (Figure 3.21) is

$$\mathcal{E}_{zipSY_m}(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}, j) = (\mathcal{T}_{zipSY_m}(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}, j), \mathcal{V}_{zipSY_m}(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}, j))$$
where
$$\begin{aligned} \mathcal{T}_{zipSY_m}(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}, j) &= T(\overrightarrow{i_1}) = \cdots = T(\overrightarrow{i_m}) \\ \mathcal{V}_{zipSY_m}(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}, j) &= (V(\overrightarrow{i_1}, j), \ldots, V(\overrightarrow{i_m}, j)) \end{aligned}$$

The characteristic function of the combinator processes $unzipSY_n$ (Figure 3.22) is

$$\mathcal{E}_{unzipSY_n}(\overrightarrow{i}, j) = (\mathcal{E}_{P_{B_1}}(\overrightarrow{i}, j), \ldots, \mathcal{E}_{P_{B_n}}(\overrightarrow{i}, j))$$

where

$$\begin{array}{lcll} V(\overrightarrow{i}, j) & = & (v_j(1), \ldots, v_j(n)) & \\ \mathcal{T}_{P_{B_k}}(\overrightarrow{i}, j) & = & T(\overrightarrow{i}, j) & \forall k.1 \leq k \leq n \\ \mathcal{V}_{P_{B_k}}(\overrightarrow{i}, j) & = & v_j(k) & \forall k.1 \leq k \leq n \end{array}$$

5.2.4 Network of Processes

The characteristic function of a process network $P_1 \circ_m P_2$ composed by sequential composition can be determined by

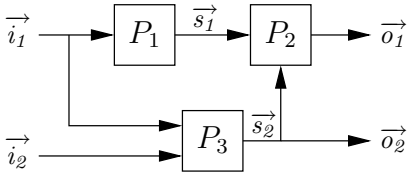
$$\mathcal{E}_{P_1 \circ_m P_2}(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}, j) = \mathcal{E}_{P_1}(\bigoplus_{k=0}^{\infty} \mathcal{E}_{P_2}(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}, k), j)$$

since the output from the process P_2 is $\bigoplus_{k=0}^{\infty} \mathcal{E}_{P_2}(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}, k)$.

The characteristic function of a process network $P_1 \parallel P_2$ composed by parallel composition can be determined by

$$\mathcal{E}_{P_1 \| P_2}(\overrightarrow{i_1}, \overrightarrow{i_2}, j) = (\mathcal{E}_{P_1}(\overrightarrow{i_1}, j), \mathcal{E}_{P_2}(\overrightarrow{i_2}, j))$$

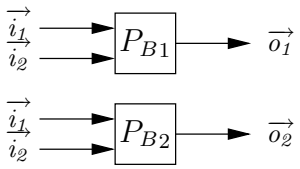
In order to illustrate how the characteristic functions of more complicated networks can be calculated, the characteristic function of the network in Figure 5.4 is derived.



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

Figure 5.4. Process network

The first step is to transform the process network into its equivalent form with two basic processes as shown in Figure 5.5.



$$\begin{aligned} P_{B1}(\overrightarrow{i_1}, \overrightarrow{i_2}) &= P_2(P_1(\overrightarrow{i_1}), P_3(\overrightarrow{i_1}, \overrightarrow{i_2})) \\ P_{B2}(\overrightarrow{i_1}, \overrightarrow{i_2}) &= P_3(\overrightarrow{i_1}, \overrightarrow{i_2}) \end{aligned}$$

Figure 5.5. Equivalent process network

For this process network the characteristic function can be derived as

$$\begin{aligned} \mathcal{E}_{PN}(\overrightarrow{i_1}, \overrightarrow{i_2}, j) &= (E(\overrightarrow{o_1}, j), E(\overrightarrow{o_2}, j)) \\ &= (\mathcal{E}_{P_{B1}}(\overrightarrow{i_1}, \overrightarrow{i_2}, j), \mathcal{E}_{P_{B2}}(\overrightarrow{i_1}, \overrightarrow{i_2}, j)) \\ &= ((\mathcal{T}_{P_{B1}}(\overrightarrow{i_1}, \overrightarrow{i_2}, j), \mathcal{V}_{P_{B1}}(\overrightarrow{i_1}, \overrightarrow{i_2}, j)), \\ &\quad (\mathcal{T}_{P_{B2}}(\overrightarrow{i_1}, \overrightarrow{i_2}, j), \mathcal{V}_{P_{B2}}(\overrightarrow{i_1}, \overrightarrow{i_2}, j))) \end{aligned} \tag{5.2}$$

The characteristic tag function is

$$\mathcal{T}_{P_{B1}}(\overrightarrow{i_1}, \overrightarrow{i_2}, j) = \mathcal{T}_{P_{B2}}(\overrightarrow{i_1}, \overrightarrow{i_2}, j) = T(\overrightarrow{i_1}, j) = T(\overrightarrow{i_2}, j)$$

since all processes are synchronous processes.

The characteristic value function $\mathcal{V}_{P_{B_1}}$ can be derived by

$$\begin{aligned} \mathcal{V}_{P_{B1}}(\overrightarrow{i_1}, \overrightarrow{i_2}, j) &= V(\overrightarrow{o_1}, j) \\ &= \mathcal{V}_{P_2}(\overrightarrow{s_1}, \overrightarrow{s_2}, j) \\ &= V(\overrightarrow{s_1}, j) * V(\overrightarrow{s_2}, j) \\ &= \mathcal{V}_{P_1}(\overrightarrow{i_1}, j) * \mathcal{V}_{P_3}(\overrightarrow{i_1}, \overrightarrow{i_2}, j) \\ &\quad \text{where } \mathcal{V}_{P_1}(\overrightarrow{i_1}, j) = \begin{cases} s_0 & j = 0 \\ V(\overrightarrow{i_1}, j-i) & \text{otherwise} \end{cases} \\ &\qquad \mathcal{V}_{P_3}(\overrightarrow{i_1}, \overrightarrow{i_2}, j) = V(\overrightarrow{i_1}, j) + V(\overrightarrow{i_2}, j) \\ &= \begin{cases} s_0 * (V(\overrightarrow{i_1}, 0) + V(\overrightarrow{i_2}, 0)) & \text{if } j = 0 \\ V(\overrightarrow{i_1}, j-1) * (V(\overrightarrow{i_1}, j) + V(\overrightarrow{i_2}, j)) & \text{if } j > 0 \end{cases} \end{aligned}$$

and the characteristic value function $\mathcal{V}_{P_{B_2}}$ by

$$\begin{aligned} \mathcal{V}_{P_{B2}}(\overrightarrow{i_1}, \overrightarrow{i_2}, j) &= V(\overrightarrow{o_2}, j) \\ &= \mathcal{V}_{P_3}(\overrightarrow{i_1}, \overrightarrow{i_2}, j) \\ &= V(\overrightarrow{i_1}, j) + V(\overrightarrow{i_2}, j) \end{aligned}$$

The results can be inserted into the expression of the characteristic function of (5.2).

Since ForSyDe disables zero-delay feedback loops, it is possible to derive a characteristic function for all meaningful process networks that are based on the process constructors and processes defined in Chapter 3 and which have at least one input signal. In practice only small process networks can be described by their characteristic function, since the mathematical expressions increase in complexity with the number of processes, the sequential depth and the number of synchronous sub-domains. However, this is often sufficient, since the characteristic function is mainly used for design transformation rules that are applied to process networks of limited size.

5.3 Design Transformations

The designer applies transformations to a system model by choosing transformation rules from the *transformation library*. The transformation rules are characterized by a *name*, the *required format and constraints of the original process network*, the *format of the transformed process network* and the *implication for the design*, i.e. the relation between the original and transformed process network expressed by the characteristic function.

Transformation rules are exemplified by the semantics preserving transformation *MapMerge* that is based on the identity

$$mapSY(f \circ g) = mapSY(f) \circ mapSY(g)$$

The transformation rule *MapMerge* takes a process network of the form

$$PN(\overrightarrow{i}) = (mapSY(f) \circ mapSY(g))(\overrightarrow{i})$$

and transforms it into the form

$$PN'(\overrightarrow{i}) = (mapSY(f \circ g))(\overrightarrow{i})$$

as illustrated in Figure 5.6.

Using the characteristic function for both expressions, it can be formally proven that both process networks PN and PN' are semantically equivalent. This means that the transformation *MapMerge* is semantic preserving. The proof is given below.

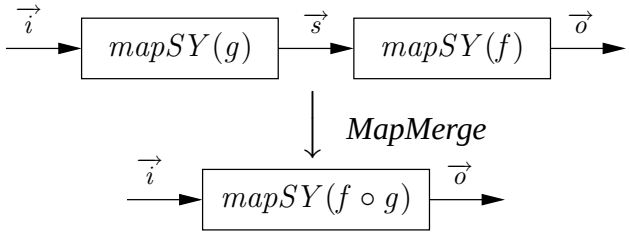


Figure 5.6. Illustration of the transformation rule *MapMerge*

First the characteristic function for $mapSY(f \circ g)$ is derived. The characteristic function can be divided into the characteristic tag function $\mathcal{T}_{\diamond(f\circ g)}$ and the characteristic value function $\mathcal{V}_{\diamond(f\circ g)}$.

$$\mathcal{E}_{\diamond(f\circ g)}(\overrightarrow{i}, j) = (\mathcal{T}_{\diamond(f\circ g)}(\overrightarrow{i}, j), \mathcal{V}_{\diamond(f\circ g)}(\overrightarrow{i}, j))$$

The characteristic tag and value functions are derived following the general expression for $mapSY$ (5.1).

$$\begin{aligned} \mathcal{T}_{\diamond(f\circ g)}(\overrightarrow{i}, j) &= T(\overrightarrow{i}, j) \\ \\ \mathcal{V}_{\diamond(f\circ g)}(\overrightarrow{i}, j) &= (f \circ g)(V(\overrightarrow{i}, j)) \\ &= f(g(V(\overrightarrow{i}, j))) \end{aligned}$$

The characteristic function $\mathcal{E}_{\diamond(f)\circ\diamond(g)}$ is given as

$$\mathcal{E}_{\diamond(f)\circ\diamond(g)}(\overrightarrow{i}, j) = (\mathcal{T}_{\diamond(f)\circ\diamond(g)}(\overrightarrow{i}, j), \mathcal{V}_{\diamond(f)\circ\diamond(g)}(\overrightarrow{i}, j))$$

where the characteristic tag function is derived as

$$\mathcal{T}_{\diamond(f)\circ\diamond(g)}(\overrightarrow{i}, j) = T(\overrightarrow{i}, j)$$

since $mapSY$ is a synchronous process constructor.

The characteristic value function returns the output value of the signal $\overrightarrow{o}$ at index j of the process $\diamond(f) \circ \diamond(g)$. The output signal $\overrightarrow{o}$ is given by $\diamond(f)(\overrightarrow{s})$ as indicated in Figure 5.6. Thus $\mathcal{V}_{\diamond(f\circ g)}(\overrightarrow{i}, j)$ can be formulated as

$$\begin{aligned} \mathcal{V}_{\diamond(f\circ g)}(\overrightarrow{i}, j) &= V(\overrightarrow{o}, j) \\ &= \mathcal{V}_{mapSY(g)}(\overrightarrow{s}, j) \end{aligned}$$

The signal $\overrightarrow{s}$ can be expressed as

$$\overrightarrow{s} = \bigoplus_{k=0}^{\infty} \mathcal{E}_{\diamond(g)}(\overrightarrow{i}, k)$$

which leads to

$$\begin{aligned}
\mathcal{V}_{\diamond(f\circ g)}(\overrightarrow{i}, j) &= \mathcal{V}_{\diamond(f)}(\overbrace{\bigoplus_{k=0}^{\infty} \mathcal{E}_{\diamond(g)}(\overrightarrow{i}, k)}^{\overrightarrow{s}}, j) \\
&= f(V(\bigoplus_{k=0}^{\infty} g(V(\overrightarrow{i}, k)), j)) \\
&= f(g(V(\overrightarrow{i}, j)))
\end{aligned}$$

Since the characteristic function of both process networks PN and PN' is equivalent, it shows that the transformation rule *MapMerge* is semantic preserving.

Transformation Rule: $MapMerge(PN)$
Orignal Process Network:

$$\begin{aligned}
\overrightarrow{o} &= PN(\overrightarrow{i}) \\
PN(\overrightarrow{i}) &= (mapSY(f) \circ mapSY(g))(\overrightarrow{i})
\end{aligned}$$

Transformed Process Network:

$$\begin{aligned}
\overrightarrow{o} &= PN'(\overrightarrow{i}) \\
PN'(\overrightarrow{i}) &= (mapSY(f \circ g))(\overrightarrow{i})
\end{aligned}$$

Implication:

$$\mathcal{E}_{PN}(\overrightarrow{i}, j) = \mathcal{E}_{PN'}(\overrightarrow{i}, j)$$

Figure 5.7. The transformation rule *MapMerge*

The transformation rule for *MapMerge* is given in Figure 5.7. All process networks that match the format given in Original Process Network can be transformed by the application of *MapMerge* into the format given in Transformed Process Network. The Implication expresses that the transformation rule is semantic preserving and thus does not change the meaning of the process network, since the characteristic functions of both process networks are identical.

The implication part provides the designer with the information to what extent an application of a transformation rule changes the meaning of the original process network.

The transformation rule can be expressed in Haskell by the following code.

```
-- Transformation MapMerge
mapMerge f g s = pn_ref f g s

-- Original Process
pn_org f g s = (mapSY f . mapSY g) s

-- Transformed Process
pn_ref f g s = mapSY (f . g) s
```

An occurrence of the pattern `mapSY f . mapSY g` in a ForSyDe model can be replaced by the pattern `mapSY (f . g)` or alternatively with the pattern `mapMerge f g`. Given a signal

$$\overrightarrow{s_1} = \ll 1, 2, 3, 4, 5, 6 \gg$$

the following Haskell session with the Haskell interpreter Hugs 98 shows how the transformation rule *MapMerge* can be used.

```
Hugs98Prompt> (mapSY (+1) . mapSY (*3)) s1
{4,7,10,13,16,19}
Hugs98Prompt> (mapSY ((+1) . (*3))) s1
{4,7,10,13,16,19}
Hugs98Prompt> mapMerge (+1) (*3) s1
{4,7,10,13,16,19}
```

In the same way all other transformation rules can be expressed and applied in Haskell. To date transformations are introduced manually in the Haskell code of the system model. However, tools like ULTRA[1] [3], which is an interactive program transformation system for Haskell programs, may be used in future versions of ForSyDe to support the transformation process.

Since *MapMerge* is semantic preserving, there also exists the reverse transformation rule, which is named *MapSplit* and given in Figure 5.8.

Figure 5.9 illustrates the application of a transformation rule by a simple example. Here, *MapMerge* is used to merge two processes into a single process over subsystem borders. Given the trivial case that

$$\begin{aligned} f(x) &= x + 2 \\ g(x) &= x + 5 \end{aligned}$$

the transformed process is $mapSY(h)$, where

$$h(x) = x + 7$$

[1] Ulm's transformation tool

Transformation Rule: $MapSplit(PN)$
Orignal Process Network:

$$\begin{aligned} \overrightarrow{o} &= PN(\overrightarrow{i}) \\ PN(\overrightarrow{i}) &= (mapSY(f \circ g))(\overrightarrow{i}) \end{aligned}$$

Transformed Process Network:

$$\begin{aligned} \overrightarrow{o} &= PN'(\overrightarrow{i}) \\ PN'(\overrightarrow{i}) &= (mapSY(f) \circ mapSY(g))(\overrightarrow{i}) \end{aligned}$$

Implication:

$$\mathcal{E}_{PN'}(\overrightarrow{i}, j) = \mathcal{E}_{PN}(\overrightarrow{i}, j)$$

Figure 5.8. The transformation rule *MapSplit*

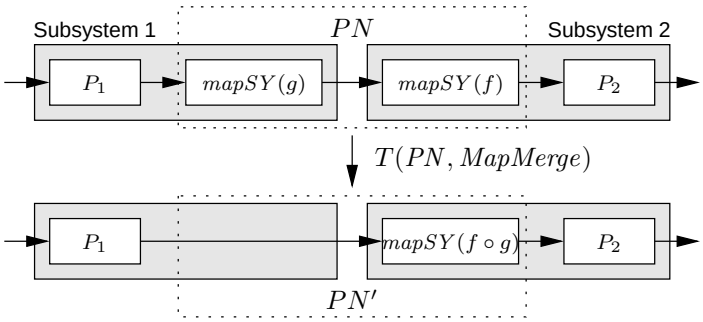


Figure 5.9. Application of a semantic preserving transformation

Thus the merging of the two processes would optimize the design by saving one adder.

Note, that two processes of the form $mapSY(f) \circ mapSY(g)$ can always be merged, while the splitting of a process $mapSY(h)$ into two processes usually needs support from the designer, since two suitable functions f and g have to be selected, so that $h = f \circ g$. The splitting of processes is often not efficient as in the case of $h(x) = x + 7$.

Figure 5.9 illustrates another powerful feature of ForSyDe. Since a system model is a hierarchy of functions, processes can easily be moved between subsystem borders as in the case of Figure 5.9, where $mapSY(f \circ g)$ can be moved from subsystem 2 to subsystem 1 and vice versa..

Another example for a semantic preserving transformation is *BalancedTree* which is illustrated in Figure 5.10.

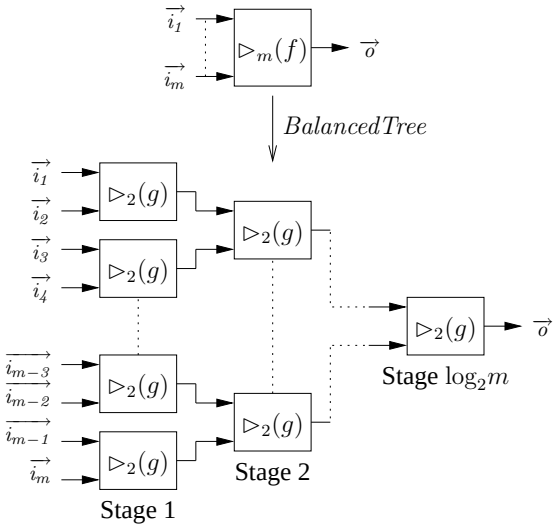


Figure 5.10. Illustration of the transformation rule $BalancedTree$

The transformation *BalancedTree* transforms a combinational m-input process into a balanced network of $m - 1$ processes with two inputs. In contrast to the transformation *MapMerge*, which is applicable to all processes that have the format $\diamond(f) \circ \diamond(g)$, the transformation *BalancedTree* puts additional requirements on the process format.

It is not sufficient, that the combinational process has the format $\triangleright_m(f)$, but there are also constraints on the number of inputs m, which has to be 4,8,16,... and

on the form of the combinational function

$$f(x_1, \ldots, x_m) = x_1 \otimes \cdots \otimes x_m$$

where the operator $\otimes$ must be associative. These additional requirements are formulated in the Original Process Network part of the transformation rule *BalancedTree* that is given in Figure 5.11. The transformation rule *BalancedTree* can for instance be used to transform a 4-input adder into a balanced circuit with three 2-input adders as shown later in this section.

Transformation Rule: $BalancedTree(PN)$
Orignal Process Network:

$$\begin{array}{lcl}
\overrightarrow{o} & = & PN(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}) \\
PN(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}) & = & \triangleright_m(f)(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}) \\
m & = & 2^k \quad k \in \mathbb{N}_1 \\
f(x_1, \ldots, x_m) & = & x_1 \otimes \cdots \otimes x_m; \quad \otimes \text{ is associative}
\end{array}$$

Transformed Process Network:

$$\begin{array}{lcl}
\overrightarrow{o} & = & PN'(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}) \\
PN'(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}) & = & \triangleright_2(g)(\triangleright_2(g) \ldots (\triangleright_2(g)(\triangleright_2(g)(\overrightarrow{i_1}, \overrightarrow{i_2}), \triangleright_2(g)(\overrightarrow{i_3}, \overrightarrow{i_4}))), \\
 & & \triangleright_2(g) \ldots (\triangleright_2(g)(\triangleright_2(g)(\overrightarrow{i_{m-3}}, \overrightarrow{i_{m-2}}), \triangleright_2(g)(\overrightarrow{i_{m-1}}, \overrightarrow{i_m})))) \\
g(x, y) & = & x \oplus y
\end{array}$$

Implication:

$$\mathcal{E}_{PN'}(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}, j) \quad = \quad \mathcal{E}_{PN}(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}, j)$$

Figure 5.11. The transformation rule *BalancedTree*

Design decisions change the meaning of the system model and thus imply also a change of the characteristic function of the transformed process. Using the characteristic function design decision rules can be defined that change the meaning of the system model in a controlled manner.

In order to classify design decision rules, the terms *functional equivalence* (Definition 5.7) and *tail equivalence* (Definition 5.8) are introduced.

Definition 5.7 (Functional Equivalence) *A process P' is* functionally equivalent *to a process P, if and only if*

$$\mathcal{E}_{\triangle_k(s_0) \circ_m P}(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}, j) \quad = \quad \mathcal{E}_{P'}(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}, j)$$

for some $k > 0$ and s_0.

The output of the process P' is the output of the process P n cycles delayed.

Definition 5.8 (Tail Equivalence) *Two processes P and P' are* tail equivalent, *if there exists a k such that*

$$\mathcal{E}_P(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}, j) = \mathcal{E}_{P'}(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}, j) \quad \forall j > k \in \mathbb{N}_0$$

The output signals of the process networks P and P' are identical for events with index $j > k$, but not for the initial $k + 1$ events.

A transformation rule that transforms a process P into a process P' is called a *functionally equivalent transformation rule*, if the processes P' is functionally equivalent to P, and a *tail equivalent transformation rule*, if the processes P and P' are tail equivalent.

Functional equivalence and tail equivalence are examples for a relation between original and transformed process network as required by the definition of a design decision rule (Definition 5.6).

Transformation Rule: $AddDelay(k, s_0)$
Orignal Process Network:

$$\overrightarrow{o} = PN(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m})$$

Transformed Process Network:

$$\overrightarrow{o'} = PN'(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m})$$
$$PN'(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}) = \triangle_k(s_0) \circ_m PN(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m})$$

Implication:

$$\mathcal{E}_{PN'}(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}, j) = \mathcal{E}_{\triangle_k(s_0) \circ_m PN}(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}, j)$$

Figure 5.12. The transformation rule *AddDelay*

The transformation rule *AddDelay* (Figure 5.12), which is illustrated in Figure 5.13 is the basic example for a functionally equivalent transformation rule. The application of this rule adds a delay process to the output of the original process network.

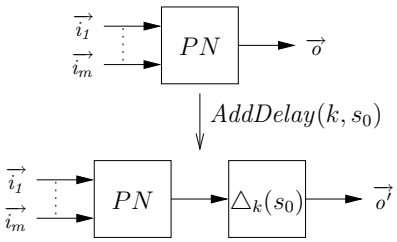


Figure 5.13. Illustration of the transformation rule *AddDelay*

Transformation Rule: $MoveDelayToInput(k, s_0)$
Orignal Process Network:

$$\begin{aligned} \overrightarrow{o} &= PN(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}) \\ PN(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}) &= \triangle_k(s_0) \circ_m \triangleright_m(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}) \end{aligned}$$

Transformed Process Network:

$$\begin{aligned} \overrightarrow{o'} &= PN'(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}) \\ PN'(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}) &= \triangleright_m(f)(\triangle_k(s_0)(\overrightarrow{i_1}), \ldots, \triangle_k(s_0)(\overrightarrow{i_m})) \end{aligned}$$

Implication:

$$\mathcal{E}_{PN'}(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}, j) = \mathcal{E}_{PN}(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}, j) \quad \forall j \geq k$$

Figure 5.14. The transformation rule *MoveDelayToInput*

The transformation rule *MoveDelayToInput* (Figure 5.14), which is illustrated in Figure 5.15 is an example of a tail equivalent transformation rule. The application of this rule moves a delay from the output of the original process network to the input.

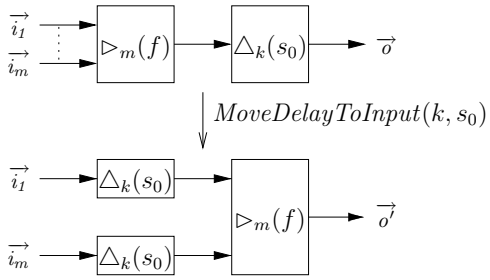


Figure 5.15. Illustration of the transformation rule *MoveDelayToInput*

The transformation rules *AddDelay* and *MoveDelayToInput* are used for the more complex design decision rule *PipelinedTree* (Figure 5.16), which is illustrated in Figure 5.17.

This transformation rule is composed of an initial $AddDelay(\log_2 m, s_0)$ transformation and a number of $MoveDelayToInput(1, s_0)$ transformations.The implication of *PipelinedTree* shows that this transformation is a combination of functional equivalent and tail equivalent transformations. The output is k cycles delayed, but is otherwise equivalent to the output of the original process network for all indexes $j > k$.

Combining the transformation rules $BalancedTree$ and $PipelinedTree$, a new transformation rule *BalancedPipelinedTree* can be generated that can be used to

Transformation Rule: $PipelinedTree$

Orignal Process Network:

$$\begin{aligned}
\overrightarrow{o} &= PN(\overrightarrow{i_1},\ldots,\overrightarrow{i_m}) \\
PN(\overrightarrow{i_1},\ldots,\overrightarrow{i_m}) &= \triangleright_2(f_{m-1})(\ldots(\triangleright_2(f_1)(\overrightarrow{i_1},\overrightarrow{i_2}),\ldots), \\
&\quad \ldots(\triangleright_2(\ldots,\triangleright_2(f_{m/2})(\overrightarrow{i_{m-1}},\overrightarrow{i_m}))) \\
m &= 2^k \quad k>1
\end{aligned}$$

Transformed Process Network:

$$\begin{aligned}
PN'(\overrightarrow{i_1},\ldots,\overrightarrow{i_m}) &= \triangle_1(s_0)\circ\triangleright_2(f_{m-1})(\ldots(\triangle_1(s_0)\circ\triangleright_2(f_1)(\overrightarrow{i_1},\overrightarrow{i_2}),\ldots), \\
&\quad \ldots(\ldots,\triangle_1(m_0)\circ\triangleright_2(f_{m/2})(\overrightarrow{i_{m-1}},\overrightarrow{i_m})))
\end{aligned}$$

Implication:

$$\mathcal{E}_{PN'}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j) = \mathcal{E}_{\triangle_k(m_0)\circ PN}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j) \quad \forall j \geq k$$

Figure 5.16. The transformation rule *PipelinedTree*

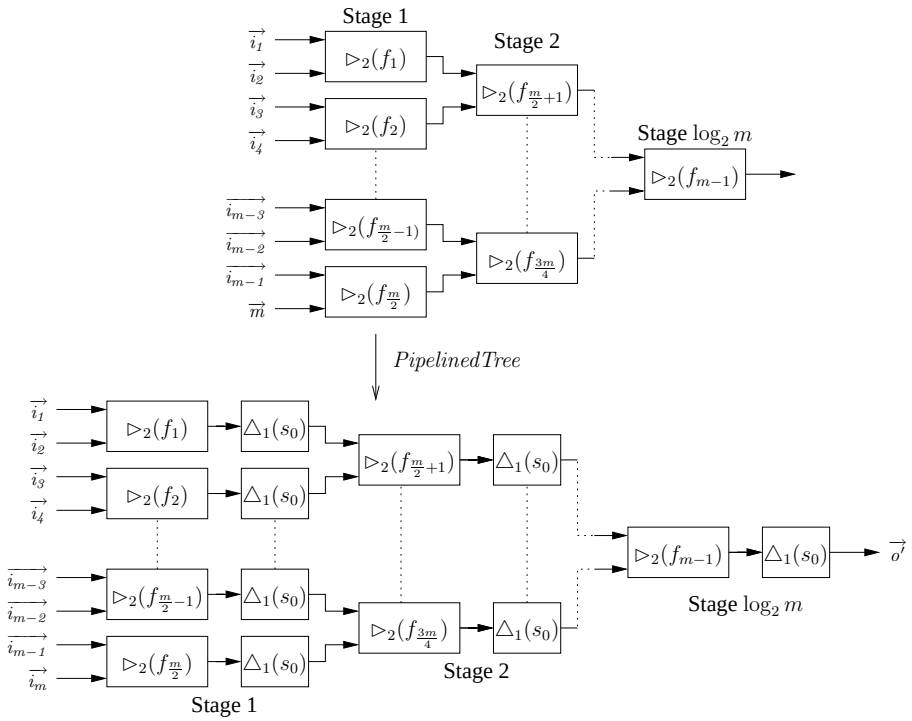


Figure 5.17. Illustration of the transformation rule *PipelinedTree*

enhance the throughput of a combinational function by pipelining as illustrated in Figure 5.18.

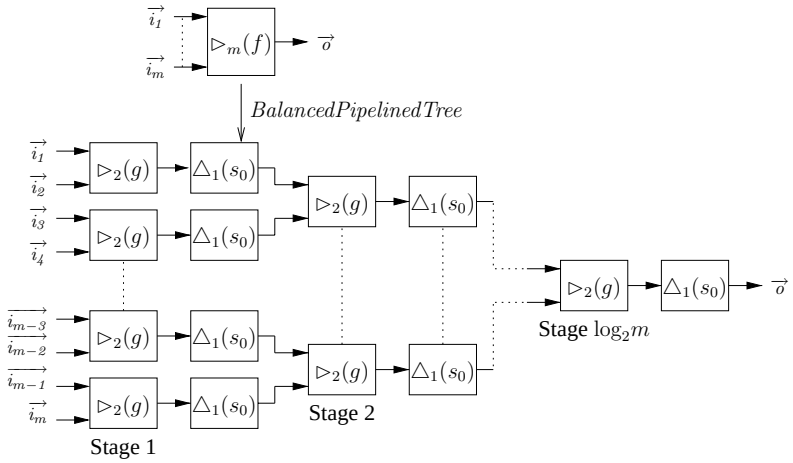


Figure 5.18. Combining *BalancedTree* and *PipelinedTree*

A direct translation of a computation intensive algorithm such as an n-th-order FIR-filter results in an implementation with a large amount of multipliers and adders. Using the concept of synchronous sub-domains the transformation $SerialClockDomain$ was developed to transform a combinational process with a regular structure into a structure with two separate clock domains. The transformed process network uses an FSM process to schedule the operations into several clock cycles. This transformation, illustrated in Figure 5.19 and formally given in Figure 5.20, allows an efficient implementation, if there are identical operations that can be shared.

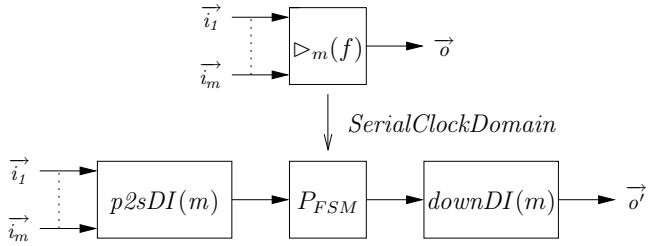


Figure 5.19. Illustration of the transformation rule *SerialClockDomain*

Transformation Rule: $SerialClockDomain$
Orignal Process Network:

$$\begin{array}{lcl}
\overrightarrow{o} & = & PN(\overrightarrow{i_1},\ldots,\overrightarrow{i_m}) \\
PN(\overrightarrow{i_1},\ldots,\overrightarrow{i_m}) & = & \triangleright_m(f)(\overrightarrow{i_1},\ldots,\overrightarrow{i_m}) \\
f(x_1,\ldots,x_m) & = & g_{m-1}(h_{m-1}(x_m),(\ldots,(g_1(h_1(x_2),h_0(x_1)))\ldots))
\end{array}$$

Transformed Process Network:

$$\begin{array}{lcl}
\overrightarrow{o'} & = & PN'(\overrightarrow{i_1},\ldots,\overrightarrow{i_m}) \\
PN'(\overrightarrow{i_1},\ldots,\overrightarrow{i_m}) & = & (downDI(m)\circ P_{FSM}\circ p2sDI(m))(\overrightarrow{i_1},\ldots,\overrightarrow{i_m}) \\
P_{FSM} & = & mooreSY(u,w,s_0) \\
u(x,(k,s)) & = & \begin{cases} (1,h_k(x)) & \text{if } k=0 \\ (k+1,g_k(h_k(x),s)) & \text{if } 0<k<m-1 \\ (0,g_k(h_k(x),s)) & \text{if } k=m-1 \end{cases} \\
w(k,s) & = & \begin{cases} s & k=0 \\ \perp & 0<k<m \end{cases} \\
s_0 & = & (0,initValue)
\end{array}$$

Implication:

$$\mathcal{E}_{PN'}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j) \quad = \quad \mathcal{E}_{\triangle_1(s_0)\circ PN}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j)$$

Figure 5.20. Transformation rule *SerialClockDomain*

The transformation rule requires the combinational function f to have the format

$$f(x_1,\ldots,x_m) = g_{m-1}(h_{m-1}(x_m),(\ldots,(g_1(h_1(x_2),h_0(x_1)))\ldots))$$

in order to be able to schedule the computation of f into m steps. In the first step

$$y_0 = h_0(x_1)$$

is calculated and in the following $m-1$ steps the intermediate value y_{i-1} is used to calculate the final result y_{m-1} according to

$$y_i = g_i(h_i(x(i+1)),y_{i-1})$$

The transformed process network works as follows. During an input event cycle the domain interface $p2sDI(m)$ (Parallel to Serial) reads all input values at signal rate r and outputs them as a sequence $< x_1,\ldots,x_m >$ with the signal rate mr one by one in the corresponding m event cycles. The process P_{FSM} is based on the process constructor $mooreSY$ and executes the combinational function f of the original process within m event cycles. In state 0 the function h_0 is applied

to the first input value x_1 and the result is stored as intermediate value s. In the $n-1$ following states the function g_i is applied to $h_i(x_{i+1})$ and the intermediate value. At tag 0,m,2m,... the process outputs the intermediate value, otherwise the output is absent ($\perp$). The domain interface $downDI(m)$ down-samples the input signal to signal rate r and outputs only each m-th input value starting with tag 0, thus suppressing the absent values from the output of P_{FSM}.

As domain interfaces can be characterized by a characteristic function, it means that, though not shown here, the characteristic function for the whole transformed process network can be developed. It follows that $SerialClockDomain$ delays the output of the transformed process network one event cycle compared to the original process network as stated in the Implication. The first event of the output signal has the second component of the initial state, the value *initValue*.

The transformation is expressed in Haskell as follows. The original process format is express by

```
pn_org hs gs = zipWithxSY f
  where f = makeFunc hs gs
```

where `zipWithxSY` is the vector-based implementation of $zipWithSY_m$ for any size m and `makeFunc hs gs` expresses the format of the combinational function f according to the format given in OriginalProcessNetwork; `hs` and `gs` are lists of functions, where `hs` represents the functions h_i and `gs` the functions g_i.

```
makeFunc [h] [] (x:>NullV)     = h x
makeFunc (h:hs) (g:gs) (x:>xv) = g (h x) (makeFunc hs gs xv)
```

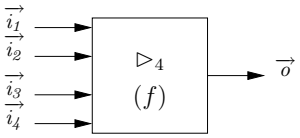
The transformed process is expressed by

```
pn_ref hs gs s_0 = (downDI m . p_fsm hs gs s_0 . par2serxDI)
   where
     p_fsm hs gs s_0 = mooreSY u w s_0
     u (k,s) x  | k == 0       = (1, h(k) x)
                | 0 < k && k < m-1 = (k+1, g(k) (h(k) x) s)
                | k == m-1     = (0, g(k) (h(k) x) s)
     w (k,s)    | k == 0       = Prst s
                | 0 < k && k < m = Abst
     h(k) = hs !! k
     g(k) = gs !! (k-1)
     m = length hs
```

where `hs !! k` returns the k-th value (starting with 0) in the list `hs`. Thus *SerialClockDomain* is a functionally equivalent transformation rule.

The parameter s_0 is always a tuple of the kind $(0, initValue)$ where $initValue$ can be any value of the demanded data type. Thus the transformation is expressed as

```
serialClockDomain hs gs s_0 = pn_ref hs gs s_0
```



$$\overrightarrow{o} = zipWithSY_4(f)(\overrightarrow{i_1}, \overrightarrow{i_2}, \overrightarrow{i_3}, \overrightarrow{i_4})$$
$$where$$
$$f(x_1, x_2, x_3, x_4) = x_1 + x_2 + x_3 + x_4$$

Figure 5.21. A four-input adder process

The chapter is concluded by the example of a four input adder (Figure 5.21) that is refined with the following transformations.

- *BalancedTree*
- *PipelinedBalancedTree*
- *SerialClockDomain*

The example is expressed in Haskell to show the result of the transformations. For all examples a tuple of four signals, vs, is used, which has the following values.

$$vs = (\ll 1,2,3,4,5 \gg, \ll 1,2,3,4,5 \gg, \ll 1,2,3,4,5 \gg, \ll 1,2,3,4,5 \gg)$$

Since *BalancedTree* is a semantic preserving design transformation rule the refined adder behaves exactly as the original one.

```
Hugs98Prompt> balancedTree (+) vs
{4,8,12,16,20}
```

The refined pipelined adder introduces two extra delays. Here 0 is given as initial value for the delay elements.

```
Hugs98Prompt> balancedPipelinedTree 0 (+) vs
{0,0,4,8,12}
```

Using the transformation *SerialClockDomain* the adder is refined into

```
adder_ref = serialClockDomain hs gs s0
  where
    hs = [id, id, id, id]
    gs = [(+), (+), (+)]
    s0 = (0, 0)
```

Simulation shows that the output of the refined adder is one event cycle delayed.

```
Hugs98Prompt> adder_org vs
{4,8,12,16,20}
Hugs98Prompt> adder_ref vs
{0,4,8,12,16}
```

5.4 Refinement of the Equalizer

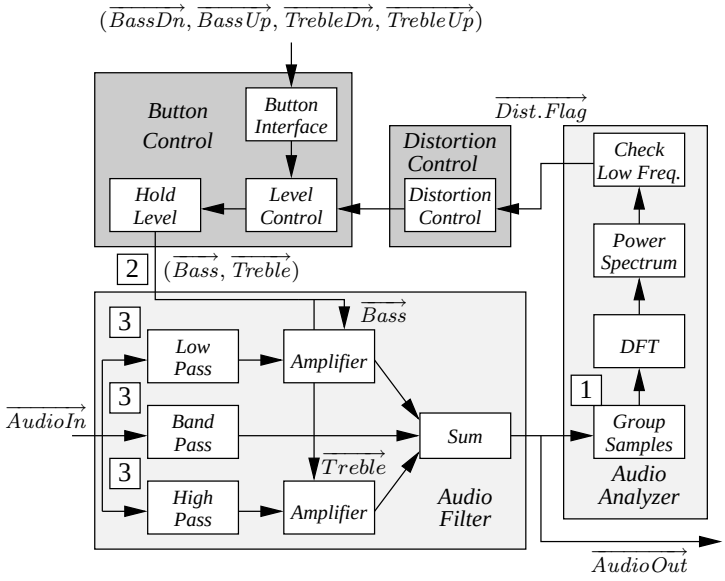


Figure 5.22. Refinement of the equalizer (simplified figure)

In the following the specification model of the equalizer (Chapter 4) is used to discuss three refinement techniques as indicated in Figure 5.22.

1. Refinement of the Clock Domain (Section 5.4.1)
2. Communication Refinement (Section 5.4.2)

3. Resource Sharing (Section 5.4.3)

5.4.1 Refinement of the Clock Domain

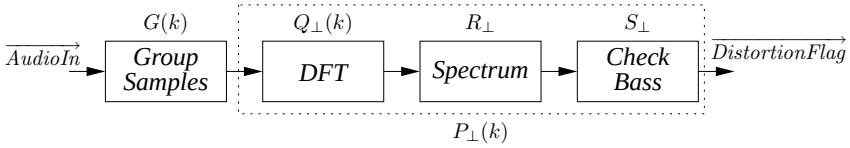


Figure 5.23. The $AudioAnalyzer$

Figure 5.23 shows the *Audio Analyzer* subsystem, which includes a Discrete-Fourier Transform (DFT) algorithm. The internal function *dft* takes a vector of $k = 2^u$ samples and produces the corresponding DFT result in form of a vector of size k that is denoted as $\langle x_0, x_1, \ldots, x_{k-1} \rangle$. The DFT is used to determine the frequency spectrum of a signal and is modeled in the process *DFT* ($Q_\perp(k)$). The process *Power Spectrum* ($R_\perp$) calculates the power spectrum. The process *Check Low Frequencies* ($S_\perp$) analyzes if the power of the low frequencies exceeds a threshold and issues a warning in this case, which is sent to the *Distortion Control*. The process *Group Samples* ($G(k)$) reads k samples and groups them into a vector of size k. The computation of this vector takes k event cycles and serves as input for the DFT. Since a synchronous computational model is used in the specification model, the grouping process has to produce an output event for each input event. This results in the output of $k-1$ absent values ($\perp$) for each computed DFT value.

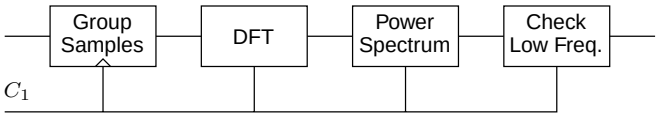


Figure 5.24. Direct implementation of the $AudioAnalyzer$

Due to the definition of $GroupSamples$, which is based on $groupSY$ and defined in (4.1), the process $P_\perp(k)$ has to process many absent values. This is not a drawback for the specification phase, but a direct implementation (Figure 5.24) using the semantics of Chapter 6 can make no use of the fact, that the DFT has only to be calculated at each k-th clock cycle. Instead the DFT process has to produce the result during a single event cycle and will be idle during $k-1$ clock cycles. This implementation of the *Audio Analyzer* is very inefficient, since the computation of the DFT function is clearly the most time consuming and will determine the clock period for the whole system and thus the overall system performance.

In order to get a more efficient specification, the ForSyDe methodology allows to introduce synchronous sub-domains into the system model during the refinement process as discussed in Section 3.4.

The idea of the following transformation is to introduce a new clock domain after the process $GroupSamples$, in order to filter the absent values and allow for a more efficient use of the DFT process.

In order to develop this transformation, the characteristic function for $P_\perp(k) \circ G(k)$ has to be known.

The characteristic function of $G(k)$ is given as

$$\mathcal{E}_{G(k)}(\overrightarrow{i}, j) = (\mathcal{T}_{G(k)}(\overrightarrow{i}, j), \mathcal{V}_{G(k)}(\overrightarrow{i}, j))$$

where

$$\begin{aligned} \mathcal{T}_{G(k)}(\overrightarrow{i}, j) &= T(\overrightarrow{i}, j) \\ \mathcal{V}_{G(k)}(\overrightarrow{i}, j) &= \begin{cases} \langle\rangle & \text{if } j = 0 \\ \langle V(\overrightarrow{i}, j-k), \ldots, V(\overrightarrow{i}, j-1)\rangle & \text{if } \exists w \in \mathbb{N}_1 . j = wk \\ \perp & \text{otherwise} \end{cases} \end{aligned}$$

All processes $Q_\perp(n)$, $R_\perp$, $S_\perp$ are constructed with the process constructor *mapSY* and the absent extension function Ψ (3.1).

$$\begin{aligned} Q_\perp(k) &= mapSY(\Psi(q(k))) \\ R_\perp &= mapSY(\Psi(r)) \\ S_\perp &= mapSY(\Psi(s) \end{aligned}$$

where $q(k)$, r and s model the computation functions of the processes $Q_\perp(k)$, $R_\perp$ and $S_\perp$.

Using the identities

$$\begin{aligned} \Psi(f) \circ \Psi(g) &= \Psi(f \circ g) \\ mapSY(f) \circ mapSY(g) &= mapSY(f \circ g) \end{aligned}$$

which gives

$$mapSY(\Psi(f)) \circ mapSY(\Psi(g)) = mapSY(\Psi(f \circ g))$$

these processes can be composed into one single process $P_\perp(k)$.

$$\begin{aligned} P_\perp(n) &= mapSY(\Psi(p(k))) \\ &\text{where } p(k) = s \circ r \circ q(k) \end{aligned}$$

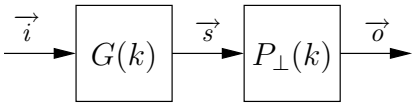
with the characteristic function

$$\mathcal{E}_{P_\perp(k)}(\overrightarrow{i},j) = (\mathcal{T}_{P_\perp(k)}(\overrightarrow{i},j), \mathcal{V}_{P_\perp(k)}(\overrightarrow{i},j))$$

where

$$\mathcal{T}_{P_\perp(k)}(\overrightarrow{i},j) = T(\overrightarrow{i},j)$$

$$\mathcal{V}_{P_\perp(k)}(\overrightarrow{i},j) = \begin{cases} \perp & \text{if } V(\overrightarrow{i},j) = \perp \\ p(V(\overrightarrow{i},j)) & \text{otherwise} \end{cases}$$



$$\overrightarrow{i} =< v_0, v_1, \cdots >$$

$$\overrightarrow{s} =< \langle\rangle, \underbrace{\perp, \ldots, \perp}_{k-1}, \langle v_0, \ldots, v_{k-1}\rangle, \perp, \cdots >$$

$$\overrightarrow{o} =< p(\langle\rangle), \underbrace{\perp, \ldots, \perp}_{k-1}, p(\langle v_0, \ldots, v_{k-1}\rangle), \perp, \cdots >$$

Figure 5.25. Mathematical abstraction of the specification model

Thus the specification model of the *Audio Analyzer* can be expressed as $P_\perp(k) \circ G(k)$ and is illustrated in Figure 5.25. The process network has the characteristic function

$$\begin{aligned} &\mathcal{E}_{P_\perp(k)\circ G(k)}(\overrightarrow{i},j) = (\mathcal{T}_{P_\perp(k)\circ G(k)}(\overrightarrow{i},j), \mathcal{V}_{P_\perp(k)\circ G(k)}(\overrightarrow{i},j)) \\ &\text{where} \\ &\quad \mathcal{T}_{P_\perp(k)\circ G(k)}(\overrightarrow{i},j) = T(\overrightarrow{i},j) \\ &\quad \mathcal{V}_{P_\perp(k)\circ G(k)}(\overrightarrow{i},j) \\ &\quad = \begin{cases} p(\langle\rangle) & \text{if } j = 0 \\ p(\langle V(\overrightarrow{i},j-k), \ldots, V(\overrightarrow{i},j-1)\rangle) & \text{if } \exists w \in \mathbb{N}_1 . j = wk \\ \perp & \text{otherwise} \end{cases} \end{aligned} \tag{5.3}$$

Following the initial idea that is based on the introduction of a synchronous sub-domain, the identity

$$G(k) = U(k) \circ D(k) \circ G(k)$$

where $U(k) = upDI(k)$ and $D(k) = downDI(k)$, can be derived. The identity uses the special characteristic of the grouping process $G(k)$[2] and is proven by

[2] $U(k) \circ D(k)$ is not an identity!

comparison of the characteristic functions $\mathcal{E}_{G(k)}$ and $\mathcal{E}_{U(k)\circ D(k)\circ G(k)}$. First the characteristic function of the process network $U(k) \circ D(k)$ is developed.

The characteristic tag function for the process $U(k) \circ D(k)$ is

$$\begin{aligned}
\mathcal{T}_{U(k)\circ D(k)}(\overrightarrow{i}, j) &= \mathcal{T}_{U(k)}(\bigoplus_{u=0}^{\infty} \mathcal{E}_{D(k)}(\overrightarrow{i}, u), j) \\
&= \tfrac{j}{k} C(\bigoplus_{u=0}^{\infty} \mathcal{E}_{D(k)}(\overrightarrow{i}, u)) \\
&= \tfrac{j}{k} k C(\overrightarrow{i}) \\
&= T(\overrightarrow{i}, j)
\end{aligned}$$

The characteristic value function for the process $U(k) \circ D(k)$ is

$$\begin{aligned}
\mathcal{V}_{U(k)\circ D(k)}(\overrightarrow{i}, j) &= \mathcal{V}_{U(k)}(\bigoplus_{u=0}^{\infty} \mathcal{E}_{D(k)}(\overrightarrow{i}, u), j) \\
&= \begin{cases} V(\bigoplus_{u=0}^{\infty} \mathcal{E}_{D(k)}(\overrightarrow{i}, u), \mathrm{mod}(j,k)) & \text{if } \exists w \in \mathbb{N}_0 . j = wk \\ \bot & \text{otherwise} \end{cases} \\
&= \begin{cases} V(\mathcal{E}_{D(k)}(\overrightarrow{i}, \mathrm{mod}(j,k))) & \text{if } \exists w \in \mathbb{N}_0 . j = wk \\ \bot & \text{otherwise} \end{cases} \\
&= \begin{cases} \mathcal{V}_{D(k)}(\overrightarrow{i}, \mathrm{mod}(j,k)) & \text{if } \exists w \in \mathbb{N}_0 . j = wk \\ \bot & \text{otherwise} \end{cases} \\
&= \begin{cases} V(\overrightarrow{i}, \mathrm{mod}(j,k)) & \text{if } \exists w \in \mathbb{N}_0 . j = wk \\ \bot & \text{otherwise} \end{cases}
\end{aligned}$$

The characteristic function of the process $U(k) \circ D(k)$ shows that the process is synchronous. The process propagates only every k-th event (counted from 0) to the output. All other output events have an absent value.

The characteristic function of the process $U(k) \circ D(k)$ is used to derive the characteristic function for $U(k) \circ D(k) \circ G(k)$. The characteristic tag function is trivial, since both processes $U(k) \circ D(k)$ and $G(k)$ are synchronous.

$$\mathcal{T}_{U(k)\circ D(k)\circ G(k)}(\overrightarrow{i}, j) = T(\overrightarrow{i}, j)$$

The characteristic value function for $U(k) \circ D(k) \circ G(k)$ is given below.

$$
\begin{aligned}
&\mathcal{V}_{U(k)\circ D(k)\circ G(k)}(\overrightarrow{i}, j) \\
&= \begin{cases} V(\bigoplus_{u=0}^{\infty}(\overrightarrow{i}, u), \mathrm{mod}(j,k)) & \text{if } \exists w \in \mathbb{N}_0 . j = wk \\ \bot & \text{otherwise} \end{cases} \\
&= \begin{cases} \mathcal{V}_{G(k)}(\overrightarrow{i}, \mathrm{mod}(j,k)) & \text{if } \exists w \in \mathbb{N}_0 . j = wk \\ \bot & \text{otherwise} \end{cases} \\
&= \begin{cases} <> & \text{if } j = 0 \\ < V(\overrightarrow{i}, w-k), \ldots, V(\overrightarrow{i}, w-1) > & \text{if } \exists w \in \mathbb{N}_0 . j = wk \\ \bot & \text{otherwise} \end{cases}
\end{aligned}
$$

Thus it is shown by their characteristic function that $G(k)$ and $U(k) \circ D(k) \circ G(k)$ are equivalent processes.

Using another identity

$$P_\bot(k) \circ U(k) = U(k) \circ P_\bot(k)$$

which can easily be proven, it follows that

$$
\begin{aligned}
P_\bot(k) \circ G(k) &= P_\bot(k) \circ U(k) \circ D(k) \circ G(k) \\
&= U(k) \circ P_\bot(k) \circ D(k) \circ G(k) \\
&= U(k) \circ P(k) \circ D(k) \circ G(k)
\end{aligned} \tag{5.4}
$$

In the last step $P_\bot(k)$ is replaced with $P(k)$ since $D(k) \circ G(k)$ does not produce any absent values. Since only semantic preserving transformations have been used, it follows that the refined process network has the same semantical meaning as the process network of Figure 5.25 and thus the same characteristic function (5.3). Analyzing (5.4) it can be stated that the process $P(k)$ processes events only at each k-th tag and thus can be implemented with a slower clock.

Based on these considerations a semantic-preserving transformation that expresses the identity

$$P_\bot(k) \circ G(k) = U(k) \circ P(k) \circ D(k) \circ G(k)$$

and introduces a synchronous sub-domain can be defined. This transformation is called *GroupToMultiRate* and given in Figure 5.26.

The synchronous sub-domain is implemented with a clock frequency f_{C_2} that is $k = 2^u$ times slower than the clock frequency f_{C_1} of the main synchronous

Transformation Rule: $GroupToMultiRate$
Orignal Process Network:

$$\begin{aligned}\overrightarrow{o} &= PN(\overrightarrow{i})\\ PN(\overrightarrow{i}) &= (mapSY(\Psi(f)) \circ groupSY(k))(\overrightarrow{i})\end{aligned}$$

Transformed Process Network:

$$\begin{aligned}\overrightarrow{o} &= PN'(\overrightarrow{i})\\ PN'(\overrightarrow{i}) &= (upDI(k) \circ mapSY(f) \circ downDI(k) \circ groupSY(k))(\overrightarrow{i})\end{aligned}$$

Implication:

$$\mathcal{E}_{PN}(\overrightarrow{i}, j) = \mathcal{E}_{PN'}(\overrightarrow{i}, j)$$

Figure 5.26. Transformation rule *GroupToMultiRate*

domain. In a further design step the DFT can be refined into an FFT[3]. The implementation of the transformed $AudioAnalyzer$ is illustrated in Figure 5.27, where the time interval that is available for the calculation of the DFT is k times longer than in Figure 5.24 without slowing down the rest of the system.

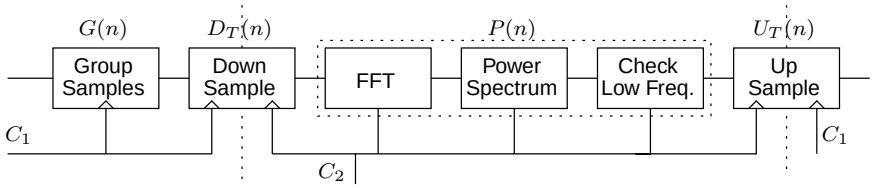


Figure 5.27. The $AudioAnalyzer$ after Refinement

The transformation *GroupToMultiRate* can be directly expressed in Haskell. The implementation of the process $GroupSamples$ follows directly the formal definition of the process in (4.1).

```
groupSY k = mooreSY f g s0
  where
    s0 = NullV
    f v x | lengthV v == 0 = unitV x
          | lengthV v == k = unitV x
          | otherwise      = v <: x
    g v   | lengthV v == 0 = Prst NullV
    g v   | lengthV v == k = Prst v
    g v   | otherwise      = Abst
```

The original process format is expressed in Haskell as

[3]Both algorithms are part of the ForSyDe standard library (Appendix A.3.3).

```
pn_org f k = mapSY (psi f) . groupSY k
```

where `f` and `k` can be seen as variables for the function f and the parameter k that are used in the transformation *GroupToMultiRate*. Since Haskell supports the concept of higher-order functions, the transformation can be expressed by means of

```
groupToMultiRate f k = pn_ref
  where
    pn_ref = upDI k . mapSY (psi f) . downDI k . groupSY k
```

Given a simple process network

$$PN = mapSY(\Psi(sumV)) \circ G(k)$$

where

$$sumV(v) = \begin{cases} 0 & \text{if } v = \langle\rangle \\ v_1 + \cdots + v_n & \text{if } v = \langle v_1, \ldots, v_n \rangle \end{cases}$$

The transformed network can now be expressed in Haskell for $k = 3$ as

```
groupToMultiRate sumV 3
```

Simulation shows that both process networks PN_{org} and PN_{ref} have as expected the same behavior ($k = 3$). For these examples a signal

$$\overrightarrow{s} = \ll 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 \gg$$

is used.

```
Hugs98Prompt> pn_org sumV 3 s
{0,_,_,6,_,_,15,_,_,24,_,_}
Hugs98Prompt> groupToMultiRate sumV 3 s
{0,_,_,6,_,_,15,_,_,24,_,_}
```

However, there is a difference in the types of the results, since Haskell infers an double extended data type as output. This results from the fact that `upDI` extends the extended data type a second time, due to the type declaration of `upDI`.

```
upDI :: Num a => a -> Signal b -> Signal (AbstExt b)
```

Haskell logically infers the data type `Signal (AbstExt (AbstExt Integer))`, since it lacks the knowledge that a double extended data type is not meaningful here. However this phenomena refers to the type system of Haskell and is not a principle problem of ForSyDe.

There is also the possibility to define a transformation that works in the same way as *GroupToMultiRate*, but without the up-sample process in the last stage of

the transformed process network. Such a transformation would be very useful since very often there is no need to up-sample the frequency data rate after processes like DFT or FFT. Such a transformation would of course introduce a design decision and may also affect other parts of the system. In the case of the equalizer it would require another transformation in order to introduce an up-sample domain interface, since the internal feedback loop would otherwise lead to signal rate inconsistencies.

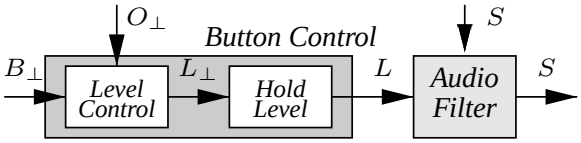
5.4.2 Communication Refinement

The specification model uses the same synchronous communication mechanism between all its subsystems. This is a nice feature for modeling and analyzing, since partitioning issues and special interfaces between subsystems have not to be taken into account in this phase. However, large systems are usually not implemented as one single unit, but are partitioned into hardware and software blocks communicating with each other via a dedicated communication protocol. The ForSyDe methodology offers transformations of a synchronous communication into other protocols. Looking at the equalizer example, the rate of present values of the input and output signals of the *Button Control* and the *Distortion Control* subsystems is considerably lower than the rate for the corresponding signals of the *Audio Filter* and *Audio Analyzer*. In the following, it is assumed that the *Button Control* and *Distortion Control* shall be implemented in software and the *Audio Filter* and *Audio Analyzer* in hardware. For the communication between these parts a handshaking protocol with $FIFO$, *Send* and *Receive* processes is chosen.

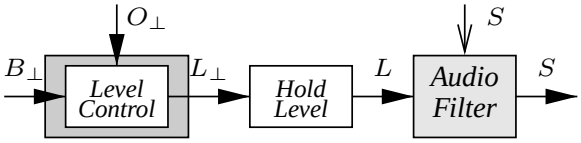
The refinement of the synchronous interface between the *Button Control* and the *Audio Filter* subsystems is shown in Figure 5.28. The figure emphasizes the data types of the signals. Please note that the data types $B_\perp$, $O_\perp$, $L_\perp$ are extended data types, containing absent values. Such signals have a lower rate of present values as the corresponding signals with the data types B, O, L since they do not carry a value in each event cycle. The process *Hold Level*[4] is used to convert an absent extended signal into a with no absent values by outputting the last present value, when receiving an absent value.

The refinement is done in two steps. First, the process *Hold Level* is moved out of the subsystem *Button Control* in order to implement the interface between the process *Level Control* and the process *Hold Level*, since this communication channel has a significantly lower data rate as expressed by the data type of the signal ($L_\perp$). The second step is to refine the interface into a handshake protocol. This is

[4] *Hold Level* is modeled by means of $holdSY$ (see Figure 3.29)



Step 1:Move Subsystem Borders



Step 2: Interface Refinement

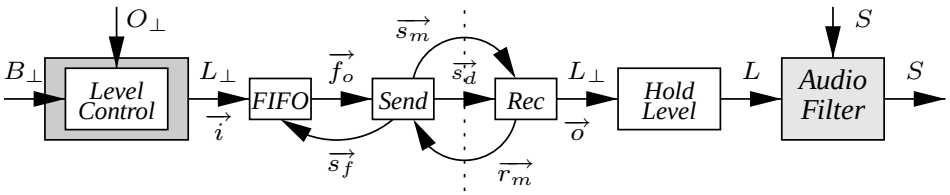


Figure 5.28. Refinement into a handshake protocol

done by the transformation of the channel between *Level Control* and *Hold Level* by means of the transformation *ChannelToHandshake*, which is given in Figure 5.29.

The transformation introduces a $FIFO$, a *Send* and a *Receive* process. When *Send* is idle, it tries to read data from the $FIFO$ on $\overrightarrow{s_f}$. Then it sends the message `DataReady` on $\overrightarrow{s_m}$ to the *Receiver* and after receiving the message `Ready` on $\overrightarrow{r_m}$, it sends the data on $\overrightarrow{s_d}$. The *Receiver* sends a message `Ack` on $\overrightarrow{r_m}$, when the data is received.

The handshake protocol implies a delay of several cycles for each event, as *Send* and *Receive* are synchronous processes. This means, that the timing behavior of the refined interface is different from the original interface. This does also mean, that the *Audio Filter* will not process exactly the same combination of values in each event cycle as in the system model.

These consequences have to be taken into account, when interfaces are refined. In this case, it can be shown that the refined interface still behaves in practice as the system model, if two assumptions are made.

1. The average data rate of the process *Level Control* is much lower than the data rate of the *Audio Filter*. If the FIFO is correctly dimensioned there will

Transformation Rule: $ChannelToHandshake$
Orignal Process Network:
$\overrightarrow{o} = \overrightarrow{i}$ where $sigtype(\overrightarrow{o}) = sigtype(\overrightarrow{i}) = V_\perp$
Transformed Process Network:
$\overrightarrow{o} = PN'(\overrightarrow{i})$
where

$$\begin{array}{lcl} (\overrightarrow{o}, \overrightarrow{r_m}) & = & receive(\overrightarrow{s_m}, \overrightarrow{s_d}) \\ (\overrightarrow{s_m}, \overrightarrow{s_d}, \overrightarrow{s_f}) & = & send(\overrightarrow{f_o}, \overrightarrow{r_m}) \\ \overrightarrow{f_o} & = & fifo(\overrightarrow{i}, \overrightarrow{s_f}) \end{array}$$

Implication:
see text

Figure 5.29. The transformation *ChannelToHandshake*

be no buffer overflow in the FIFO and all values reach the *Audio Control* after a small number of event cycles.

2. The output function of the *Audio Filter* does not change significantly, if the input signals of the *Level Control* are delayed. That is clearly the case, as a small delay of the level signal only delays the change of the amplitude for the same small time, but does not effect the signals shape.

These assumptions point to obligations on other design activities. A further formalization of the design decisions will allow to make all assumptions and obligations explicit. The FIFO buffers have to be dimensioned sufficiently large based on a separate analysis. This will imply a further design decision transformation as illustrated in Figure 5.30. Assumptions about the environment and the application, such as the kind of expected input signal, in this case the data rate, have to be validated to justify the applied design decisions.

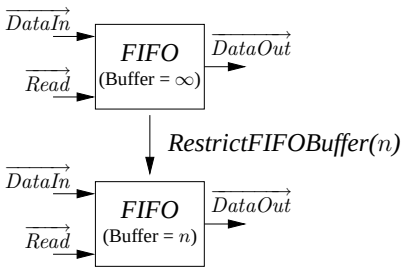


Figure 5.30. Refinement of a FIFO buffer

The interface can be synthesized using the hardware semantics of ForSyDe. The sole purpose of the transformation is to *prepare* for an asynchronous implementation. Note that the model derived is not truly asynchronous in the sense that it is still completely *deterministic* without non-deterministic channel delays. Of course, the channel can be modeled more realistically if desired. The ForSyDe methodology suggests to avoid a non-deterministic model but to use a stochastic channel model instead, which is supported by means of stochastic process constructors [59].

5.4.3 Resource Sharing

Figure 5.31 shows the application of the design decision *SerialClockDomain* to the FIR-filter of Figure 4.7. Since the process $ipV(h)$ is defined as

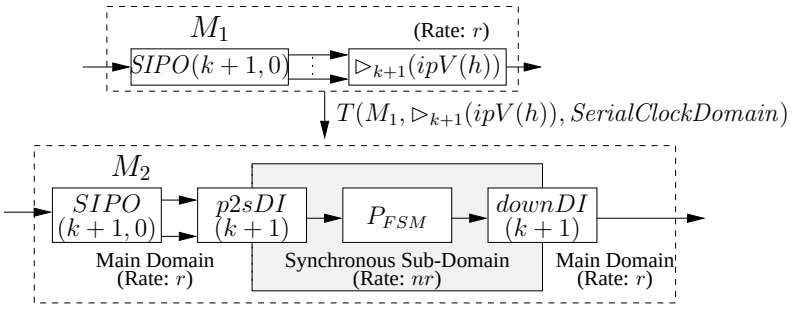


Figure 5.31. Transformation of a FIR-filter

$$(ipV(h))(x_0, \ldots, x_n) = h_0x_0 + \cdots + h_nx_n$$

it complies to the Input Process Network format of the transformation rule *SerialClockDomain*, where

$$\begin{aligned} g_i(x,y) &= x + y \\ h_i(x) &= h_ix \end{aligned}$$

This rule can be used to apply the transformation

$$T(M_1, zipWithSY_{k+1}(ipV(h)), SerialClockDomain)$$

to the FIR-filter model M_1 in order to receive a model M_2, where $SIPO(k+1)$ remains unchanged and the FIR-filter is realized with two clock domains and only one multiplier and one adder (Figure 5.31). The original and transformed FIR-filter have been translated to VHDL and synthesized [73] using the mapping procedure given in Chapter 6.

5.5 Summary and Discussion

The objective of transformational design refinement in ForSyDe is to convert an abstract and high-level specification model into an implementation model that includes all the necessary details in order to allow for an efficient mapping of the implementation model onto the selected target architecture as elaborated in Chapter 6.

ForSyDe allows the application of design decision rules, i.e. transformation rules that change the semantics of the design. Design decisions are needed in order to transform abstract concepts like infinite buffers or real numbers into representations (finite buffers, fixed-point numbers) that can be mapped efficiently onto an architectural component.

In order to allow for a transparent refinement process, each transformation rule is accompanied with a design implication that informs the designer to what extent the application of the rule changes the semantical meaning of the process network. This chapter presented how the characteristic function of a process network can be used to express the implication of a transformation rule. In addition the potential of transformational refinement has been illustrated by the application of powerful transformations to the digital equalizer model.

A transformational refinement approach leads to a well-documented and structured design refinement, where each refinement step is given by the application of a transformation rule. Thus the designer can trace the entire refinement process and may select other transformation options at any point in the refinement process, if the final result was not satisfactory.

As discussed in Section 2.4 a transformational refinement method requires a sufficient amount of transformation rules and an efficient transformation strategy in order to yield good solutions. A prerequisite for a transformation strategy is the availability of cost measures that allow it to indicate, if one model can be implemented more efficiently on a given architecture than another. Such cost measures could be given by an estimation tool that provides estimates in terms of speed, area or power for a particular model implemented on a given architecture.

The transformation problem can be compared with chess. Each transformation rule corresponds to a chess move and each estimation corresponds to the evaluation of a position on the chess board. This analogy also indicates how complex the transformation task is. In chess the average number of possible moves in a typical position is around 30 to 40 and the majority of chess games finishes after no more than 50 moves of white and 50 moves of black. Although there is quite a large market for commercial chess programs, these programs - running on faster and faster

hardware - have not been able to show their superiority to human chess players[5]. The situation in chess can be transfered to system design, although the problem in system design is even more complicated due to the large number of possible moves (transformations) in systems of increasing complexity. While a tool has to check all possible transformations, an experienced designer often knows which transformations may give an improvement of the design.

At present ForSyDe defines only a limited amount of transformations, but this should be sufficient to indicate the potential of ForSyDe. In order to make ForSyDe applicable for larger problems, more transformations have to be developed and an estimation tool has to be incorporated. In order to use the experience of the designer, the future transformation process should be semi-automatic. A tool should provide the designer with information about possible transformations, their implication (defined in the transformation rule) and their possible design improvement. The designer selects transformations and may also go back one or more steps in order to find an efficient solution.

Since the architecture of embedded systems is heterogeneous and extremely complex, it is likely that ForSyDe will initially only be used for dedicated parts in an embedded system. Possible areas are communication refinement for selected protocols and communication patterns, e.g. a GALS structure using handshake protocols, or the refinement of a data-flow path for given architectures.

[5] In 2002 the chess program ”Deep Fritz” drew the match (4-4) against Vladimir Kramnik who is number 2 of the world chess rating list, and in 2003 ”Deep Junior” drew the match (3-3) against Garry Kasparov (number 1 in the rating list).

Chapter 6

Implementation Mapping

This chapter describes how a ForSyDe implementation model is mapped onto a pure hardware implementation. The mapping procedure converts the implementation model into a VHDL-description. This description can be synthesized into a netlist of gates for a given ASIC-technology using commercial logic synthesis tools. In order to obtain an effective solution, the implementation model must already be optimized for the given architecture, since the mapping process does not focus on optimization. A discussion on the mapping of an implementation model to other architectures concludes the chapter.

6.1 Introduction

During *implementation mapping* the implementation model is mapped onto a given architecture. For system-on-chip designs these architectures can consist of many different components, such as micro-controller cores, digital signal processors, memories, switches and routers or custom logic. A large part of a system will be implemented in software that may run on different operating systems. Several IP[1]-blocks may be part of the design.

Thus implementation mapping is very complex. It involves the subtasks of partitioning, allocation, scheduling and mapping, which are research areas in hardware/software co-synthesis as discussed in Section 2.3.1.

[1]Intellectual property (IP) refers to the creation of mind. Here an IP-block is a component that can be acquired from an IP-vendor. An example for such a component is a microprocessor core that can be placed on a system on a chip.

To date ForSyDe defines a mapping to hardware (VHDL) and sequential software (C/C++). This thesis focuses on the mapping to hardware, while the mapping to software is described in [73] and [72]. At this state ForSyDe does not support hardware/software partitioning. This task is left to the designer.

6.2 Mapping of the implementation model to VHDL

ForSyDe defines mapping rules from the implementation model to VHDL. In order to get a good implementation, the implementation model must include the necessary details that allow to map it into efficient synthesizable VHDL-code. The implementation model can be viewed as the design entry to RTL[2] synthesis and compared to other RTL-descriptions like RTL-VHDL or Verilog. The better the implementation model can be mapped onto a VHDL-description for which an efficient implementation exists, the more efficient is the hardware implementation.

The concept of process constructors allows to define these efficient mappings. For each process based on process constructors there exists a mapping onto a VHDL-component or a netlist of VHDL-components that allows an efficient hardware implementation. In addition the mapping procedure comprises other activities such as the mapping of functions, data types or process networks. The procedure for the mapping activities is given below.

1. Generation of a VHDL-description for processes defined by a single process constructor
    - Selection of a VHDL-template according to the process constructor
    - Mapping of data types
    - Mapping of process parameters and constants
    - Translation of functions to VHDL
2. Generation of a VHDL-description for process networks

The mapping procedure is illustrated by two examples. Section 6.2.1 uses the process $DistortionControl$ from Section 4.2.2 to explain how VHDL-descriptions are generated for processes that are based on a single process constructor. Processes that are based on other process constructors are translated in an analog way into VHDL. Appendix C gives VHDL-templates for synchronous processes and

[2] Register Transfer Level

domain interface constructors. Section 6.2.2 uses the ForSyDe model for the handshake protocol from Section 5.4.2 to show how VHDL-descriptions are generated from a process network.

6.2.1 Generation of a VHDL-description for a process that is defined by a process constructor

The model of the process $DistortionControl$ was discussed in Section 4.2.2. The ForSyDe model of this process is given here once again in order to illustrate the generation of VHDL-descriptions for a process based on a process constructor.

$$
\begin{aligned}
&DistortionControl = mealySY_1(ns, out, (\mathsf{Passed}, lim))\\
&\text{where}\\
&ns((st, cnt), inp)\\
&\quad = \begin{cases}
(st, cnt) & \text{if } inp = \perp\\
(\mathsf{Passed}, cnt) & \text{if } st = \mathsf{Passed} \wedge inp = \mathsf{Pass}\\
(\mathsf{Failed}, lim) & \text{if } st = \mathsf{Passed} \wedge inp = \mathsf{Fail}\\
(\mathsf{Locked}, cnt) & \text{if } st = \mathsf{Failed} \wedge inp = \mathsf{Pass}\\
(\mathsf{Failed}, cnt) & \text{if } st = \mathsf{Failed} \wedge inp = \mathsf{Fail}\\
(\mathsf{Failed}, lim) & \text{if } st = \mathsf{Locked} \wedge inp = \mathsf{Fail}\\
(\mathsf{Passed}, cnt - 1) & \text{if } st = \mathsf{Locked} \wedge inp = \mathsf{Pass} \wedge cnt = 1\\
(\mathsf{Locked}, cnt - 1) & \text{if } st = \mathsf{Locked} \wedge inp = \mathsf{Pass} \wedge cnt \neq 1
\end{cases}\\
&out((st, cnt), inp)\\
&\quad = \begin{cases}
\mathsf{Lock} & \text{if } st = \mathsf{Passed} \wedge inp = \mathsf{Fail}\\
\mathsf{CutBass} & \text{if } st = \mathsf{Failed} \wedge inp = \mathsf{Fail}\\
\mathsf{Release} & \text{if } st = \mathsf{Locked} \wedge inp = \mathsf{Pass} \wedge cnt = 1\\
\perp & \text{otherwise}
\end{cases}\\
&lim = 3
\end{aligned}
$$

The first step is to select the appropriate VHDL-template for the process constructor of the process $DistortionControl$. Since the process is defined as

$$DistortionControl = mealySY_1(ns, out, (\mathsf{Passed}, lim))$$

the mapping template for a process based on $mealySY_1$ has to be selected. The mapping is visualized in Figure 6.1, which shows that the result of the mapping is the VHDL-description of a Mealy finite state machine. The combinational functions f and g are mapped to the combinational circuits $\mathsf{f_{HW}}$ and $\mathsf{g_{HW}}$, where $\mathsf{f_{HW}}$ reads as "the hardware implementation of f". The internal process $delaySY_1(s_0)$

is mapped on a bank of registers, where the reset state is given by s_0. The registers have an asynchronous active low reset.

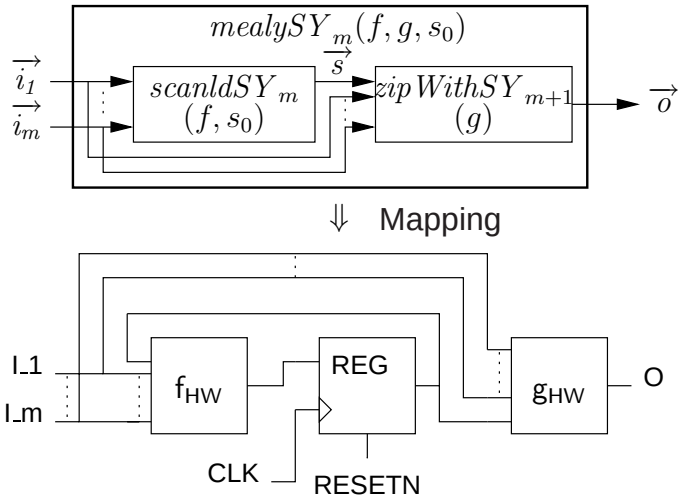


Figure 6.1. Hardware implementation of a process based on $mealySY_1$

The VHDL-template for $mealySY_1(f, g, s_0)$ can be divided into two parts, the *entity part* and the *package part*.

The entity part gives the entity and architecture declaration, while the package part defines the data types, constants and functions that are used in the entity part. The entity part of the VHDL-template for $mealySY_1$ is shown below. The architecture declaration contains two processes. The first process is a sequential process and implements the state registers of the finite state machine. The second process is a combinational process that implements both the next-state decoder (by means of the function `f`) and the output decoder (by means of the function `g`). Please note that the entity part will be basically the same for all processes based on $mealySY_1$. The only difference is that all occurrences of `mealySY_f_g` are replaced by the name of the process, here `DistortionControl`.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.mealySY_f_g_lib.ALL;
LIBRARY synopsys;
USE synopsys.attributes.ALL;

ENTITY mealySY_f_g IS
   PORT (
      i     : IN  type_mealySY_f_g_i;
```

```
      o      : OUT type_mealySY_f_g_o;
      clk    : IN  std_logic;
      resetn : IN  std_logic);
END mealySY_f_g;

ARCHITECTURE Seq OF mealySY_f_g IS
   SIGNAL state, nextstate : type_mealySY_f_g_state;
   ATTRIBUTE state_vector        : string;
   ATTRIBUTE state_vector OF Seq : ARCHITECTURE IS "state";
BEGIN  -- Seq
   PROCESS (clk, resetn)
   BEGIN  -- PROCESS
      IF resetn = '0' THEN -- asynchronous reset (active low)
         state <= s0;
      ELSIF clk'event AND clk = '1' THEN -- rising clock edge
         state <= nextstate;
      END IF;
   END PROCESS;

   PROCESS (i,state)
   BEGIN  -- PROCESS
      nextstate <= f(i,state);
      o <= g(i,state);
   END PROCESS;
END Seq;
```

The package part of the VHDL-template of $mealySY_1$ is shown next. This part defines the data types, functions and constants for the process. During the mapping procedure, all occurrences of `mealySY_f_g` are exchanged with the name of the process, here $DistortionControl$, and all occurrences of `to_be_defined` are replaced by the corresponding VHDL expression for each ForSyDe data type, constant or function.

```
PACKAGE mealySY_f_g_lib IS
   TYPE type_mealySY_f_g_i IS to_be_defined; -- Type of i
   TYPE type_mealySY_f_g_o is to_be_defined; -- Type of o
   TYPE type_mealySY_f_g_state IS to_be_defined;
                                        -- Type of state
   CONSTANT s0 : type_mealySY_f_g_state := to_be_defined;
                                        -- Initial Value
   FUNCTION f (
      i     : type_mealySY_f_g_i;
      state : type_mealySY_f_g_state
      ) RETURN type_mealySY_f_g_state;

   FUNCTION g (
      i     : type_mealySY_f_g_i;
```

```
      state : type_mealySY_f_g_state
      ) RETURN type_mealySY_f_g_state;
END;

PACKAGE BODY mealySY_f_g_lib IS
   FUNCTION f (
      i     : type_mealySY_f_g_i;
      state : type_mealySY_f_g_state
      ) RETURN type_mealySY_f_g_o IS
   BEGIN
      RETURN to_be_defined; -- Definition of f;
   END;

   FUNCTION g(
      i     : type_mealySY_f_g_i;
      state : type_mealySY_f_g_state
      ) RETURN type_mealySY_f_g_o IS
   BEGIN
      RETURN to_be_defined; -- Definition of g;
   END;
END mealySY_f_g_lib;
```

The next step in the mapping process is to map the ForSyDe data types to VHDL. Chapter 3 did not explicitly define data types for ForSyDe. However it is assumed, that the notation can be easily extended to include data types, like in the Haskell representation of ForSyDe models.

The values of the input signals are of an extended data type (they include the value $\perp$). Extended data types can be mapped to a VHDL data type in two ways. The first possibility is to model these data types as a `Record` of two variables. The first entry `isPresent` defines, if the value is present, while the second entry `value` gives the value, if present. Following this approach, the extended data type

$$D_\perp = \{\perp, v_1, v_2, v_3\}$$

is mapped to

```
TYPE type_D_Abst IS RECORD
                      isPresent : boolean;
                      value: type_D;
                    END RECORD;
```

The disadvantage of this approach is that it may create unnecessary logic and registers, such as in case of the example above, where `isPresent` has to be represented by one bit and `value` by another two bits, though only two bits are needed for $D_\perp$.

This leads to the second approach, which can be used efficiently for enumeration types. Here the absent value $\perp$ is treated as an additional value of the enumeration type, which leads to the following type definition.

```
TYPE type_D_Abst IS (Abst, v1, v2, v3);
```

However, this technique can not be used for extended integers or real data types and thus in order to keep the mapping process simple, this thesis does only use the mapping technique based on records.

In this example the data type of the values of the input signal $\overrightarrow{i}$ is an extended data type with the values $\perp$, Pass and Fail and thus mapped to

```
TYPE type_i_value IS (Pass, Fail);
TYPE type_DistortionControl_i IS RECORD
                                   isPresent : boolean;
                                   value : type_i_value;
                                END RECORD;
```

Also the values of the output signal have an extended data type, which is mapped on

```
TYPE type_o_value IS (Lock, CutBass, Release);
TYPE type_DistortionControl_o IS RECORD
                                    isPresent : boolean;
                                    value     : type_o_value;
                                 END RECORD;
```

The internal state of the process $DistortionControl$ is a tuple, where the first part st is an enumeration type, while the second part cnt is a constrained integer. A ForSyDe tuple is mapped on a VHDL record, which in this case has two entries, one is an enumeration type and the other one is a constrained integer.

```
TYPE type_st IS (Passed, Locked, Failed);
SUBTYPE type_cnt IS integer RANGE 0 TO 3;
TYPE type_DistortionControl_state IS RECORD
                                        st  : type_st;
                                        cnt : type_cnt;
                                     END RECORD;
```

The process $DistortionControl$ defines also two constants. The constant $s_0 = (\mathsf{Passed}, 0)$ is the initial state of the finite state machine and will be translated as reset state, while $lim = 3$ is an internal parameter. These constants are also defined in the package part.

```
CONSTANT s0  : type_DistortionControl_state := (Passed, 0);
CONSTANT lim : integer RANGE 0 TO 3         := 3;
```

Finally the combinational functions ns and out have to be translated into VHDL. Since ForSyDe demands to use combinational functions and so far restricts them

to non-recursive functions when targeting hardware, the mapping onto a VHDL function is straight forward. In the case of the process $DistortionControl$, both the ForSyDe model of the nextstate and the output decoder are described by if-clauses which naturally translate to if-clauses in VHDL. Below is the VHDL-code for the function ns (implemented as function `f`) which implements the next state decoder of the finite state machine.

```
FUNCTION f (
   i     : type_DistortionControl_i;
   state : type_DistortionControl_state
   ) RETURN type_DistortionControl_state IS
   VARIABLE result : type_DistortionControl_state;
BEGIN
   IF (i.isPresent = false) THEN
      result := (state.st, state.cnt);
   ELSIF (state.st = Passed) AND (i.value = Pass) THEN
      result := (Passed, state.cnt);
   ELSIF (state.st = Passed) AND (i.value = Fail) THEN
      result := (Failed, state.cnt);
   ELSIF (state.st = Failed) AND (i.value = Pass) THEN
      result := (Locked, state.cnt);
   ELSIF (state.st = Failed) AND (i.value = Fail) THEN
      result := (Failed, state.cnt);
   ELSIF (state.st = Locked) AND (i.value = Fail) THEN
      result := (Failed, lim);
   ELSIF (state.st = Locked) AND (i.value = Pass)
                             AND (state.cnt = 1) THEN
      result := (Passed, state.cnt-1);
   ELSIF (state.st = Locked) AND (i.value = Pass)
                             AND (state.cnt /= 1) THEN
      result := (Passed, state.cnt-1);
   END IF;
   RETURN result;
END;
```

and the function out (implemented as function `g`) which implements the output decoder.

```
FUNCTION g(
   i     : type_DistortionControl_i;
   state : type_DistortionControl_state
   ) RETURN type_DistortionControl_o IS
   VARIABLE result : type_DistortionControl_o;
BEGIN
   IF (state.st = Passed) AND (i.value = Pass) THEN
      result := (true, Lock);
   ELSIF (state.st = Failed) AND (i.value = Fail) THEN
      result := (true, CutBass);
```

```
   ELSIF (state.st = Locked) AND (i.value = Pass)
                             AND (state.cnt = 1) THEN
      result := (true, Release);
   ELSE
      result := (false, Lock);
   END IF;
   RETURN result;
END;
```

The VHDL implementation of the process $DistortionControl$ is synthesizable. The synthesis result of a synthesis process for the LSI 10K technology[3] and is given as schematic in Figure 6.2.

6.2.2 Generation of a VHDL-description for a process network

The refinement of a signal with values of an extended data type into a handshake protocol has been discussed in detail in Section 5.4.2. It is illustrated in Figure 5.28. The Figure did not give all details of the processes involved in the handshaking protocol. In particular, it did not show the delay processes, which must exist in the ForSyDe model in order to prevent zero-delay feedback loops. The ForSyDe model for the handshake protocol is given below.

$$\overrightarrow{o} = handshake(\overrightarrow{i})$$

where

$$\begin{aligned} (\overrightarrow{o}, \overrightarrow{r_m}) &= receive(\overrightarrow{s_m}, \overrightarrow{s_d}) \\ (\overrightarrow{s_m}, \overrightarrow{s_d}, \overrightarrow{s_f}) &= send(\overrightarrow{f_o}, \overrightarrow{r_m}) \\ \overrightarrow{f_o} &= fifo(\overrightarrow{i}, \overrightarrow{s_f}) \end{aligned}$$

Figure 6.3 shows the delay processes in the ForSyDe model of the handshake protocol explicitly. All delay processes are located inside the process $Send$. It also shows the mapping of a process network into a VHDL-description of hardware.

A process network is translated into a structural VHDL-description where

- ForSyDe signals are mapped to VHDL signals;
- ForSyDe processes are mapped to VHDL-components.

The following code shows the structural VHDL-description of the handshake protocol. Each ForSyDe process is translated into a component and each ForSyDe

[3]The design has a maximum clock frequency of $f_{max} = 170$ MHz and occupies an area equivalent to 74 NAND-gates.

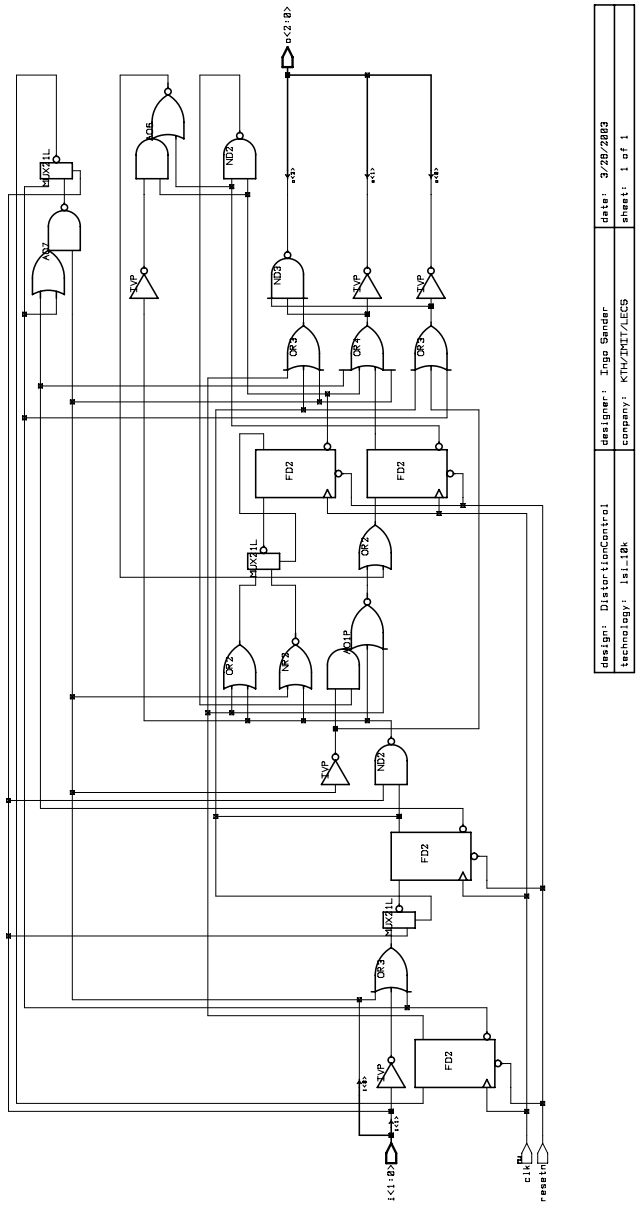


Figure 6.2. The synthesized implementation of the process *DistortionControl*

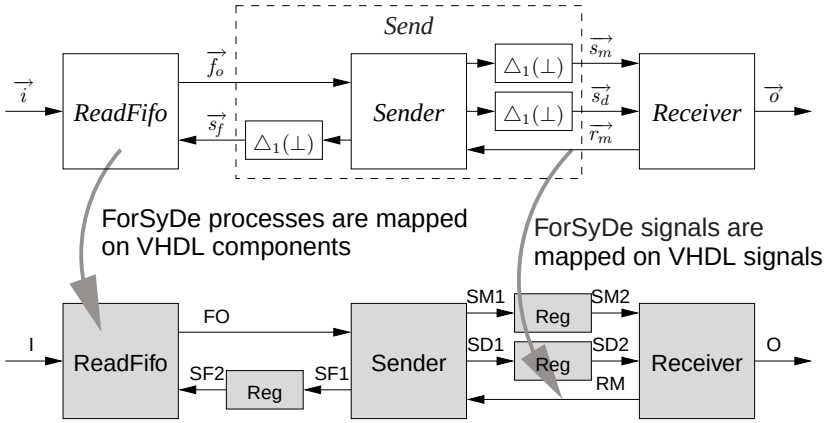


Figure 6.3. Mapping of the handshake protocol to hardware

signal into a VHDL signal. The code is a VHDL netlist of components. The architecture part lists the components and defines internal signals. Then the components are connected to the internal and external signals by `PORT MAP`. Since the output of the components `Sender` and `Receiver` are records of signals, each signal in these records has to be connected to an internal signal as expressed in the last lines of the architecture description.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.Handshake_Pkg.ALL;

ENTITY HandshakeProtocol IS

   PORT (
      i      : IN  type_fifoValues;
      o      : OUT type_fifoValues;
      resetn : IN  std_logic;
      clk    : IN  std_logic);

END HandshakeProtocol;

ARCHITECTURE Structure OF HandshakeProtocol IS

   COMPONENT ReadFifo

      PORT (
         i        : IN  type_fifoValues;
         readFifo : IN  type_readFifo;
         o        : OUT type_fifoValues;
```

```
      clk      : IN  std_logic;
      resetn   : IN  std_logic);

END COMPONENT;

COMPONENT Sender

   PORT (
      fifoOutput : IN  type_fifoValues;
      recMsg     : IN  type_recMsg;
      o          : OUT type_senderOutput;
      clk        : IN  std_logic;
      resetn     : IN  std_logic);

END COMPONENT;

COMPONENT Receiver

   PORT (
      sendMsg  : IN  type_SendMsg;
      sendData : IN  type_fifoValues;
      o        : OUT type_ReceiverOutput;
      clk      : IN  std_logic;
      resetn   : IN  std_logic);

END COMPONENT;

COMPONENT delaySendMsg
   PORT
      (
         i      : IN  type_sendMsg;
         o      : OUT type_sendMsg;
         resetn : IN  std_logic;
         clk    : IN  std_logic
         );
END COMPONENT;

COMPONENT delayReadFifo
   PORT
      (
         i      : IN  type_readFifo;
         o      : OUT type_readFifo;
         resetn : IN  std_logic;
         clk    : IN  std_logic
         );
END COMPONENT;
```

```
   COMPONENT delaySendData
      PORT
         (
            i      : IN  type_fifoValues;
            o      : OUT type_fifoValues;
            resetn : IN  std_logic;
            clk    : IN  std_logic
            );
   END COMPONENT;

   SIGNAL fo       : type_fifoValues;
   SIGNAL sf1, sf2 : type_readFifo;
   SIGNAL sm1, sm2 : type_sendMsg;
   SIGNAL sd1, sd2 : type_fifoValues;
   SIGNAL rm       : type_recMsg;

   SIGNAL senderOutput   : type_senderOutput;
   SIGNAL receiverOutput : type_receiverOutput;

BEGIN
   fifo : ReadFifo
      PORT MAP (
         i        => i,
         readFifo => sf2,
         o        => fo,
         clk      => clk,
         resetn   => resetn);

   send : sender
      PORT MAP (
         fifoOutput => fo,
         recMsg     => rm,
         o          => senderOutput,
         clk        => clk,
         resetn     => resetn);

   rec : receiver
      PORT MAP (
         sendMsg  => sm2,
         sendData => sd2,
         o        => receiverOutput,
         clk      => clk,
         resetn   => resetn);

   delSM : delaySendMsg
      PORT MAP(
         i      => sm1,
```

```
         o      => sm2,
         resetn => resetn,
         clk    => clk);

   delRM : delayReadFifo
      PORT MAP (
         i      => sf1,
         o      => sf2,
         resetn => resetn,
         clk    => clk);

   del : delaySendData
      PORT MAP(
         i      => sd1,
         o      => sd2,
         resetn => resetn,
         clk    => clk);

   sm1 <= senderOutput.sendMsg;
   sd1 <= senderOutput.sendData;
   sf1 <= senderOutput.readFifo;

   rm <= receiverOutput.recMsg;
   o  <= receiverOutput.recData;
END Structure;
```

Since all ForSyDe processes in the handshake protocol are constructed by process constructors, they have been mapped to VHDL using the techniques described in Section 6.2.1. The handshake protocol has been synthesized with the LSI 10K standard cell library as target architecture. The top module of the synthesized description is shown in Figure 6.4. The Figure shows clearly the mapping of ForSyDe processes onto VHDL-components. The synthesis results are given in Table 6.4.

Component	Area	Max. Frequency
ReadFifo	201	100 MHz
Sender	94	130 MHz
Receiver	63	180 MHz
Total Design	421	100 MHz

Table 6.1. Synthesis results for the handshake protocol

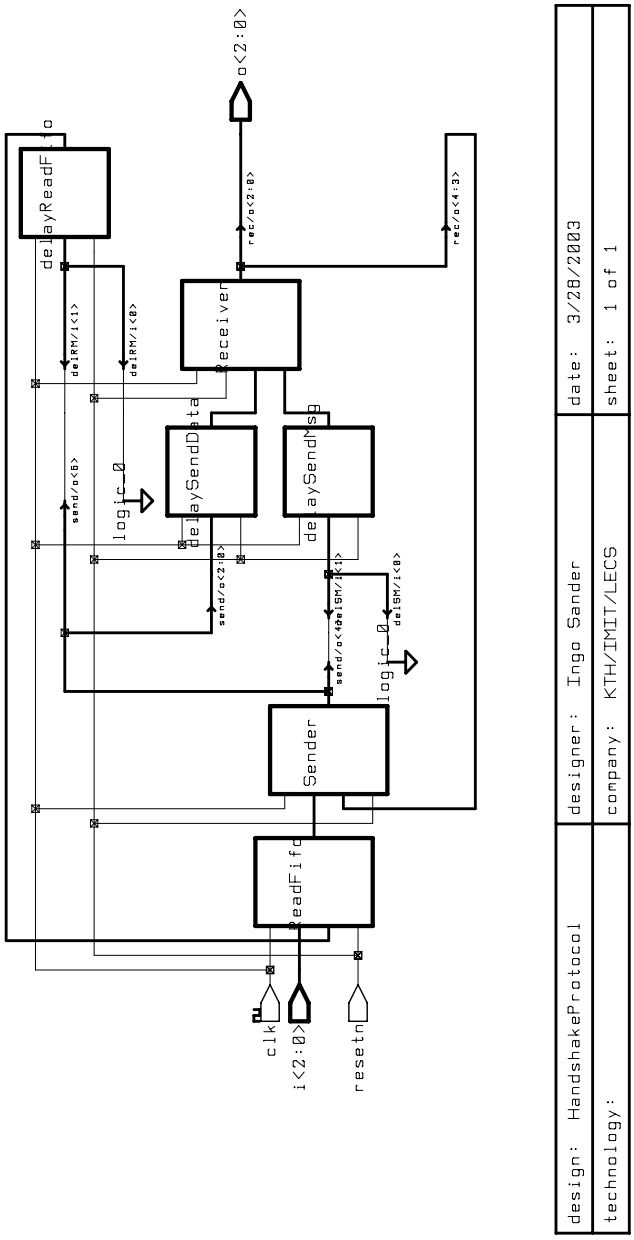


Figure 6.4. The synthesized implementation of the handshake protocol (top module)

6.2.3 The Importance of an optimized Implementation Model

The mapping technique described in the previous sections can be used to translate ForSyDe implementation models into VHDL-descriptions, since for all ForSyDe leaf processes a VHDL mapping template is defined. So far the mapping is done manually, but since VHDL-templates and mapping rules are defined, the mapping process can be automated.

In order to be able to synthesize an implementation model into hardware, it is obvious that the ForSyDe implementation model must be synthesizable, i.e. all ForSyDe constructs must be mappable to synthesizable VHDL-constructs. Thus a FIFO, which contains an infinite buffer is not synthesizable, since there are no infinite data structures in hardware.

In order to get an *efficient* hardware implementation, the ForSyDe implementation model must be optimized, since the translation from ForSyDe to VHDL is a mapping process and does not include any optimization. If the application is speed critical, already the implementation model must be optimized in order to minimize the *critical path*.

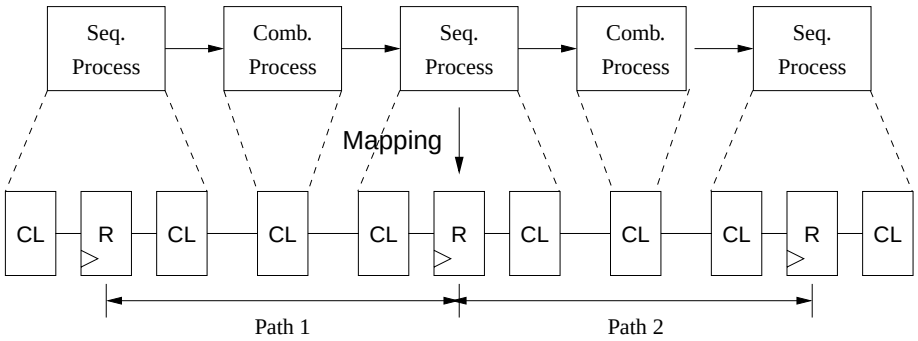


Figure 6.5. The critical path in the implementation model

The critical path is illustrated in Figure 6.5. A ForSyDe model contains combinational and sequential processes. Each sequential process is translated to combinational components and registers. The maximum frequency of the circuit is determined by the path with the longest combinational delay, i.e. the critical path.

Thus it is of crucial importance that the implementation model is already optimized for the critical path. But this means also that the transformational design process must be supported by an estimation tool that allows to give a good estimate for the combinational delay in process networks. However, to date ForSyDe does not have support from an estimation tool.

6.3 Discussion

This chapter describes the mapping of a ForSyDe implementation model into custom hardware. ForSyDe defines a mapping for processes and process networks to VHDL. The VHDL-description is synthesized by commercial synthesis tools into a gate netlist of the chosen target technology. At present the mapping process is done manually, but can be automated.

In order to target more complex embedded systems, mappings from a ForSyDe implementation model to other architectures have to be defined. The paper [73] describes a mapping procedure for sequential software aimed for a single microprocessor and discusses a possible hardware/software implementation of the digital equalizer.

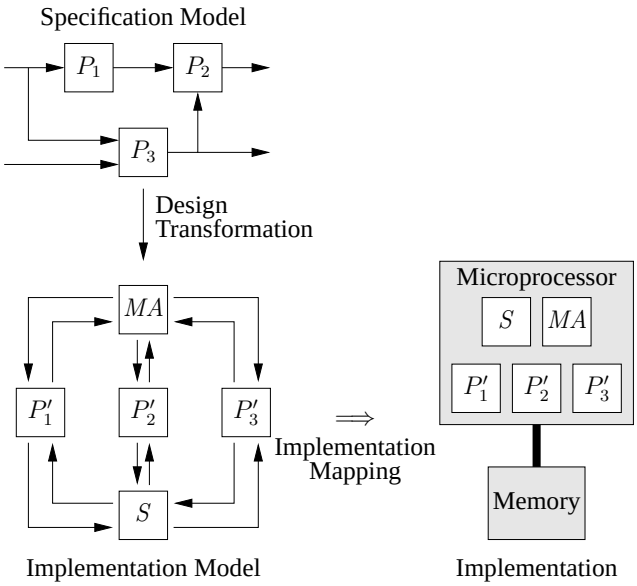


Figure 6.6. Design transformation and implementation mapping for concurrent software running on a single processor with a scheduling process S

However, compared to the mapping onto other architectures used in embedded systems, the mapping to pure hardware and sequential software is almost straightforward. If the target architecture is a single processor, where processes run in parallel and communicate via a shared memory the design task gets more difficult. As illustrated in Figure 6.6 the process network of the specification model, where processes communicate directly with each other via signals, has to be transformed into an implementation model, where a scheduling process S takes care of

the scheduling of processes and where the communication is done indirectly via a memory access process MA. Then each process in the implementation model can be mapped onto software components. If a real-time operating system (RTOS) is used, the scheduling process could be mapped on the RTOS.

This example shows that each architecture does not only need its own mapping procedure, but there must also exist transformation rules that allow to prepare the implementation model for mapping. The development of these transformation and mapping rules is feasible, but will require a lot of effort. As ForSyDe is a research project, transformation and mapping rules may only be defined for a limited amount of architectures.

Chapter 7

Conclusion

This chapter concludes the thesis by giving a summary of the previous chapters, where the key ideas and concepts of ForSyDe are emphasized. Based on the present state of ForSyDe it gives an overview about future directions for research with the objective to use ForSyDe for larger applications.

7.1 Summary of the Thesis

The increasing capacity of integrated circuits makes it possible to integrate more and more functionality on a single chip. SoC architectures may include a variety of components and allow the design of very powerful applications. Ideally a design process for such applications starts with a high-level and abstract model that allows the designer to capture the functionality of the system. On the other hand many low-level details are required to efficiently implement a system on such target architectures. Thus there exists a large abstraction gap between an ideal specification model and an efficient implementation.

The objective of the ForSyDe methodology is to bridge this abstraction gap. ForSyDe has been based on carefully chosen foundations. The specification model uses a synchronous computational model that separates computation from communication and is expressed by a clean and simple mathematical formalism that allows for formal reasoning. In addition a synchronous model implies a total order of events that allows the formulation of abstract timing constraints. Processes are constructed by synchronous process constructors, which implement the synchronous computational model, allow for design transformation, and facilitate the

mapping of process networks to custom hardware or sequential software. The implementation model allows the establishment of synchronous sub-domains inside a multi-rate model. These foundations prepare ForSyDe for the incorporation of formal methods, which will be of increasing importance, since simulation alone will not be able to solve the verification task for systems of increasing size.

The details of the ForSyDe methodology are described in Chapter 3 to 6. The definitions of the specification and implementation model are given in Chapter 3. These models can be expressed and simulated using the functional language Haskell. System modeling in ForSyDe is illustrated by the specification model of the digital equalizer in Chapter 4. Chapter 5 describes transformational design refinement in ForSyDe. This phase uses both semantic preserving transformations and design decisions. The effect of a design decision is given in the implication part of the transformation rule as information for the designer. The characteristic function, which can be derived for any process network in ForSyDe, serves as a useful tool for the development of a transformation rule. The potential of transformational refinement has been illustrated by means of powerful transformations that have been applied to the specification model of the digital equalizer. The last step in the ForSyDe design flow is implementation mapping and is discussed in Chapter 6. So far mapping rules exist for custom hardware and sequential software. The mapping procedure does not include optimization, since the idea of ForSyDe is to move design refinement to higher levels of abstraction. Thus optimization shall be performed during transformational design refinement inside the functional domain of ForSyDe. Only during implementation mapping the design process leaves the functional domain. Here commercial tools, such as software compilers or logic synthesis tools, take over in order to generate a software or hardware implementation.

The main contribution of the thesis is to show how the system design process can be moved to a higher level of abstraction by using a carefully selected model of computation together with the high-level concept of process constructors. Although ForSyDe has not been used for larger examples, the thesis should give an indication that a transformational system design approach at least can be used for application-specific domains in system design.

7.2 Future Work

To date the main concepts of ForSyDe have been formulated and validated by smaller case studies. In order to target more realistic applications, future work

should focus on restricted application domains and target architectures. The following list defines areas for future work on ForSyDe.

- **Incorporation of other models of computation**

  The multi-rate model is very suitable to model applications that shall be implemented in custom hardware. However, if the target architecture is concurrent software that runs on a single processor other computational models like synchronous data flow [67] [68] or communicating sequential processes [47] are better suited and candidates for the integration into the implementation model of ForSyDe.

- **Transformation rules**

  The number of transformation rules in ForSyDe has to be extended and must include data type and memory refinement. The effort should be put on the development of transformation rules for the selected application domain and the selected target architecture.

- **Tool support for design refinement**

  A transformation tool should be developed that informs the designer of the possible transformations, their implications and ideally also the possible improvement in terms of performance measures which could be given by an estimation tool. A starting point could be to incorporate the ULTRA tool [3] into ForSyDe, which allows to apply transformations on Haskell code.

- **Automation of the mapping of a ForSyDe implementation model to VHDL**

  As indicated in this thesis the mapping of a ForSyDe implementation model can be automated. A similar tool could also be developed for the synthesis to software and future architectures.

- **Incorporation of verification methods**

  Since ForSyDe is based on a formal synchronous model, similar verification methods as used for Lustre [43] or Esterel [14] could be incorporated into ForSyDe. Also the Haskell based hardware description Lava [19] includes formal verification methods and it should be investigated to which extent such methods could be used in ForSyDe.

- **Modeling language**

  Industrial designers are in general used to languages like C or VHDL and do not have any experience with a functional language like Haskell. Thus in order to make ForSyDe more attractive for industry, another modeling language like SystemC might be a better practical choice for ForSyDe, though it does not support the concepts of ForSyDe to the same extent as Haskell.

- **Graphical user interface**

  A graphical user interface (GUI) would also increase the acceptance in industry. A GUI could hide many formal concepts of ForSyDe and instead presents them in a way that designers are familiar with. A process constructor $mooreSY$ could be represented by a schematic of a Moore finite state machine, where the designer has to specify the initial state and the functions for the next state and output decoder.

References

[1] The ForSyDe webpage. `http://www.ele.kth.se/ForSyDe`.

[2] The Haskell home page. `http://www.haskell.org`.

[3] The ULTRA home page. `http://www.informatik.uni-ulm.de/pm/ultra/`.

[4] A. Allan, D. Edenfeld, W. H. Joyner Jr., A. B. Kahng, M. Rodgers, and Y. Zorian. 2001 technology roadmap for semiconductors. *IEEE Computer*, January 2002.

[5] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Progamming in ERLANG*. Prentice-Hall, 1996.

[6] F. Balarin, M. Chiodo, P. Giusti, H. Hsieh, A. Jurescka, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara. *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*. Kluwer Academic Publishers, 1997.

[7] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, A. Sangiovanni-Vincentelli, E. M. Sentovich, and K. Suzuki. Synthesis of software programs for embedded control applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(6):834–849, June 1999.

[8] F. L. Bauer, B. Möller, H. Partsch, and P. Pepper. Formal program construction by transformations – computer-aided, intuition guided programming. *IEEE Transactions on Software Engineering*, 15(2), February 1989.

[9] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, September 1991.

[10] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. D. Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, January 2003.

[11] J. Öberg, A. Kumar, and A. Hemani. Grammar-based hardware synthesis from port size independent specifications. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(2):184–194, 2000.

[12] G. Berry. Real time programming: Special purpose or general purpose language. In G. Ritter, editor, *Information Processing 8*, pages 11–18. Elsevier Science Publishers B.V., North-Holland, 1989.

[13] G. Berry. A hardware implementation of pure Esterel. In *Proc. International Workshop on Formal Methods in VLSI Design*, January 1991.

[14] G. Berry. The foundations of Esterel. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 1998.

[15] G. Berry. The constructive semantics of pure Esterel. Draft Version 3, 1999.

[16] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[17] R. S. Bird. *An Introduction to the theory of lists*. Oxford University Computing Laboratory, Technical Monograph PRG-56 edition, 1986.

[18] R. S. Bird. *Lectures on Constructive Functional Programming*. Oxford University Computing Laboratory, Technical Monograph PRG-69 edition, 1988.

[19] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in Haskell. In *International Conference on Functional Programming*, 1998.

[20] P. Bjuréus and A. Jantsch. MASCOT: A specification and cosimulation method integrating data and control flow. In *Proceedings of the Design and Test Europe Conference (DATE)*, 2000.

[21] P. Bjuréus and A. Jantsch. Modeling of mixed control and dataflow systems in MASCOT. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(5):690–703, 2001.

[22] F. Boussinot and R. de Simone. The ESTEREL language. *Proceedings of the IEEE*, 79(9):1293–1304, September 1991.

[23] C. G. Cassandras. *Discrete Event Systems: Modeling and Performance Analysis*. Asken Associates, 1993.

[24] K. Claessen. *Embedded Languages for Describing and Verifying Hardware*. PhD thesis, Chalmers University of Technology and Göteborg University, 2001.

[25] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. Pitman, London, 1989.

[26] R. Dömer, D. Gajski, and A. Gerstlauer. SpecC methodology for high-level modeling. In *9th IEEE/DATC Electronic Design Processes Workshop*, Monterey, California, April 2002.

[27] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli. Design of embedded systems: Formal models, validation, and synthesis. *Proceedings of the IEEE*, 85(3):366–390, March 1997.

[28] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity—the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, January 2003.

[29] J. Ellsberger, D.Hogrefe, and A. Sarma. *SDL - Formal Object Oriented Language for Communicating Systems*. Prentice-Hall, 1997.

[30] R. Ernst. Codesign of embedded systems: Status and trends. *IEEE Design & Test of Computers*, 15(2):45–54, April–June 1998.

[31] R. Ernst, J. Henkel, and T. Brenner. Hardware-software cosynthesis from microcontrollers. *IEEE Design & Test of Computers*, 10(4):64–75, December 1993.

[32] M. J. Flynn. Some computer organisations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, September 1972.

[33] D. D. Gajski, N. D. Dutt, A. C.-H. Wu, and S. Y.-L. Lin. *High-Level Synthesis*. Kluwer Academic Publishers, 1992.

[34] D. D. Gajski and L. Ramachdran. Introduction to high-level synthesis. *IEEE Design & Test of Computers*, 11(4), 1994.

[35] D. D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao. *Spec C: Specification Language and Methodology*. Kluwer Academic Publishers, 2000.

[36] A. Girault, B. Lee, and E. A. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(6):742–760, June 1999.

[37] M. Gordon, R. Milner, L. Morris, M. Newey, and C. P. Wadsworth. A metalanguage for interactive proof in LCF. In *Conference Record of the 5th Annual ACM Symposium on Principles of Programming Languages*, pages 119–130. ACM, 1978.

[38] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[39] P. L. Guernic, T. Gautier, M. L. Borgne, and C. de Marie. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1335, September 1991.

[40] R. K. Gupta. *Co-Synthesis of Hardware and Software for Digital Embedded Systems*. Kluwer Academic Publishers, 1995.

[41] R. K. Gupta and G. D. Micheli. Hardware-software cosynthesis for digital systems. *IEEE Design & Test of Computers*, 10(3):29–41, September 1993.

[42] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993.

[43] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[44] N. Halbwachs and P. Raymond. Validation of synchronous reactive systems: from formal verification to automatic testing. In *ASIAN'99, Asian Computing Science Conference*, Phuket (Thailand), December 1999. LNCS 1742, Springer Verlag.

[45] D. C. Hanselman and B. Littlefield. *Mastering MATLAB 5: A comprehensive Tutorial and Reference*. Prentice-Hall, 1998.

[46] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[47] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8), 1978.

[48] H. Hsieh, F. Balarin, L. Lavagno, and A. Sangiovanni-Vincentelli. Synchronous approach to functional equivalence of embedded system implementations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 20(8):1016–1033, August 2001.

[49] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, September 1989.

[50] P. Hudak and M. Jones. Haskell vs Ada vs C++ vs Awk vs ...: An experiment in software prototyping productivity. Technical report, Dept. of Computer Science, Yale University, July 1994.

[51] P. Hudak, J. Peterson, and J. H. Fasel. *A Gentle Introduction to Haskell 98*. October 1999. `http://www.haskell.org/tutorial`.

[52] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[53] IEEE. *IEEE Standard for Verilog Hardware Description Language*. IEEE, 2001.

[54] IEEE. *IEEE Standard VHDL Language Reference Manual*. IEEE, 2002.

[55] J. D. II, M. Goel, C. Hylands, B. Kienhuis, E. A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay, and Y. Xiong. Overview of the Ptolemy project. Technical Report UCB/ERL No. M99/37, Dept. EECS, University of California, Berkeley, CA 94720, 1999.

[56] A. Jantsch and P. Bjuréus. Composite signal flow: A computational model combining events, sampled streams, and vectors. In *Proceedings of the Design and Test Europe Conference (DATE)*, 2000.

[57] A. Jantsch, S. Kumar, I. Sander, B. Svantesson, J. Öberg, A. Hemani, P. Ellervee, and M. O'Nils. Comparison of six languages for system level descriptions of telecom systems. In *First International Forum on Design*

Languages - Proceedings, volume 2, pages 139–148, Lausanne, Switzerland, September 1998.

[58] A. Jantsch and I. Sander. On the roles of functions and objects in system specification. In *Proceedings of the International Workshop on Hardware/-Software Codesign*, 2000.

[59] A. Jantsch, I. Sander, and W. Wu. The usage of stochastic processes in embedded system specifications. In *Proceedings of the Ninth International Symposium on Hardware/Software Codesign*, April 2001.

[60] A. Jantsch and H. Tenhunen, editors. *Networks on Chip*. Kluwer Academic Publishers, 2003.

[61] G. Jones and M. Sheeran. Circuit design in Ruby. In *Formal Methods for VLSI Design*, pages 13–70. North-Holland, 1990.

[62] S. P. Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, 2003.

[63] G. Kahn. The semantics of a simple language for parallel programming. In *Proceedings IFIP Congress '74*. North-Holland, 1974.

[64] G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. In *IFIP '77*. North-Holland, 1977.

[65] K. Keutzer, S. Malik, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: Orthogonolization of concerns and platform-based design. *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, 19(12):1523–1543, December 2000.

[66] E. A. Lee. What's ahead for embedded software. *IEEE Computer*, 33(9):18–26, September 2000.

[67] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, C-36(1):24–35, January 1987.

[68] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

[69] E. A. Lee and T. M. Parks. Dataflow process networks. *IEEE Proceedings*, 1995.

[70] E. A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, December 1998.

[71] Y. Li and M. Leeser. HML, a novel hardware description language and its translation to VHDL. *IEEE Transactions on VLSI*, 8(1):1–8, February 2000.

[72] Z. Lu. Refinement of a system specification for a digital equalizer into HW and SW implementations. Master's thesis, Department of Microelectronics and Information Technology, Royal Institute of Technology, December 2001. IMIT/2001-18.

[73] Z. Lu, I. Sander, and A. Jantsch. A case study of hardware and software synthesis in ForSyDe. In *Proceedings of the 15th International Symposium on System Synthesis*, Kyoto, Japan, October 2002.

[74] W. Luk and T. Wu. Towards a declarative framework for hardware-software codesign. In *Proc. Third International Workshop on Hardware/Software Codesign*, pages 181–188, 1994.

[75] F. Maraninchi. The Argos language: Graphical representation of automata and description of reactive systems. In *IEEE Workshop on Visual Languages*, oct 1991.

[76] J. Matthews, B. Cook, and J. Launchbury. Microprocessor specification in HAWK. In *International Conference on Computer Languages*, 1998.

[77] G. D. Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[78] G. D. Micheli, R. Ernst, and W. Wolf. *Readings in Hardware/Software Co-Design*. Morgan Kaufmann Publishers, 2002.

[79] G. D. Micheli and R. K. Gupta. Hardware/software co-design. *Proceedings of the IEEE*, 85(3):349–365, March 1997.

[80] A. Mihal, C. Kulkarni, M. Moskewicz, M. tsai, N. Shah, S. Weber, Y. Jin, K. Keutzer, C. Sauer, K. Vissers, and S. Malik. Developing architectural platforms: A disciplined approach. *IEEE Design & Test of Computers*, 19(6):6–16, November-December 2002.

[81] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 1978.

[82] R. Milner. A calculus of communicating systems. *LNCS*, 92, 1980.

[83] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[84] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML - Revised*. MIT Press, 1997.

[85] G. E. Moore. Cramming more components onto integrated circuits. *Electronis*, 38(8), 1965.

[86] A. Mycroft and R. Sharp. Hardware/software co-design using functional languages. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2031 of *LNCS*. Springer-Verlag, 2000.

[87] J. O'Donnell. From transistors to computer architecture: Teaching functional circuit specification in Hydra. In Springer-Verlag, editor, *Symposium on Functional Programming Languages in Education*, volume 1022 of *LNCS*, pages 195–214, 1995.

[88] H. A. Partsch. *Specification and Transformation of Programs*. Springer-Verlag, 1990.

[89] A. Pettorossi and M. Proietti. Rules and strategies for transforming functional and logic programs. *ACM Computing Surveys*, 28(2):361–414, June 1996.

[90] J. Plosila. *Self-Timed Circuit Design - The Action Systems Approach*. PhD thesis, University of Turku, Finland, 1999.

[91] J. G. Proakis and D. G. Manolakis. *Digital Signal Processing*. Prentice Hall, 3 edition, 1996.

[92] H. J. Reekie. *Realtime Signal Processing*. PhD thesis, University of Technology at Sydney, Australia, 1995.

[93] F. Rocheteau and N. Halbwachs. Implementing reactive programs on circuits: A hardware implementation of lustre. In *REX Workshop Proceedings*, June 1992.

[94] I. Sander and A. Jantsch. Formal system design based on the synchrony hypothesis. In *Proceedings of the 12th international conference on VLSI Design*, pages 318–323, Goa, India, January 1999.

[95] I. Sander and A. Jantsch. System synthesis based on a formal computational model and skeletons. In *Proceedings IEEE Workshop on VLSI'99*, pages 32–39, Orlando, Florida, April 1999. IEEE Computer Society.

[96] I. Sander and A. Jantsch. System synthesis utilizing a layered functional model. In *Proceedings Seventh International Workshop on Hardware/Software Codesign*, pages 136–140, Rome, Italy, May 1999. ACM Press.

[97] I. Sander and A. Jantsch. Transformation based communication and clock domain refinement for system design. In *39th Design Automation Conference (DAC 2002)*, New Orleans, USA, June 2002.

[98] I. Sander and A. Jantsch. System modeling and transformational design refinement in ForSyDe. Submitted to the Journal *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2003.

[99] I. Sander, A. Jantsch, and Z. Lu. Development and application of design transformations in ForSyDe. In *Design, Automation and Test in Europe Conference (DATE 2003)*, pages 364–369, Munich, Germany, March 2003.

[100] T. Seceleanu. *Systematic Design of Synchronous Digital Circuits*. PhD thesis, University of Turku, Finland, 2001.

[101] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-Vincentelli. Addressing the system-on-a-chip interconnect woes through communication based design. In *Design Automation Conference (DAC 2001)*, Las Vegas, Nevada, USA, June 2001.

[102] R. Sharp and A. Mycroft. A higher level language for hardware synthesis. In *Proceedings of 11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, volume 2144 of *LNCS*. Springer-Verlag, 2001.

[103] S. Singh. System level specifcation in Lava. In *Design, Automation and Test in Europe Conference (DATE 2003)*, pages 370–375, March 2003.

[104] D. Skillicorn. *Foundations of Parallel Programming*. Cambridge University Press, 1994.

[105] D. B. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123–169, June 1998.

[106] K. Strehl, L. Thiele, M. Gries, D. Ziegenbein, R. Ernst, and J. Teich. FunState - an internal design representation for codesign. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(4):524–544, August 2001.

[107] B. Stroustrup. *The C++ Programming Language*. Addison Wesley, 2000.

[108] S. Thompson. *Haskell - The Craft of Functional Programming*. Addison-Wesley, 2 edition, 1999.

[109] J. Voeten. On the fundamental limitataions of transformational design. *ACM Transactions on Design Automation of Electronic Systems*, 6(4):553–552, October 2001.

[110] Z. Wan. Functional reactive programming from first principles. In *Programming Language Design and Implementation (PLDI '00)*, 2000.

[111] Z. Wan, W. Taha, and P. Hudak. Real-time FRP. In *International Conference on Functional Programming (ICFP '01)*, 2001.

[112] Z. Wan, W. Taha, and P. Hudak. Event-driven FRP. In *Fourth International Symposium on Practical Aspects of Declarative Languages (PADL '02)*, 2002.

[113] G. Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.

[114] W. H. Wolf. Hardware-software co-design of embedded systems. *Proceedings of the IEEE*, 82(7):967–989, July 1994.

[115] W. Wu, I. Sander, and A. Jantsch. Transformational system design based on a formal computational model and skeletons. In *Forum on Design Languages 2000*, Tübingen, Germany, September 2000.

[116] D. Ziegenbein, K. Richter, R. Ernst, L. Thiele, and J. Teich. SPI—a system model for heterogeneously specified embedded systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 10(4):379–389, August 2002.

Appendix A

The ForSyDe Standard Library

A.1 Introduction

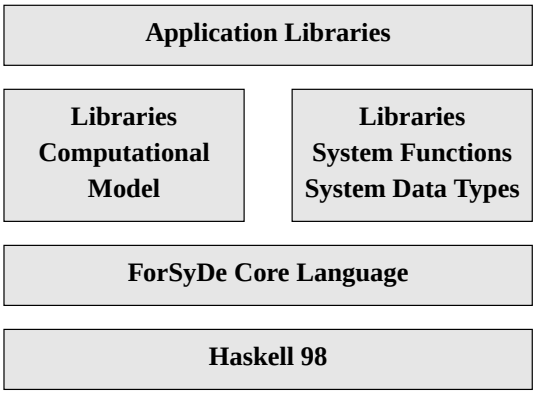


Figure A.1. The ForSyDe Standard Library

The ForSyDe Standard Library consists of several layers as illustrated in Figure A.1. The bottom layer is the Haskell 98 language [62]. The layer above Haskell 98 defines the *ForSyDe Core Language*. Here the fundamental data types, such as signal and vector, and the corresponding functions are defined. Computational models are defined in a *Computational Model Library* and are located on top of the core language. Also on top of the core language there are the *Libraries of System Functions and Data Types*, which contains functions and data types that are typical for system applications and are independent of the computational model. The top layer of the ForSyDe Standard Library consists of *Application Libraries*.

These libraries include components and functions that are modeled for specific computational models.

The ForSyDe Standard Library can be imported with

```
import ForSyDeStdLib
```

which includes all sub-modules of the library.

This appendix covers only the parts of the library that are used in this thesis. For preliminary versions of other computational models or the stochastic library see the ForSyDe web page [1].

The code is written in literate Haskell style, which makes it possible to include LaTeX code for documentation. Only those parts of the literate program that are entirely enclosed between `\begin{code} ... \end{code}` are treated as program text; all other lines are comments. This allows to include usual LaTeX text, but also figures and equations as comments.

A.1.1 The Module `ForSyDeStdLib`

Overview

The ForSyDe Standard Library contains the data types and functions for the ForSyDe design methodolgy.

The module `ForSyDeStdLib` works as a container and exports all other libraries.

```
module ForSyDeStdLib(
                        module DomainInterfaces,
                        module SynchronousProcessLib,
                        module SynchronousLib,
                        module Vector, module Signal, module Memory,
                        module AbsentExt, module Queue,
                        module Combinators, module DFT,
                        module FIR
                    ) where

import DomainInterfaces
import SynchronousProcessLib
import SynchronousLib
import Vector
import Signal
import Memory
import AbsentExt
import Queue
```

```
import Combinators
import FIR
import DFT
```

A.2 ForSyDe Core Language

The ForSyDe core language includes the modules `Signal`, `Vector`, `AbsentExt` and `Combinators`.

A.2.1 The Module `Signal`

Overview

The module `Signal` defines the data type `Signal` and functions operating on this data type.

```
module Signal( Signal (NullS, (:-)), (-:), (+-+), (!-),
               signal, fromSignal,
               unitS, nullS, headS, tailS, atS, takeS, dropS,
               lengthS, infiniteS, copyS, selectS, writeS, readS
             )
where

infixr 5       :-
infixr 5       -:
infixr 5       +-+
infixr 5       !-
```

The Data Type `Signal`

A signal is defined as a list of events. An event has a tag and a value. The tag of an event is defined by the position in the list.

```
data Signal a = NullS
              | a :- Signal a deriving (Eq)
```

A signal is defined as an instance of the classes `Read` and `Show`. The signal `1 :- 2 :- NullS` is represented as {`1,2`}.

Functions on the Data Type `Signal`

The module defines the following functions on the data type `Signal`:

```
signal            :: [a] -> Signal a
fromSignal        :: Signal a -> [a]
unitS             :: a -> Signal a
nullS             :: Signal a -> Bool
headS             :: Signal a -> a
tailS             :: Signal a -> Signal a
atS               :: Int -> Signal a -> a
takeS             :: Int -> Signal a -> Signal a
dropS             :: Int -> Signal a -> Signal a
selectS           :: Int -> Int -> Signal a -> Signal a
lengthS           :: Num a => Signal b -> a
infiniteS         :: (a -> a) -> a -> Signal a
writeS            :: Show a => Signal a -> [Char]
readS             :: Read a => [Char] -> Signal a
(-:)              :: Signal a -> a -> Signal a
(+-+)             :: Signal a -> Signal a -> Signal a
```

The functions `signal` and `fromSignal` convert a list into a signal and vice versa. The function `unitS` creates a signal with one value. The function `nullS` checks if a signal is empty. The function `headS` gives the first value - the head - of a signal, while `tailS` gives the rest of the signal - the tail. The function `atS` returns the n-th event in a signal. The numbering of events in a signal starts with 0. There is also an operator version of this function, `(!-)`. The function `takeS` returns the first n values of a signal, while the function `dropS` drops the first n values from a signal. The function `selectS` takes three parameters, an offset, a stepsize and a signal and returns some elements of the signal such as in the following example:

```
Signal> selectS 2 3 (signal[1,2,3,4,5,6,7,8,9,10])
{3,6,9} :: Signal Integer
```

New signals can be created by means of the following functions. The data constructor `(:-)` adds an element to the signal at the head of the signal. The operator `(-:)` adds at an element to a signal at the tail. Finally the operator `(+-+)` concatinates two signals into one signal. The function `lengthS` returns the length of a *finite* stream. The function `infiniteS` creates an infinite signal. The first argument `f` is a function that is applied on the current value. The second argument `x` gives the first value of the signal.

```
Signal> takeS 5 (infiniteS (*3) 1)
{1,3,9,27,81} :: Signal Integer
```

The function `copyS` creates a signal with n values `x`. The function `writeS` transforms a signal into a string of the following format:

```
Signal> writeS (signal[1,2,3,4,5])
```

```
"1\n2\n3\n4\n5\n" :: [Char]
```

The function `readS` transforms such a formatted string into a signal.

```
Signal> readS "1\n2\n3\n4\n5\n" :: Signal Int
{1,2,3,4,5} :: Signal Int
```

The combinator `fanS` takes two processes `p1` and `p2` and and generates a process network, where a signal is split and processed by the processes `p1` and `p2`.

```
fanS :: (Signal a -> Signal b) -> (Signal a -> Signal c)
         -> Signal a -> (Signal b, Signal c)
```

```
instance (Show a) => Show (Signal a) where
        showsPrec p NullS = showParen (p > 9) (
                              showString "{}")
        showsPrec p xs    = showParen (p > 9) (
                              showChar '{' . showSignal1 xs)
                            where
                              showSignal1 NullS
                                 = showChar '}'
                              showSignal1 (x:-NullS)
                                 = shows x . showChar '}'
                              showSignal1 (x:-xs)
                                 = shows x . showChar ','
                                   . showSignal1 xs

instance Read a => Read (Signal a) where
        readsPrec _ s = readsSignal s

readsSignal           :: (Read a) => ReadS (Signal a)
readsSignal s         =  [((x:-NullS), rest)
                          | ("{", r2)   <- lex s,
                            (x, r3)     <- reads r2,
                            ("}", rest) <- lex r3]
                       ++ [(NullS, r4)
                          | ("{", r5) <- lex s,
                            ("}", r4) <- lex r5]
                       ++ [((x:-xs), r6)
                          | ("{", r7) <- lex s,
                            (x, r8)   <- reads r7,
                            (",", r9) <- lex r8,
                            (xs, r6)  <- readsValues r9]

readsValues           :: (Read a) => ReadS (Signal a)
readsValues s         =  [((x:-NullS), r1)
```

```
                          | (x, r2)   <- reads s,
                            ("}", r1) <- lex r2]
                        ++ [((x:-xs), r3)
                          | (x, r4)   <- reads s,
                            (",", r5) <- lex r4,
                            (xs, r3)  <- readsValues r5]

signal []                   =  NullS
signal (x:xs)               =  x :- signal xs

fromSignal NullS            =  []
fromSignal (x:-xs)          =  x : fromSignal xs

unitS x                     =  x :- NullS

nullS NullS                 =  True
nullS _                     =  False

headS NullS                 =  error "headS: Signal is empty"
headS (x:-_)                =  x

tailS NullS                 =  error "tailS: Signal is empty"
tailS (_:-xs)               =  xs

atS _ NullS
     =  error "atS: Signal has not enough elements"
atS 0 (x:-_)                =  x
atS n (_:-xs)               =  atS (n-1) xs

(!-) xs n                   = atS n xs

takeS 0 _                   = NullS
takeS _ NullS               = NullS
takeS n (x:-xs) | n <= 0    = NullS
                | otherwise = x :- takeS (n-1) xs

dropS 0 NullS               = NullS
dropS _ NullS               = NullS
dropS n (x:-xs) | n <= 0    = x:-xs
         | otherwise        = dropS (n-1) xs


selectS offset step xs = select1S step (dropS offset xs)
  where
```

```
   select1S step NullS   = NullS
   select1S step (x:-xs) = x :- select1S step (dropS (step-1) xs)

(-:) xs x                      = xs +-+ (x :- NullS)

(+-+) NullS   ys               = ys
(+-+) (x:-xs) ys               = x :- (xs +-+ ys)

lengthS NullS                  = 0
lengthS (_:-xs)                = 1 + lengthS xs

infiniteS f x                  = x :- infiniteS f (f x)

copyS 0 x                      = NullS
copyS n x                      = x :- copyS (n-1) x

fanS p1 p2 xs                  = (p1 xs, p2 xs)

writeS NullS   = []
writeS (x:-xs) = show x ++ "\n" ++ writeS xs

readS xs       = readS' (words xs)
  where
    readS' []        = NullS
    readS' ("\n":xs) = readS' xs
    readS' (x:xs)        = read x :- readS' xs
```

A.2.2 The Module `Vector`

Overview

The module `Vector` defines the data type `Vector` and the corresponding functions. It is a development of the module `Vector` defined by Reekie in [92]. Though the vector is modeled as a list, it should be viewed as an array, i.e. a vector has a *fixed size*. Unfortunately, it is not possible to have the size of the vector as a parameter of the vector data type, due to restrictions in Haskells type system. Still most operations are defined for vectors with the same size.

```
module Vector (
             Vector (..), vector, fromVector, unitV, nullV, lengthV,
             atV, replaceV, headV, tailV, lastV, initV, takeV, dropV,
             selectV, groupV, (<+>), (<:), mapV, foldlV, foldrV, scanlV,
             scanrV, meshlV, meshrV, zipWithV, filterV, zipV, unzipV,
```

```
              concatV, reverseV, shiftlV, shiftrV, rotrV, rotlV,
              generateV, iterateV, copyV, serialV, parallelV )
      where

infixr 5 :>
infixl 5 <:
infixr 5 <+>
```

The Data Type `Vector`

The data type `Vector` is modeled similar to a list. It has two data type constructors. `NullV` constructs the empty vector, while `:>` a vector by adding an value to an existing vector. Using the inheritance mechanism of Haskell we have declared `Vector` as an instance of the classes `Read` and `Show`.

This means that the vector `1:>2:>3:>NullV` is shown as `<1,2,3>`.

```
data Vector a = NullV
              | a :> (Vector a) deriving (Eq)
```

Functions on the Data Type `vector`

The function `vector` converts a list into a vector, while the function `fromVector` converts a vector into a list.

```
vector     :: [a] -> Vector a
fromVector :: Vector a -> [a]
```

The function `unitV` creates a vector with one element. The function `nullV` returns `True` if a vector is empty. The function `lengthV` returns the number of elements in a value. The function `atV` returns the n-th element in a vector, starting from zero. The function `replaceV` replaces an element in a vector.

```
unitV    :: a -> Vector a
nullV    :: Vector a -> Bool
lengthV  :: Num a => Vector b -> a
replaceV :: Vector a -> Int -> a -> Vector a
atV      :: Num a => Vector b -> a -> b
```

The functions `headV` and `lastV` return the first element or the the last element of a vector. The functions `tailV` returns all, but the first element of a vector, while `initV` returns all but the last elements of a vector. The function `takeV` returns the first n elements of a vector while the function `dropV` drops the first n elements of a vector.

```
headV :: Vector a -> a
tailV :: Vector a -> Vector a
lastV :: Vector a -> a
initV :: Vector a -> Vector a
takeV :: (Num a, Ord a) => a -> Vector b -> Vector b
dropV :: (Num a, Ord a) => a -> Vector b -> Vector b
```

The function `selectV` selects elements in the vector. The first argument gives the initial element, starting from zero, the second argument gives the stepsize between elements and the last argument gives the number of elements.

```
selectV :: (Num a, Ord a) => a -> a -> a -> Vector b -> Vector b
```

The function `groupV` groups a vector into a vector of vectors of size n.

```
groupV :: (Num a, Ord a) => a -> Vector b -> Vector (Vector b)
```

The data constructor `(:>)` adds an element add the front of the vector, while the operator `(<:)` adds an element at the end. The operator `<+>` concatinates two vectors. The function `concatV` concats a vector of vectors into a single vector.

```
(<+>) :: Vector a -> Vector a -> Vector a
(<:)  :: Vector a -> a -> Vector a
```

The higher-order function `mapV` applies a function on all elements of a vector.

```
mapV :: (a -> b) -> Vector a -> Vector b
```

The higher-order function `zipWithV` applies a function pairwise on to vectors.

```
zipWithV :: (a -> b -> c) -> Vector a -> Vector b -> Vector c
```

The higher-order functions `foldlV` and `foldrV` fold a function from the right or from the left over a vector using an initial value.

```
foldlV :: (a -> b -> a) -> a -> Vector b -> a
foldrV :: (b -> a -> a) -> a -> Vector b -> a
```

The higher-function `filterV` takes a predicate function and a vector and creates a new vector with the elements for which the predicate is true.

```
filterV :: (a -> Bool) -> Vector a -> Vector a
```

The function `zipV` zips two vectors into a vector of tuples. The function `unzipV` unzips a vector of tuples into two vectors.

```
zipV   :: Vector a -> Vector b -> Vector (a, b)
unzipV :: Vector (a, b) -> (Vector a, Vector b)
```

The function `shiftlV` shifts a value from the left into a vector. The function `shiftrV` shifts a value from the right into a vector. The functions `rotlV`, `rotrV` rotates a vector to the left or to the right. Note that these fuctions do not change the size of a vector.

```
shiftlV :: Vector a -> a-> Vector a
shiftrV :: Vector a -> a -> Vector a
rotrV   :: Vector a -> Vector a
rotlV   :: Vector a -> Vector a
```

The function `concatV` transforms a vector of vectors to a single vector. The function `reverseV` reverses the order of elements in a vector.

```
concatV   :: Vector (Vector a) -> Vector a
reverseV  :: Vector a -> Vector a
```

The function `iterateV` generates a vector with a given number of elements starting from an initial element using a supplied function for the generation of elements. The function `generateV` behaves in the same way, but starts with th e application of the supplied function to the supplied value. The function `copyV` generates a vector with a given number of copies of the same element.

```
Vector> iterateV 5 (+1) 1
<1,2,3,4,5> :: Vector Integer
Vector> generateV 5 (+1) 1
<2,3,4,5,6> :: Vector Integer
Vector> copyV 7 5
<5,5,5,5,5,5,5> :: Vector Integer
```

```
iterateV  :: Num a => a -> (b -> b) -> b -> Vector b
generateV :: Num a => a -> (b -> b) -> b -> Vector b
copyV     :: Num a => a -> b -> Vector b
```

The functions `serialV` and `parallelV` can be used to construct serial and parallel networks of processes.

```
serialV         :: Vector (a -> a) -> a -> a
parallelV       :: Vector (a -> b) -> Vector a -> Vector b
```

The functions `scanlV` and `scanrV` ”scan” a function through a vector. The functions take an initial element apply a functions recursively first on the element and then on the result of the function application.

```
scanlV    :: (a -> b -> a) -> a -> Vector b -> Vector a
scanrV    :: (b -> a -> a) -> a -> Vector b -> Vector a
```

Reekie also proposed the `meshlV` and `meshrV` iterators. They are like a combination of `mapV` and `scanlV` or `scanrV`. The argument function supplies a pair of values: the first is input into the next application of this function, and the second is the output value. As an example consider the expression:

```
f x y = (x+y, x+y)

s1 = vector [1,2,3,4,5]
```

Here `meshlV` can be used to calculate the running sum.

```
Vector> meshlV f 0 s1
(15,<1,3,6,10,15>)

meshlV   :: (a -> b -> (a, c)) -> a -> Vector b -> (a, Vector c)
meshrV   :: (a -> b -> (c, b)) -> b -> Vector a -> (Vector c, b)
```

Implementation

```
instance (Show a) => Show (Vector a) where
      showsPrec p NullV = showParen (p > 9) (
                            showString "<>")
      showsPrec p xs    = showParen (p > 9) (
                            showChar '<' . showVector1 xs)
                          where
                            showVector1 NullV
                              = showChar '>'
                            showVector1 (x:>NullV)
                              = shows x . showChar '>'
                            showVector1 (x:>xs)
                              = shows x . showChar ','
                                        . showVector1 xs


instance Read a => Read (Vector a) where
      readsPrec _ s = readsVector s

readsVector :: (Read a) => ReadS (Vector a)
readsVector s = [((x:>NullV), rest) | ("<", r2) <- lex s,
                                      (x, r3)   <- reads r2,
                                      (">", rest) <- lex r3]
              ++
             [(NullV, r4)        | ("<", r5) <- lex s,
```

```
                                  (">", r4) <- lex r5]
              ++
             [((x:>xs), r6)      | ("<", r7) <- lex s,
                                   (x, r8)   <- reads r7,
                                   (",", r9) <- lex r8,
                                   (xs, r6) <- readsValues r9]

readsValues :: (Read a) => ReadS (Vector a)
readsValues s = [((x:>NullV), r1) | (x, r2)   <- reads s,
                                    (">", r1) <- lex r2]
              ++
              [((x:>xs), r3)    | (x, r4)   <- reads s,
                                  (",", r5) <- lex r4,
                                  (xs, r3)  <- readsValues r5]

vector []     = NullV
vector (x:xs) = x :> (vector xs)

fromVector NullV   = []
fromVector (x:>xs) = x : fromVector xs

unitV x = x :> NullV

nullV NullV   = True
nullV _             = False

lengthV NullV   = 0
lengthV (_:>xs) = 1 + lengthV xs

replaceV vs n x
    | n <= lengthV vs && n >= 0 = takeV n vs <+> unitV x
                                            <+> dropV (n+1) vs
    | otherwise                =  vs

NullV   `atV` _ = error "atV: Vector has not enough elements"
(x:>_)  `atV` 0 = x
(_:>xs) `atV` n = xs `atV` (n-1)

headV NullV   = error "headV: Vector is empty"
headV (v:>_) = v

tailV NullV   = error "tailV: Vector is empty"
tailV (_:>vs) = vs
```

```
lastV NullV      = error "lastV: Vector is empty"
lastV (v:>NullV) = v
lastV (_:>vs)    = lastV vs

initV NullV      = error "initV: Vector is empty"
initV (_:>NullV) = NullV
initV (v:>vs)    = v :> initV vs

takeV 0 _                  = NullV
takeV _ NullV              = NullV
takeV n (v:>vs) | n <= 0   = NullV
                | otherwise = v :> takeV (n-1) vs

dropV 0 vs                 = vs
dropV _ NullV              = NullV
dropV n (v:>vs) | n <= 0   = v :> vs
                | otherwise = dropV (n-1) vs

selectV f s n vs | n <= 0
                   = NullV
                 | (f+s*n-1) > lengthV vs
                   = error "selectV: Vector has not enough elements"
                 | otherwise
                   = atV vs f :> selectV (f+s) s (n-1) vs

groupV n v
      | lengthV v < n = NullV
      | otherwise     = selectV 0 1 n v
                        :> groupV n (selectV n 1 (lengthV v-n) v)

NullV <+> ys   = ys
(x:>xs) <+> ys = x :> (xs <+> ys)

xs <: x = xs <+> unitV x

mapV _ NullV   = NullV
mapV f (x:>xs) = f x :> mapV f xs

zipWithV f (x:>xs) (y:>ys) = f x y :> (zipWithV f xs ys)
zipWithV _ _       _       = NullV

foldlV _ a NullV   = a
foldlV f a (x:>xs) = foldlV f (f a x) xs
```

```
foldrV _ a NullV   = a
foldrV f a (x:>xs) = f x (foldrV f a xs)

filterV _ NullV   = NullV
filterV p (v:>vs) = if (p v) then
                       v :> filterV p vs
                    else
                       filterV p vs

zipV (x:>xs) (y:>ys) = (x, y) :> zipV xs ys
zipV _       _       = NullV

unzipV NullV           = (NullV, NullV)
unzipV ((x, y) :> xys) = (x:>xs, y:>ys)
                         where (xs, ys) = unzipV xys

shiftlV vs v = v :> initV vs

shiftrV vs v = tailV vs <: v

rotrV NullV = NullV
rotrV vs    = tailV vs <: headV vs

rotlV NullV = NullV
rotlV vs    = lastV vs :> initV vs

concatV = foldrV (<+>) NullV

reverseV NullV   = NullV
reverseV (v:>vs) = reverseV vs <: v

generateV 0 _ _ = NullV
generateV n f a = x :> generateV (n-1) f x
                  where x = f a

iterateV 0 _ _ = NullV
iterateV n f a = a :> iterateV (n-1) f (f a)

copyV k x = iterateV k id x

serialV  fs      x = serialV' (reverseV fs ) x
  where
    serialV' NullV   x = x
    serialV' (f:>fs) x = serialV fs (f x)
```

```
parallelV NullV   NullV   = NullV
parallelV _       NullV
   = error "parallelV: Vectors have not the same size!"
parallelV NullV   _
   = error "parallelV: Vectors have not the same size!"
parallelV (f:>fs) (x:>xs) = f x :> parallelV fs xs

scanlV _ _ NullV   = NullV
scanlV f a (x:>xs) = q :> scanlV f q xs
                   where q = f a x

scanrV _ _ NullV      = NullV
scanrV f a (x:>NullV) = f x a :> NullV
scanrV f a (x:>xs)    = f x y :> ys
                      where ys@(y:>_) = scanrV f a xs

meshlV _ a NullV   = (a, NullV)
meshlV f a (x:>xs) = (a'', y:>ys)
                   where (a', y)   = f a x
                         (a'', ys) = meshlV f a' xs

meshrV _ a NullV    = (NullV, a)
meshrV f a (x:>xs)  = (y:>ys, a'')
                    where (y, a'') = f x a'
                          (ys, a') = meshrV f a xs
```

A.2.3 The Module `AbsentExt`

Overview

The module `AbsentExt` is used to extend existing data types with the value ”absent” ($\perp$).

```
module AbsentExt(
                 AbstExt (Abst, Prst), fromAbstExt, abstExt, psi,
                 isAbsent, isPresent, abstExtFunc)
               where
```

The data type `AbstExt` has two constructors. The constructor `Abst` is used to model the absence of a value, while the constructor `Prst` is used to model present values.

```
data AbstExt a               =  Abst
                             |  Prst a deriving (Eq)
```

The data type `AbstExt` is defined as an instance of `Show` and `Read`. '_' represents the value `Abst` while a present value is represented with its value, e.g. `Prst 1` is represented as '1'.

Functions on the Data Type `AbsentExt`

The module defines the following functions:

```
fromAbstExt          :: a -> AbstExt a -> a
isPresent            :: AbstExt a -> Bool
isAbsent             :: AbstExt a -> Bool
abstExtFunc          :: (a -> b) -> AbstExt a -> AbstExt b
psi                  :: (a -> b) -> AbstExt a -> AbstExt b
```

The function `abstExt` converts a value into an extended value. The function `fromAbstExt` converts a value from a extended value.

The functions `isPresent` and `isAbsent` check for the presence or absence of a value.

The function `abstExtFunc` extends a function in order to process absent extended values. If the input is $\perp$, the output will also be $\perp$. The function `psi` is identical to `abstExtFunc` and should be used in future.

Implementation of Library Functions

```
instance Show a => Show (AbstExt a) where
         showsPrec _ x          = showsAbstExt x

showsAbstExt Abst               = (++) "_"
showsAbstExt (Prst x)           = (++) (show x)

instance Read a => Read (AbstExt a) where
         readsPrec _ x          =  readsAbstExt x

readsAbstExt                    :: (Read a) => ReadS (AbstExt a)
readsAbstExt s                  =    [(Abst, r1)    | ("_", r1) <- lex s]
                                  ++ [(Prst x, r2)  | (x, r2) <- reads s]

abstExt v                       =  Prst v

fromAbstExt x Abst              =  x
fromAbstExt _ (Prst y)          =  y

isPresent Abst                  =  False
```

```
isPresent (Prst _)            =  True

isAbsent                      =  not . isPresent

abstExtFunc f                 = f'
  where f' Abst               = Abst
        f' (Prst x)           = Prst (f x)

psi                           = abstExtFunc
```

A.2.4 The Module `Combinators`

Overview

The module contains operators for function (sequential) composition and parallel composition.

```
module Combinators where

funComb1                = (.)

funComb2 p1 p2          = p
  where p s1 s2         = p1 (p2 s1 s2)

funComb3 p1 p2          = p
  where p s1 s2 s3      = p1 (p2 s1 s2 s3)

funComb4 p1 p2          = p
  where p s1 s2 s3 s4   = p1 (p2 s1 s2 s3 s4)

parComb p1 p2           = p
  where p s1 s2         = (p1 s1, p2 s2)
```

A.3 Libraries of System Functions and Data Types

A.3.1 The Module `Memory`

Overview

This module contains the data structure and access functions for the memory model.

```
module Memory (
               module AbsentExt, Memory (..), Access (..),
               MemSize, Adr, newMem, memState, memOutput
```

```
            ) where

import Vector
import AbsentExt
```

Data Structure

The data type `Memory` is modeled as a vector. The data tpe `Access` defines two access patterns, `Read adr` and `Write adr val`, where `adr` can be of any type..

```
type Adr                    =  Int
type MemSize                =  Int

data Memory a               =  Mem Adr (Vector (AbstExt a))
                                  deriving (Eq, Show)

data Access a               =  Read Adr
                            |  Write Adr a
                                  deriving (Eq, Show)
```

Functions on the data type `Memory`

The module defines the following access functions for the memory:

```
newMem                      :: MemSize -> Memory a
memState                    :: Memory a -> Access a -> Memory a
memOutput                   :: Memory a -> Access a -> AbstExt a
```

The function `newMem` creates a new memory, where the number of entries is given by a parameter. The function `memState` gives the new state of the memory, after an access to a memory. A `Read` operation leaves the meory unchanged. The function `memOutput` gives the output of the memory after an access to the memory. A `Write` operation gives an absent value as output.

Implementation of Functions

```
newMem size                 =  Mem size (copyV size Abst)

writeMem                    :: Memory a -> (Int, a) -> Memory a
writeMem (Mem size vs) (i, x)
   | i < size && i >= 0     =  Mem size (replaceV vs i (abstExt x))
   | otherwise              =  Mem size vs
```

```
readMem                      :: Memory a -> Int -> (AbstExt a)
readMem (Mem size vs) i
   | i < size && i >= 0      =  vs `atV` i
   | otherwise               =  Abst

memState mem (Read _)        =  mem
memState mem (Write i x)     =  writeMem mem (i, x)

memOutput mem (Read i)       =  readMem mem i
memOutput _   (Write _ _)    =  Abst
```

A.3.2 The Module `Queue`

Overview

The module `Queue` provides two data types, that can be used to model queue structures, such as FIFOs. There is a data type for an queue of infinite size `Queue` and one for finite size `FiniteQueue`.

The data type Queue

A queue is modeled as a list. The data type `FiniteQueue` has an additional parameter, that determines the size of the queue.

```
module Queue where

import AbsentExt

data Queue a       =  Q [a] deriving (Eq, Show)
data FiniteQueue a =  FQ Int [a] deriving (Eq, Show)
```

Functions on the data types `Queue` and `FiniteQueue`

Table A.3.2 shows the functions an the data types `Queue` and `FiniteQueue`.

```
pushQ                        :: Queue a -> a -> Queue a
pushListQ                    :: Queue a -> [a] -> Queue a
popQ                         :: Queue a -> (Queue a, AbstExt a)
queue                        :: [a] -> Queue a
pushFQ                       :: FiniteQueue a -> a -> FiniteQueue a
pushListFQ                   :: FiniteQueue a -> [a] -> FiniteQueue a
popFQ                        :: FiniteQueue a
                             -> (FiniteQueue a, AbstExt a)
finiteQueue                  :: Int -> [a] -> FiniteQueue a
```

infinite	finite	description
`pushQ`	`pushFQ`	pushes one element on the queue
`pushListQ`	`pushListFQ`	pushes a list of elements on the queue
`popQ`	`popFQ`	pops one element from the queue
`queue`	`finiteQueue`	transforms a list into a queue

Table A.1. Functions on the data types `Queue` and `FiniteQueue`

Implementation

```
pushQ (Q q) x                = Q (q ++ [x])

pushListQ (Q q) xs           = Q (q ++ xs)

popQ (Q [])                  = (Q [], Abst)
popQ (Q (x:xs))              = (Q xs, Prst x)

queue xs                     = Q xs

pushFQ (FQ n q) x            = if length q < n then
                                  (FQ n (q ++ [x]))
                               else
                                  (FQ n q)

pushListFQ (FQ n q) xs       = FQ n (take n (q ++ xs))

popFQ (FQ n [])              = (FQ n [], Abst)
popFQ (FQ n (q:qs))          = (FQ n qs, Prst q)

finiteQueue n xs             = FQ n (take n xs)
```

A.3.3 The Module `DFT`

The module includes the standard Discrete Fourier Transform (DFT) function, which is formulated as

$$X(K) = \sum_{n=0}^{N-1} x(n) W_N^{kn} \quad 0 \leq k \leq N-1 \tag{A.1}$$

where

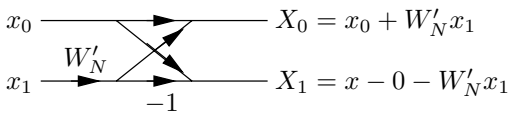


Figure A.2. Basic butterfly computation in the decimiation-in-time algorithm

$$W_N = e^{-j2\pi/N} \tag{A.2}$$

and a fast Fourier transform (FFT) algorithm, for computing the DFT, when the size N is a power of 2.

```
module DFT(dft, fft) where

import Signal
import Vector
import Complex
```

Here follows the ForSyDe implementation of the DFT:

```
dft bigN x | bigN == (lengthV x) = mapV (bigX_k bigN x) (nVector x)
           | otherwise   = error "DFT: Vector has not the right size!"
   where
     nVector x       = iterateV (lengthV x) (+1) 0
     bigX_k bigN x k = sumV (zipWithV (*) x (bigW k bigN))
     bigW k bigN     = mapV (** k) (mapV cis (fullcircle bigN))
     sumV            = foldlV (+) (0:+ 0)

fullcircle :: Integer -> Vector Double
fullcircle n = fullcircle1 0 (fromInteger n) n
          where
             fullcircle1 l m n
                | l == m    = NullV
                | otherwise = -2*pi*l/(fromInteger n)
                              :> fullcircle1 (l+1) m n
```

Here follows the Radix 2-FFT algorithm (decimation in time) implementation [91]. It is a divide and conquer algorithm, which reuses the calculation of the basic butterfly (Figure A.2):

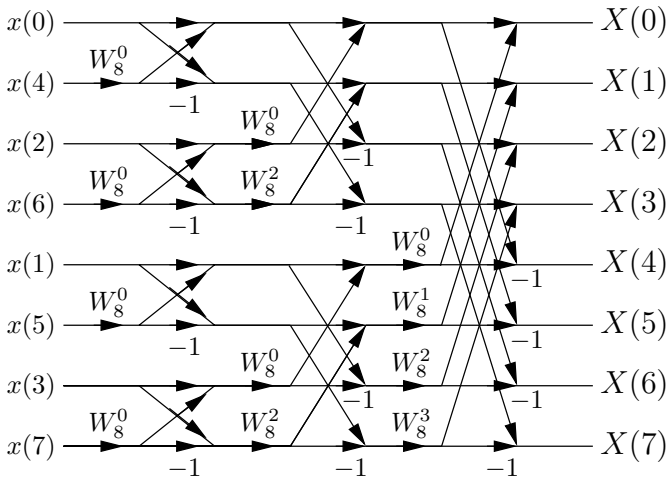


Figure A.3. Eight-point decimation-in-time FFT algorithm

$$
\begin{aligned}
X(k) &= \sum_{m=0}^{(N/2)-1} f_{even}(m) W_{N/2}^{km} + W_N^k \sum_{m=0}^{(N/2)-1} f_{odd}(m) W_{N/2}^{km} \\
&= F_{even}(k) + W_N^k F_{odd}(k) \quad k = 0, 1, \ldots, N-1 \qquad \text{(A.3)}
\end{aligned}
$$

where

$$
\begin{aligned}
f_{even}(n) &= x(2n) \\
f_{odd}(n) &= x(2n+1)
\end{aligned}
$$

The calculation of an eight-point FFT is illustrated in Figure A.3.

```
fft bigN xv | bigN == (lengthV xv) = mapV (bigX xv) (kVector bigN)
            | otherwise = error "FFT: Vector has not the right size!"

kVector bigN = iterateV bigN (+1) 0

bigX (x0:>x1:>NullV) k | even k = x0 + x1 * bigW 2 0
                       | odd k  = x0 - x1 * bigW 2 0
bigX xv k = bigF_even k + bigF_odd k * bigW bigN (fromInteger k)
     where bigF_even k = bigX (evens xv) k
           bigF_odd k = bigX (odds xv) k
           bigN = lengthV xv

bigW bigN k = cis (-2 * pi * (fromInteger k) / (fromInteger bigN))
```

```
evens NullV       = NullV
evens (v1:>NullV) = v1 :> NullV
evens (v1:>_:>v)  = v1 :> evens v

odds NullV        = NullV
odds (_:>NullV)   = NullV
odds (_:>v2:>v)   = v2 :> odds v
```

A.4 Computational Model Libraries

A.4.1 The Module `SynchronousLib`

Overview

The synchronuous library `SynchronousLib` defines process constructors and processes for the synchronous computational model. A process constructor is a higher order function which together with combinatorial function(s) and values as arguments constructs a process. Thus a process constructor can also be viewed as a *process constructor*.

```
module SynchronousLib(
                        module Vector, module Signal, module AbsentExt,
                        mapSY, zipWithSY, zipWith3SY,
                        zipWith4SY, zipWithxSY, scanlSY,
                        scanl2SY, scanl3SY, scanldSY, scanld2SY,
                        scanld3SY, delaySY, delaynSY, whenSY,
                        fillSY, holdSY, zipSY, zip3SY, zip4SY, unzipSY,
                        unzip3SY, unzip4SY, zipxSY, unzipxSY, mapxSY,
                        mooreSY, moore2SY, moore3SY, mealySY, mealy2SY,
                        mealy3SY, fstSY, sndSY, sourceSY
                      ) where

import Signal
import Vector
import AbsentExt
```

Process Constructors for Combinatorial Processes

Combinatorial processes do not posess an internal state, so that the output only depends on input signals.

The module includes the following process constructors for combinatorial processes:

```
mapSY          :: (a -> b) -> Signal a -> Signal b
zipWithSY      :: (a -> b -> c) -> Signal a -> Signal b -> Signal c
zipWith3SY     :: (a -> b -> c -> d) -> Signal a -> Signal b
                  -> Signal c -> Signal d
zipWith4SY     :: (a -> b -> c -> d -> e) -> Signal a -> Signal b
                  -> Signal c -> Signal d -> Signal e
zipWithxSY     :: (Vector a -> b) -> Vector (Signal a) -> Signal b
mapxSY         :: (a -> b) -> Vector (Signal a) -> Vector (Signal b)
```

The process constructor `mapSY` takes a combinatorial function as argument and returns a process with one input signal and one output signal. This is shown in the following, where `mapSY (+1)` is a process which increments all values of an input signal.

In a similar way `zipWithSY`, `zipWith3SY` and `zipWith4SY` apply a combinatorial function on a number of input signals.

The process constructor `zipWithxSY` works as `zipWithSY`, but works with a vector of signals as input.

The process constructor `mapxSY` creates a process network that maps a function onto all signals in a vector of signals.

Process Constructors for Sequential Processes

Sequential processes have a local state. Process Constructors that construct such processes take not only functions but also values as arguments to express the value of the local state of the process. The output of sequential processes is deterministic and depends on the initial state and the input signals.

The module includes the following process constructors for sequential processes:

```
delaySY        :: a -> Signal a -> Signal a
delaynSY       :: a -> Int -> Signal a -> Signal a
scanlSY        :: (a -> b -> a) -> a -> Signal b -> Signal a
scanl2SY       :: (a -> b -> c -> a) -> a -> Signal b -> Signal c
                   -> Signal a
scanl3SY       :: (a -> b -> c -> d -> a) -> a -> Signal b
                   -> Signal c -> Signal d -> Signal a
scanldSY       :: (a -> b -> a) -> a -> Signal b -> Signal a
scanld2SY      :: (a -> b -> c -> a) -> a -> Signal b -> Signal c
                   -> Signal a
scanld3SY      :: (a -> b -> c -> d -> a) -> a -> Signal b
                   -> Signal c -> Signal d -> Signal a
mooreSY        :: (a -> b -> a) -> (a -> c) -> a -> Signal b -> Signal c
moore2SY       :: (a -> b -> c -> a) -> (a -> d) -> a -> Signal b
```

```
                -> Signal c -> Signal d
moore3SY    :: (a -> b -> c -> d -> a) -> (a -> e) -> a -> Signal b
                -> Signal c -> Signal d -> Signal e
mealySY     :: (a -> b -> a) -> (a -> b -> c) -> a -> Signal b
                -> Signal c
mealy2SY    :: (a -> b -> c -> a) -> (a -> b -> c -> d) -> a
                -> Signal b -> Signal c -> Signal d
mealy3SY    :: (a -> b -> c -> d -> a) -> (a -> b -> c -> d -> e) -> a
                -> Signal b -> Signal c -> Signal d -> Signal e
sourceSY    :: (a -> a) -> a -> Signal a
filterSY    :: (a -> Bool) -> Signal a -> Signal (AbstExt a)
```

The process constructor `delaySY` delays the signal one event cycle by introducing an initial value at the beginning of the output signal. The process constructor `delaynSY` delays the signal n events by introducing n identical default values.

We define two different basic process constructors to construct sequential processes, `scanlSY` and `scanldSY`. Both process constructors take a function `ns` and a state `m` as arguments. Both process constructors use the function `ns` to calculate the next state, but calculate the output in a different way. `scanlSY` behaves like the Haskell prelude function `scanl` and has the value of the new state as its output value, while `scanldSY` has the current state value as output. The following example exemplifies this:

```
SynchronousLib> scanlSY (+) 0 (signal [1,2,3,4])
{1,3,6,10} :: Signal Integer
SynchronousLib> scanldSY (+) 0 (signal [1,2,3,4])
{0,1,3,6} :: Signal Integer
```

Process Constructors like `scanl2SY`, `scanl2DelaySY` are used in the same way for several input signals.

The process constructors `mooreSY` and `mealySY` are used to model state machines. These process constructors are based on the process constructor `scanldSY` as is naturally for state machines in hardware, that the output operates on the current state and not on the next state.

These process constructors take a function `ns` to calculate the next state, a function `o` to calculate the output and an value `m` for the initial state. In a process based on the `mooreSY` process constructor the output function `o` operates only on the current state of the process. In contrast the output function of a process based on the `mealySY` process constructor operates on both the current state and the input.

The process `sourceSY` takes a function f and an initial state s_0 and generates an infinite signal starting with the initial state s_0 as first output followed by the

recursive application of f on the current state which also serve as output values. The process that has the infinite signal of natural numbers as output is constructed by

```
SynchronousLib> takeS 5 (sourceSY (+1) 0)
{0,1,2,3,4} :: Signal Integer
```

The process constructor `filterSY` takes a predicate `p` and produces a process, that discards all values that do not fullfil the predicate `p`. In this case the output is $\perp$.

Processes

The module also contains the following synchronous processes:

```
whenSY      :: Signal (AbstExt a) -> Signal (AbstExt b)
               -> Signal (AbstExt a)
fillSY      :: a -> Signal (AbstExt a) -> Signal a
holdSY      :: a -> Signal (AbstExt a) -> Signal a
zipSY       :: Signal a -> Signal b -> Signal (a,b)
zip3SY      :: Signal a -> Signal b -> Signal c -> Signal (a,b,c)
zip4SY      :: Signal a -> Signal b -> Signal c -> Signal d
               -> Signal (a,b,c,d)
unzipSY     :: Signal (a,b) -> (Signal a,Signal b)
unzip3SY    :: Signal (a, b, c) -> (Signal a, Signal b, Signal c)
unzip4SY    :: Signal (a,b,c,d)
               -> (Signal a,Signal b,Signal c,Signal d)
zipxSY      :: Vector (Signal a) -> Signal (Vector a)
unzipxSY    :: Signal (Vector a) -> Vector (Signal a)
fstSY       :: Signal (a,b) -> Signal a
sndSY       :: Signal (a,b) -> Signal b
```

The process constructor `whenSY` creates a process that synchronizes a signal of timed values with another signal of timed values. The output signal has the value of the first signal whenever an event has a present value and $\perp$ when the event has an absent value.

The process constructor `fillSY` creates a process that 'fills' a signal with timed values by replacing absent values with a given value.

The process constructor `holdSY` creates a process that 'fills' a signal with values by replacing absent values by the preceding present value. Only in cases, where no preceding value exists, the absent value is replaced by the supplied value.

The process `zipSY` 'zips' two incoming signals into one signal of tuples, while the process `unzipSY` 'unzips' a signal of tuples into two signals. The functions

`zip3SY`, `zip4SY`, `unzip4SY` and `unzip4SY` perform the corresponding function for three and four signals.

The process `zipxSY` 'zip' a signal of vectors into a vector of signals. The process `unzipxSY` 'unzips' a vector of signals into a signal of vectors.

The processes `fstSY` and `sndSY` select the always the first or second value from a signal of pairs.

Implementation of Library Functions

```
mapSY _ NullS   = NullS
mapSY f (x:-xs) = f x :- (mapSY f xs)

zipWithSY _ NullS   _       = NullS
zipWithSY _ _       NullS   = NullS
zipWithSY f (x:-xs) (y:-ys) = f x y :- (zipWithSY f xs ys)

zipWith3SY _ NullS   _       _       = NullS
zipWith3SY _ _       NullS   _       = NullS
zipWith3SY _ _       _       NullS   = NullS
zipWith3SY f (x:-xs) (y:-ys) (z:-zs) = f x y z
                                       :- (zipWith3SY f xs ys zs)

zipWith4SY _ NullS   _       _       _      = NullS
zipWith4SY _ _       NullS   _       _      = NullS
zipWith4SY _ _       _       NullS   _      = NullS
zipWith4SY _ _       _       _       NullS  = NullS
zipWith4SY f (w:-ws) (x:-xs) (y:-ys) (z:-zs)
                                      = f w x y z
                                        :- (zipWith4SY f ws xs ys zs)

zipWithxSY f = mapSY f . zipxSY

mapxSY f = mapV (mapSY f)

scanlSY _ _    NullS   = NullS
scanlSY f mem (x:-xs) = f mem x :- (scanlSY f newmem xs)
                        where newmem = f mem x

scanl2SY _ _    NullS   _       = NullS
scanl2SY _ _    _       NullS   = NullS
scanl2SY f mem (x:-xs) (y:-ys) = f mem x y
                                 :- (scanl2SY f newmem xs ys)
                                    where newmem = f mem x y
```

```
scanl3SY _ _    NullS   _       _       = NullS
scanl3SY _ _    _       NullS   _       = NullS
scanl3SY _ _    _       _       NullS   = NullS
scanl3SY f mem (x:-xs) (y:-ys) (z:-zs)
       = f mem x y z :- (scanl3SY f newmem xs ys zs)
         where newmem = f mem x y z

scanldSY _ _ NullS   = NullS
scanldSY f mem (x:-xs) = mem :- (scanldSY f newmem xs)
              where newmem = f mem x

scanld2SY _ _   NullS   _       = NullS
scanld2SY _ _   _       NullS   = NullS
scanld2SY f mem (x:-xs) (y:-ys) = mem :- (scanld2SY f newmem xs ys)
                                where newmem = f mem x y

scanld3SY _ _   NullS   _       _       = NullS
scanld3SY _ _   _       NullS   _       = NullS
scanld3SY _ _   _       _       NullS   = NullS
scanld3SY f mem (x:-xs) (y:-ys) (z:-zs)
       = mem :- (scanld3SY f newmem xs ys zs)
         where newmem = f mem x y z

delaySY e es = e:-es

delaynSY e n xs | n <= 0    = xs
                | otherwise = e :- delaynSY e (n-1) xs

mooreSY nextState output initial
       = mapSY output . (scanldSY nextState initial)

moore2SY nextState output initial inp1 inp2 =
        mapSY output (scanld2SY nextState initial inp1 inp2)

moore3SY nextState output initial inp1 inp2 inp3 =
        mapSY output (scanld3SY nextState initial inp1 inp2 inp3)

mealySY nextState output initial signal =
       zipWithSY output (scanldSY nextState initial signal) signal

mealy2SY nextState output initial inp1 inp2 =
       zipWith3SY output (scanld2SY nextState initial inp1 inp2)
                         inp1 inp2
```

```
mealy3SY nextState output initial inp1 inp2 inp3 =
        zipWith4SY output (scanld3SY nextState initial inp1 inp2 inp3)
                          inp1 inp2 inp3

filterSY p NullS        = NullS
filterSY p (x:-xs)      = if (p x == True) then
                             Prst x :- filterSY p xs
                          else
                             Abst :- filterSY p xs
sourceSY f s0           = o
  where
    o                   = delaySY s0 s
    s                   = mapSY f o


whenSY NullS   _            = NullS
whenSY _       NullS        = NullS
whenSY (_:-xs) (Abst:-ys)   = Abst :- (whenSY xs ys)
whenSY (x:-xs) (_:-ys)      = x    :- (whenSY xs ys)

fillSY a xs = mapSY (replaceAbst a) xs
              where replaceAbst a Abst     = a
                    replaceAbst _ (Prst x) = x

holdSY a xs = scanlSY hold a xs
              where hold a Abst     = a
                    hold _ (Prst x) = x

zipSY (x:-xs) (y:-ys) = (x, y) :- zipSY xs ys
zipSY _      _       = NullS

zip3SY (x:-xs) (y:-ys) (z:-zs) = (x, y, z) :- zip3SY xs ys zs
zip3SY _             _  _      = NullS

zip4SY (w:-ws) (x:-xs) (y:-ys) (z:-zs) = (w, x, y, z)
                                         :- zip4SY ws xs ys zs
zip4SY _        _      _        _      = NullS

unzipSY NullS        = (NullS, NullS)
unzipSY ((x, y):-xys) = (x:-xs, y:-ys) where (xs, ys) = unzipSY xys

unzip3SY NullS             = (NullS, NullS, NullS)
unzip3SY ((x, y, z):-xyzs) = (x:-xs, y:-ys, z:-zs) where
```

```
                              (xs, ys, zs) = unzip3SY xyzs

unzip4SY NullS              = (NullS, NullS, NullS, NullS)
unzip4SY ((w,x,y,z):-wxyzs) = (w:-ws, x:-xs, y:-ys, z:-zs) where
                               (ws, xs, ys, zs) = unzip4SY wxyzs

zipxSY NullV            = NullS
zipxSY (NullS   :> xss) = zipxSY xss
zipxSY ((x:-xs) :> xss) = (x :> (mapV headS xss))
                            :- (zipxSY (xs :> (mapV tailS xss)))

unzipxSY NullS            = NullV
unzipxSY (NullV   :- vss) = unzipxSY vss
unzipxSY ((v:>vs) :- vss) = (v :- (mapSY headV vss))
                            :> (unzipxSY (vs :- (mapSY tailV vss)))

fstSY = mapSY fst

sndSY = mapSY snd
```

A.4.2 The Module `DomainInterfaces`

Overview

The module `DomainInterfaces` defines domain interface constructors for the multi-rate computational model.

```
module DomainInterfaces(downDI, upDI, par2serxDI, ser2parxDI,
                        par2ser2DI, par2ser3DI, par2ser4DI,
                        ser2par2DI, ser2par3DI, ser2par4DI) where

import Signal
import Vector
import SynchronousLib
```

Domain Interface Constructors

The domain interface constructors `downDI` and $upDI$ take a parameter k and down- and up-sample an input signal.

The domain interface constructors `par2ser2DI`, `par2ser3DI` and `par2ser4DI` implement the domain interface constructor $p2sDI(m)$ for $m = 2, 3, 4$. The domain interface constructors `par2serxDI` implements the domain interface constructor $p2sDI(m)$ for a variable m.

The domain interface constructors `ser2par2DI`, `ser2par3DI` and `ser2par4DI` implement the domain interface constructor $s2pDI(m)$ for $m = 2, 3, 4$. The domain interface constructors `ser2parxDI` implements the domain interface constructor $s2pDI(m)$ for a variable m.

```
downDI     :: Num a => a -> Signal b -> Signal b
upDI       :: Num a => a -> Signal b -> Signal (AbstExt b)
par2serxDI :: Vector (Signal a) -> Signal a
ser2parxDI :: (Num a, Ord a) => a -> Signal (AbstExt b)
                                  -> Vector (Signal (AbstExt b))
par2ser2DI :: Signal a -> Signal a -> Signal a
par2ser3DI :: Signal a -> Signal a -> Signal a -> Signal a
par2ser4DI :: Signal a -> Signal a -> Signal a -> Signal a
           -> Signal a
ser2par2DI :: Signal a -> Signal (AbstExt a,AbstExt a)
ser2par3DI :: Signal a -> Signal (AbstExt a,AbstExt a,AbstExt a)
ser2par4DI :: Signal a
           -> Signal (AbstExt a,AbstExt a,AbstExt a,AbstExt a)
```

Implementation

```
downDI n xs     = down1 n 1 xs
  where down1 n m NullS   = NullS
        down1 1 1 (x:-xs) = x :- down1 1 1 xs
        down1 n 1 (x:-xs) = x :- down1 n 2 xs
        down1 n m (x:-xs) = if m == n then
                               down1 n 1 xs
                            else
                               down1 n (m+1) xs

upDI n NullS   = NullS
upDI n (x:-xs) = (Prst x) :- ((copyS (n-1) Abst) +-+ upDI n xs)

par2ser2DI xs ys  = par2ser2DI' (zipSY xs ys)
  where par2ser2DI' NullS = NullS
        par2ser2DI' ((x,y):-xys) = x:-y:-par2ser2DI' xys

par2ser3DI xs ys zs = par2ser3DI' (zip3SY xs ys zs)
  where par2ser3DI' NullS = NullS
        par2ser3DI' ((x,y,z):-xyzs) = x:- y :-z :- par2ser3DI' xyzs

par2ser4DI ws xs ys zs = par2ser4DI' (zip4SY ws xs ys zs)
  where par2ser4DI' NullS = NullS
```

```
        par2ser4DI' ((w,x,y,z):-wxyzs)
            = w:-x:-y:-z:- par2ser4DI' wxyzs

ser2par2DI = group2SY . delaynSY Abst 2 . mapSY abstExt

ser2par3DI = group3SY . delaynSY Abst 3 . mapSY abstExt

ser2par4DI = group4SY . delaynSY Abst 4 . mapSY abstExt


par2serxDI = par2serxDI' . zipxSY
  where par2serxDI' NullS    = NullS
        par2serxDI' (xv:-xs) = (signal . fromVector) xv
                                  +-+ par2serxDI' xs

ser2parxDI n = unzipxSY . delaySY (copyV n Abst)
                        . filterAbstDI . group n

group2SY NullS = NullS
group2SY (x:-NullS) = NullS
group2SY (x:-y:-xys) = (x, y) :- group2SY xys

group3SY NullS = NullS
group3SY (x:-NullS) = NullS
group3SY (x:-y:-NullS) = NullS
group3SY (x:-y:-z:-xyzs) = (x, y, x) :- group3SY xyzs

group4SY NullS = NullS
group4SY (w:-NullS) = NullS
group4SY (w:-x:-NullS) = NullS
group4SY (w:-x:-y:-NullS) = NullS
group4SY (w:-x:-y:-z:-wxyzs) = (w, x, y, z) :- group4SY wxyzs


filterAbstDI :: Signal (AbstExt a) -> Signal a
filterAbstDI NullS          = NullS
filterAbstDI (Abst:-xs)     = filterAbstDI xs
filterAbstDI ((Prst x):-xs) = x :- filterAbstDI xs

group n xs = mapSY (output n) (scanlSY (addElement n)  (NullV, 0) xs)
             where addElement m (vs, n) x | n < m  = (vs <: x, n+1)
                                          | n == m = (unitV x, 1)
                   output m (vs, n) | m == n = Prst vs
                                    | m /= n = Abst
```

A.5 Application Libraries

A.5.1 The Module SynchronousProcessLib

Overview

The synchronous process library SynchronousProcessLib defines processes for the synchronous computational model. It is based on the synchronous library SynchronousLib.

```
module SynchronousProcessLib(
                                module SynchronousLib,
                                module Signal,
                                module AbsentExt,
                                fifoDelaySY, finiteFifoDelaySY,
                                memorySY, mergeSY, groupSY, counterSY
                              ) where

import SynchronousLib
import Signal
import AbsentExt
import Queue
import Memory
```

Processes

The library defines the following processes:

```
fifoDelaySY             :: Signal [a] -> Signal (AbstExt a)
finiteFifoDelaySY       :: Int -> Signal [a] -> Signal (AbstExt a)
memorySY                :: Int -> Signal (Access a) -> Signal (AbstExt a)
mergeSY                 :: Signal (AbstExt a) -> Signal (AbstExt a)
                           -> Signal (AbstExt a)
counterSY               :: (Enum a, Ord a) => a -> a -> Signal a
```

The process fifoDelaySY implements a synchronous model of a FIFO with infinite size, while the process finiteFifoDelaySY implements a FIFO with finite size. Both FIFOs take a list of values at each event cycle and output one value. There is a delay of one cycle. The process memorySY implements a synchronous memory. It uses access functions of the type Read adr and Write adr value. The process mergeSY merges two input signals into a single signal. The process has an internal buffer in order to prevent loss of data. The process is deterministic and outputs events according to their time tag. If there are two valid values at on both signals. The value of the first signal is output first. The function

`groupSY` groups values into a vector of size n, which takes n cycles. While the grouping takes place the output from this process consists of absent values. The process `counter` implements a counter, that counts from `min` to `max`. The process `counterS` has no input and its output is an infinite signal.

Implementation of Processes

```
fifoDelaySY xs          =  mooreSY fifoState fifoOutput (queue []) xs

fifoState               :: Queue a -> [a] -> Queue a
fifoState (Q []) xs     =  (Q xs)
fifoState q xs          =  fst (popQ (pushListQ q xs))

fifoOutput              :: Queue a -> AbstExt a
fifoOutput (Q [])       =  Abst
fifoOutput (Q (x:xs))   =  Prst x

finiteFifoDelaySY n xs
   = mooreSY fifoStateFQ fifoOutputFQ (finiteQueue n []) xs

fifoStateFQ :: FiniteQueue a -> [a] -> FiniteQueue a
fifoStateFQ (FQ n []) xs =  (FQ n xs)
fifoStateFQ q xs         =  fst (popFQ (pushListFQ q xs))

fifoOutputFQ :: FiniteQueue a -> AbstExt a
fifoOutputFQ (FQ n [])   =  Abst
fifoOutputFQ (FQ n (x:xs)) =  Prst x

memorySY size xs        =  mealySY ns o (newMem size) xs
  where
     ns mem (Read x)    =  memState mem (Read x)
     ns mem (Write x v) =  memState mem (Write x v)
     o mem (Read x)     =  memOutput mem (Read x)
     o mem (Write x v)  =  memOutput mem (Write x v)


mergeSY xs ys           =  moore2SY mergeState mergeOutput [] xs ys
   where
         mergeState []      Abst     Abst    = []
         mergeState []      Abst    (Prst y) = [y]
         mergeState []     (Prst x)  Abst    = [x]
         mergeState []     (Prst x) (Prst y) = [x, y]
         mergeState (u:us)  Abst     Abst    = us
```

```
        mergeState (u:us)  Abst    (Prst y) = us ++ [y]
        mergeState (u:us) (Prst x)  Abst    = us ++ [x]
        mergeState (u:us) (Prst x) (Prst y) = us ++ [x, y]

        mergeOutput []     = Abst
        mergeOutput (u:us) = Prst u

groupSY k = mooreSY f g s0
  where
    s0 = NullV
    f v x | lengthV v == 0 = unitV x
          | lengthV v == k = unitV x
          | otherwise      = v <: x
    g v   | lengthV v == 0 = Prst NullV
    g v   | lengthV v == k = Prst v
    g v   | otherwise      = Abst

counterSY m n = sourceSY f m
  where
    f x | x >= n    = m
        | otherwise = succ x
```

A.5.2 The Module `FIR`

A FIR-filter is described by the following equation, which is illustrated in Figure A.4:

$$y_n = \sum_{m=0}^{k} x_{n-m} h_m \tag{A.4}$$

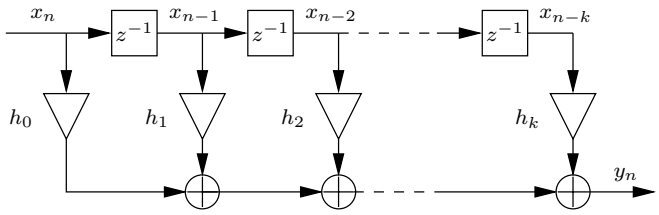


Figure A.4. FIR-filter

The state of the FIR-Filter can be seen as a shift register with parallel output. The first element in the shift register at cycle n is x_n and the last element is x_{n-k}. In the next cycle a new value x_{n+1} is shifted into the register from the left, all other

elements are shifted one place to the right, and the value x_{n-k} is discarded. We model the shift register with the process $shiftreg_k$. The process is based on the process constructor $scanlSY$ which which takes the shift function `shiftrV` as first argument and an initial vector of size $k+1$ with zeroes as initial values.

The output of the shiftregister, a signal of vectors, is transformed with the process $unzipxSY$ into a vector of signals. Then the process $innerProd$ calculates the inner product of the coefficient vector h and the output of the shift register. The process is implemented by the process constructor $zipWithSY$ that takes a parametized function $ipV(h)$ as arguments.

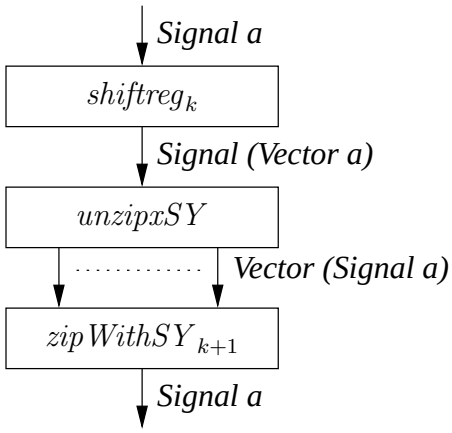


Figure A.5. FIR-filter model

```
module FIR (fir) where

import SynchronousLib

fir h = innerProd h . sipo k 0.0
    where k = lengthV h

sipo n s0 = unzipxSY . scanldSY shiftrV initState
    where initState = copyV n s0

innerProd h = zipWithxSY (ipV h)
   where ipV NullV   NullV   = 0
         ipV (h:>hv) (x:>xv) = h*x + ipV hv xv
```

All kinds of FIR-filters can now be modeled by means of fir. The only argument needed is the list of coefficients, which is given as a vector of any size. To

illustrate this, an 8-th order band pass filter is modeled as follows.

```
bp = fir (vector [
   0.06318761339784, 0.08131651217682,
   0.09562326700432, 0.10478344432968,
   0.10793629404886, 0.10478344432968,
   0.09562326700432, 0.08131651217682,
   0.06318761339784 ])
```

Appendix B

The Equalizer Specification Model

This chapter gives the executable code of the equalizer specification model. The code is written in literate Haskell style, which makes it possible to include LaTeX code for documentation. Only those parts of the literate program that are entirely enclosed between `\begin{code} ... \end{code}` are treated as program text; all other lines are comments. This allows to include usual LaTeX text, but also figures and equations as comments.

B.1 The Module `Equalizer`

B.1.1 Overview

The main task of the equalizer system is to adjust the audio signal according to the $ButtonControl$, that works as a user interface. In addition, the bass level must not exceed a predefined threshold to avoid damage to the speakers.

This specification can be naturally decomposed into four functions shown in Figure B.1. The subsystems $ButtonControl$ and $DistortionControl$, are control dominated (grey shaded), while the $AudioFilter$ and the $AudioAnalyzer$ are data flow dominated subsystems.

The $ButtonControl$ subsystem monitors the button inputs and the override signal from the subsystem $DistortionControl$ and adjusts the current bass and treble levels. This information is passed to the subsystem $AudioFilter$, which receives the audio input, and filters and amplifies the audio signal according to the current bass and treble levels. This signal, the output signal of the equalizer, is analyzed

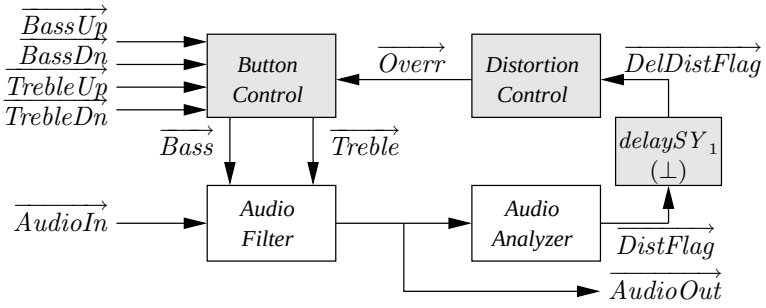


Figure B.1. Subsystems of the $Equalizer$

by the $AudioAnalyzer$ subsystem, which determines, whether the bass exceeds a predefined threshold. The result of this analysis is passed to the subsystem $DistortionControl$, which decides, if a minor or major violation is encountered and issues the necessary commands to the $ButtonControl$ subsystem.

The frequency characteristics of the $Equalizer$ is adjusted by the coefficients for the three FIR-filters in the $AudioFilter$.

```
module Equalizer(equalizer) where

import ForSyDeStdLib
import EqualizerTypes
import ButtonControl
import DistortionControl
import AudioAnalyzer
import AudioFilter
```

The structure of the equalizer is expressed as a network of blocks:

```
equalizer lpCoeff bpCoeff hpCoeff dftPts
          bassUp bassDn trebleUp trebleDn input = output
  where
    (bass, treble)  = buttonControl overrides bassUp bassDn
                                    trebleUp trebleDn
    output          = audioFilter lpCoeff bpCoeff hpCoeff bass
                                  treble input
    distFlag        = audioAnalyzer dftPts output
    overrides       = distortionControl delayedDistFlag
    delayedDistFlag = delaySY Abst distFlag
```

Since the equalizer contains a feedback loop, the signal $DistFlag$ is delayed one event cycle using the initial value $\perp$.

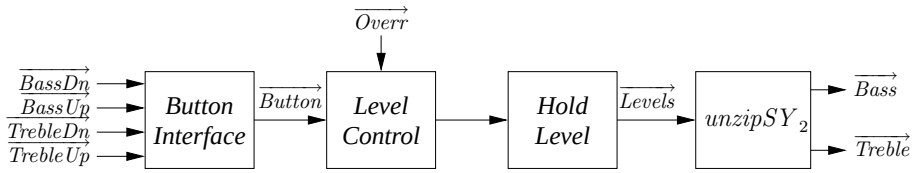


Figure B.2. The Subsystem *ButtonControl*

B.2 The Module `ButtonControl`

B.2.1 Overview

The subsystem *ButtonControl* works as a user interface in the equalizer system. It receives the four input signals $\overrightarrow{BassDn}$, $\overrightarrow{BassUp}$, $\overrightarrow{TrebleDn}$, $\overrightarrow{TrebleUp}$ and the override signal *Override* from the *DistortionControl* and calculates the new bass and treble values for the output signals $\overrightarrow{Bass}$ and $\overrightarrow{Treble}$. The subsytem contains the main processes *ButtonInterface* and *LevelControl*. The process *LevelControl* outputs a new value, if either the signal $\overrightarrow{Button}$ or the signal $\overrightarrow{Overr}$ is present, otherwise the output value is absent. The process *HoldLevel* is modeled by means of $holdSY(0.0, 0.0)$ that outputs the last present value, if the input value is absent. The process $unzipSY$ transforms a signal of tuples (the current bass and treble level) into a tuple of signals (a bass and a treble signal).

```
module ButtonControl (buttonControl) where

import ForSyDeStdLib
import EqualizerTypes

data State   = Operating
             | Locked deriving(Eq, Show)
type Level   = Double
type Bass    = Level
type Treble  = Level

buttonControl :: Signal (AbstExt OverrideMsg) -> Signal (AbstExt Sensor)
              -> Signal (AbstExt Sensor) -> Signal (AbstExt Sensor)
              -> Signal (AbstExt Sensor) -> (Signal Bass,Signal Treble)
buttonControl overrides bassDn bassUp trebleDn trebleUp
    = (bass, treble)
      where (bass, treble) = unzipSY levels
            levels = ((holdSY (0.0, 0.0)) `funComb2` levelControl)
                          button overrides
            button = buttonInterface bassDn bassUp trebleDn trebleUp
```

B.2.2 The Process *ButtonInterface*

The *ButtonInterface* monitors the four input buttons $\overrightarrow{BassDn}$, $\overrightarrow{BassUp}$, $\overrightarrow{TrebleDn}$, $\overrightarrow{TrebleUp}$ and indicates if a button is pressed. If two or more buttons are pressed the conflict is resolved by the priority order of the buttons.

```
buttonInterface :: Signal (AbstExt Sensor) -> Signal (AbstExt Sensor)
                -> Signal (AbstExt Sensor) -> Signal (AbstExt Sensor)
                -> Signal (AbstExt Button)
buttonInterface bassUp bassDn trebleUp trebleDn
   = zipWith4SY f bassUp bassDn trebleUp trebleDn
     where f (Prst Active) _ _ _ = Prst BassUp
           f _ (Prst Active) _ _ = Prst BassDn
           f _ _ (Prst Active) _ = Prst TrebleUp
           f _ _ _ (Prst Active) = Prst TrebleDn
           f _ _ _ _             = Abst
```

B.2.3 The Process *LevelControl*

The process has a local state that consists of a mode and the current values for the bass and treble levels (Figure B.3). The *LevelControl* has two modes, in the mode Operating the bass and treble values are stepwise changed in 0.2 steps. However, there exists maximum and minimum values which are -5.0 and +5.0. The process enters the mode Locked when the Override input has the value Lock. In this mode an additional increase of the bass level is prohibitet and even decreased by 1.0 in case the Override signal has the value CutBass. The subsystem returns to the Operating mode on the override value Release. The output of the process is an absent extended signal of tuples with the current bass and treble levels.

```
levelControl :: Signal (AbstExt Button) -> Signal (AbstExt OverrideMsg)
             -> Signal (AbstExt (Bass,Treble))
levelControl button overrides
  = mealy2SY nextState output (initState, initLevel) button overrides

nextState :: (State,(Double,Double)) -> AbstExt Button
          -> AbstExt OverrideMsg -> (State,(Double,Double))
nextState (state, (bass, treble)) button override
  = (newState, (newBass, newTreble)) where
     newState = if state == Operating then
                   if override == Prst Lock then
                      Locked
                   else
                      Operating
```

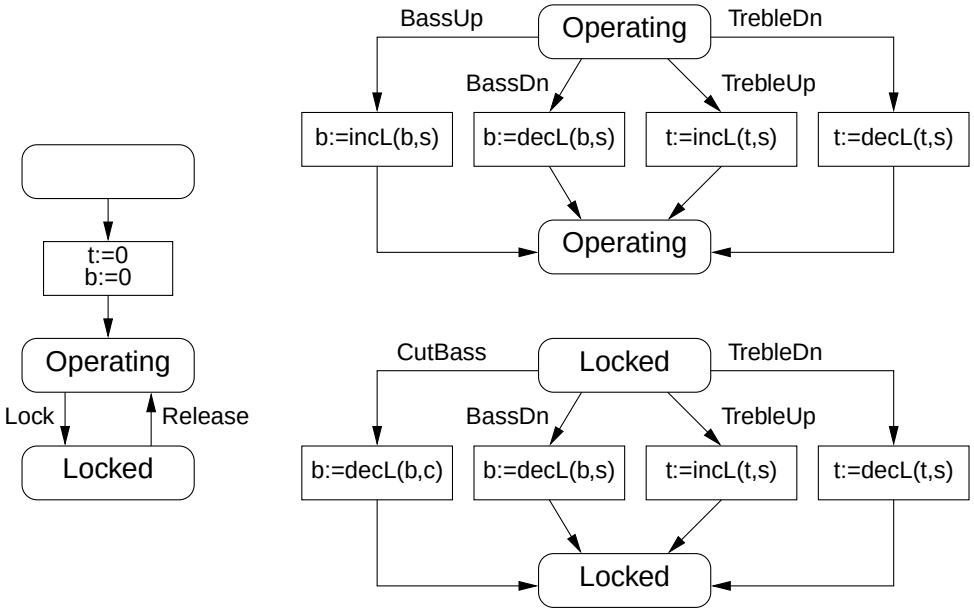


Figure B.3. The State Diagram of the Process *LevelControl*

```
            else
               if override == Prst Release then
                  Operating
               else
                  Locked

newBass = if state == Locked then
             if override == Prst CutBass then
                decreaseLevel bass cutStep
             else
                if button == Prst BassDn then
                   decreaseLevel bass step
                else
                   bass
          else -- state = Operating
             if button == Prst BassDn then
                decreaseLevel bass step
             else
                if button == Prst BassUp then
                   increaseLevel bass step
                else
```

```
                        bass

    newTreble = if button == Prst TrebleDn then
                   decreaseLevel treble step
                else
                   if button == Prst TrebleUp then
                      increaseLevel treble step
                   else
                      treble

output :: (a, (Bass, Treble)) -> AbstExt Button -> AbstExt OverrideMsg
          -> AbstExt (Bass,Treble)
output _           Abst Abst = Abst
output (_, levels) _    _    = Prst levels
```

The process uses the following initial values.

```
initState =  Operating
initLevel =  (0.0, 0.0)
maxLevel  =  5.0
minLevel  = -5.0
step      =  0.2
cutStep   =  1.0
```

The process uses the following auxiliary functions.

```
decreaseLevel :: Level -> Level -> Level
decreaseLevel level step = if reducedLevel >= minLevel then
                              reducedLevel
                           else
                              minLevel
                           where reducedLevel = level - step

increaseLevel :: Level -> Level -> Level
increaseLevel level step = if increasedLevel <= maxLevel then
                              increasedLevel
                           else
                              maxLevel
                           where increasedLevel = level + step
```

B.3 The Module `DistortionControl`

The block *DistortionControl* is directly developed from the SDL-specification, that has been used for the MASCOT-model [20]. The specification is shown in

Figure B.4.

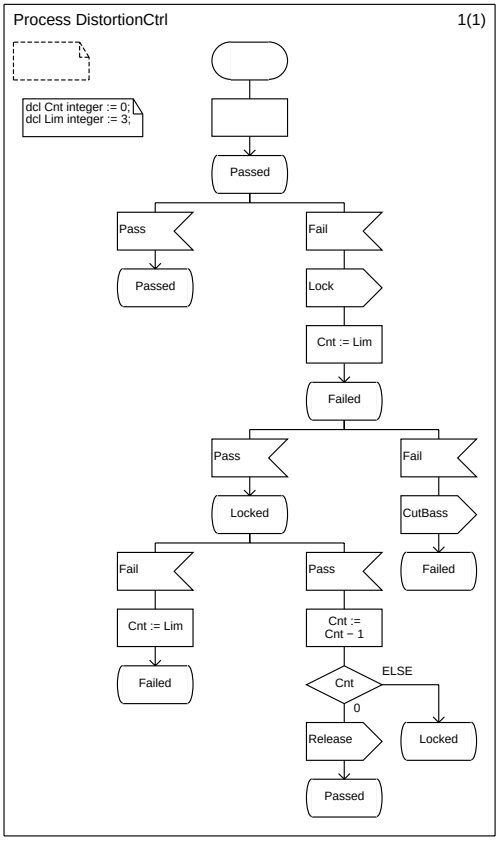


Figure B.4. SDL-description of *Distortion Control*

The $DistortionControl$ is a single FSM, which is modeled by means of the skeleton $mealySY$. The global state is not only expressed by the explicit states - Passed, Failed and Locked -, but also by means of the variable cnt. The state machine has two possible input values, Pass and Fail, and three output values, Lock, Release and CutBass.

The $mealySY$ creates a process that can be interpreted as a Mealy-machine. It takes two functions, $nxtSt$ to calculate the next state and out to calculate the output. The state is represented by a pair of the explicit state and the variable cnt. The initial state is the same as in the SDL-model, given by the tuple $(\mathsf{Passed}, 0)$. The $nxtSt$ function uses pattern matching. Whenever an input value matches a pattern of the $nxtSt$ function the corresponding right hand side is evaluated, giving

the next state. An event with an absent value leaves the state unchanged. The output function is modeled in a similar way. The output is absent, when no output message is indicated in the SDL-model.

```
module DistortionControl (distortionControl) where

import ForSyDeStdLib
import EqualizerTypes

data State =  Passed
            | Failed
            | Locked

distortionControl ::  Signal (AbstExt AnalyzerMsg)
                    -> Signal (AbstExt OverrideMsg)

distortionControl distortion
   = mealySY nxtSt out (Passed, 0) distortion

lim = 3

--     State       Input       NextState
nxtSt (state, cnt) (Abst)      = (state,cnt)
nxtSt (Passed,cnt) (Prst Pass) = (Passed,cnt)
nxtSt (Passed,_  ) (Prst Fail) = (Failed,lim)
nxtSt (Failed,cnt) (Prst Pass) = (Locked,cnt)
nxtSt (Failed,cnt) (Prst Fail) = (Failed,cnt)
nxtSt (Locked,_  ) (Prst Fail) = (Failed,lim)
nxtSt (Locked,cnt) (Prst Pass) = (newSt,newCnt)
   where newSt = if (newCnt == 0) then Passed
                                  else Locked
         newCnt = cnt - 1

--   State         Input       Output
out (Passed,_)   (Prst Pass) = Abst
out (Passed,_)   (Prst Fail) = Prst Lock
out (Failed,_)   (Prst Pass) = Abst
out (Failed,_)   (Prst Fail) = Prst CutBass
out (Locked,_)   (Prst Fail) = Abst
out (Locked,cnt) (Prst Pass) =
   if (cnt == 1) then Prst Release
                 else Abst
out _            Abst        = Abst
```

B.4 The Module `AudioFilter`

B.4.1 Overview

Figure B.5 shows the structure of the $AudioFilter$. The task of this subsystem is to amplify different frequencies of the audio signal independently according to the assigned levels. The audio signal is splitted into three identical signals, one for each frequency region. The signals are filtered and then amplified according to the assigned amplification level. As the equalizer in this design only has a bass and treble control, the middle frequencies are not amplified. The output signal from the $AudioFilter$ is the addition of the three filtered and amplified signals.

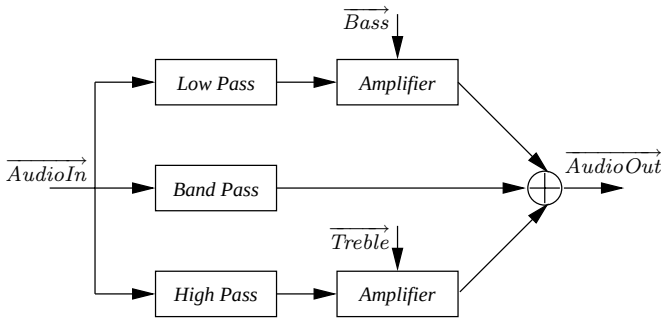


Figure B.5. Subsystems of the *Audio Filter*

We model this structure as a network of blocks directly from Figure B.5. It consists of three filters, two amplifiers and an adder. These blocks are modeled in the process layer. The $AudioFilter$ has the filter coefficients for the low pass, band pass and high pass filter as parameters.

```
module AudioFilter where

import ForSyDeStdLib

audioFilter :: Floating a => Vector a -> Vector a -> Vector a
            -> Signal a -> Signal a -> Signal a -> Signal a
audioFilter lpCoeff bpCoeff hpCoeff bass treble audioIn = audioOut
   where audioOut      = zipWith3SY add3 bassPath middlePath treblePath
         bassPath      = ((amplify bass) . lowPass) audioIn
         middlePath    = bandPass audioIn
         treblePath    = ((amplify treble) . highPass) audioIn
         lowPass       = fir lpCoeff
         bandPass      = fir bpCoeff
         highPass      = fir hpCoeff
```

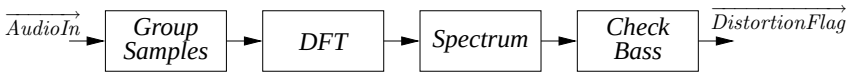


Figure B.6. The *Audio Analyzer* subsystem

```
amplify      = zipWithSY scale
add3 x y z   = x + y + z
scale x y    = y * (base ** x)
base         = 1.1
```

B.5 The Module `AudioAnalyzer`

B.5.1 Overview

The $AudioAnalyzer$ analyzes the current bass level and raises a flag when the bass level exceeds a limit.

As illutsrated in Figure B.6 the $AudioAnalyzer$ is divided into four blocks. The input signal is first grouped into samples of size N in the process $GroupSamples$ and then processed with a DFT in order to get the frequency spectrum of the signal. Then the power spectrum is calculated in $Spectrum$. In $CheckBass$ the lowest frequencies are compared with a threshold value. If they exceed this value, the output $DistortionFlag$ will have the value Fail.

Since $GroupSamples$ needs N cycles for the grouping, it produces $N - 1$ absent values $\perp$ for each grouped sample. Thus the following processes DFT, $Spectrum$ and $CheckBass$ are all Ψ-extended in order to be able to process the absent value $\perp$.

```
module AudioAnalyzer (audioAnalyzer) where

import ForSyDeStdLib
import Complex
import EqualizerTypes

limit = 1.0
nLow = 3

audioAnalyzer pts =  mapSY (psi checkBass) -- Check Bass
                   . mapSY (psi spectrum)  -- Spectrum
                   . mapSY (psi (dft pts)) -- DFT
                   . groupSY pts           -- Group Samples
                   . mapSY toComplex
```

```
spectrum =  mapV log10 . selectLow nLow . mapV power . selectHalf . dropV 1
  where
    log10 x        = log x / log 10
    selectLow n xs = takeV n xs
    selectHalf xs  = takeV half xs
              where half = floor ((lengthV xs) / 2)
    power x = (magnitude x) ^ 2

checkBass = checkLimit limit . sumV
  where
    checkLimit limit x | x > limit  = Fail
                       | otherwise  = Pass
    sumV vs = foldlV (+) 0.0 vs

toComplex x = x :+ 0
```

B.6 The Module `EqualizerTypes`

B.6.1 Overview

This module is a collection of data types that are used in the equalizer model.

```
module EqualizerTypes where

data AnalyzerMsg = Pass
                 | Fail deriving(Show, Read, Eq)

data OverrideMsg = Lock
                 | CutBass
                 | Release deriving(Show, Read, Eq)

data Sensor = Active deriving(Show, Read, Eq)

data Button = BassDn
            | BassUp
            | TrebleDn
            | TrebleUp deriving (Show, Read, Eq)
```

Appendix C

VHDL-Templates for ForSyDe Processes

This appendix gives an overview of templates that are used for the mapping of ForSyDe processes to a hardware description in VHDL. The appendix is divided into the following three parts:

- VHDL-template for processes based on combinational process constructors (Section C.1)
- VHDL-template for processes based on sequential process constructors (Section C.2)
- VHDL-template for processes based on domain interface process constructors (Section C.3)

C.1 VHDL-Templates for combinational Process Constructors

C.1.1 VHDL-Template for Processes constructed by $mapSY$

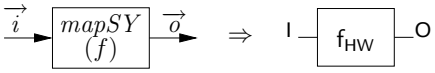


Figure C.1. Hardware implementation of $mapSY$

Figure C.1 illustrates the mapping of a process $mapSY(f)$ to hardware. The VHDL code for the template is given below.

```
PACKAGE mapSY_f_lib IS
   TYPE type_mapSY_f_i IS to_be_defined; -- Type of i
   TYPE type_mapSY_f_o IS to_be_defined; -- Type of o
   FUNCTION f(x : type_mapSY_f_i) RETURN type_mapSY_f_o;
END;

PACKAGE BODY mapSY_f_lib IS
   FUNCTION f(x : type_mapSY_f_i) RETURN type_mapSY_f_o IS
   BEGIN
      RETURN to_be_defined; -- Definition of f
   END;
END;

USE work.mapSY_f_lib.ALL;

ENTITY mapSY_f IS
   PORT(
      i : IN type_mapSY_f_i;
      o : OUT type_mapSY_f_o
      );
END;

ARCHITECTURE Comb OF mapSY_f IS
BEGIN
   o <= f(i);
END;
```

C.1.2 VHDL-Template for Processes constructed by $zipWithSY$

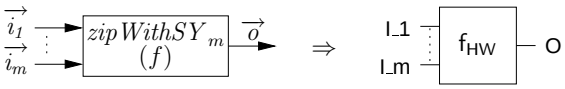


Figure C.2. Hardware implementation of $zipWithSY_m$

Figure C.2 illustrates the mapping of a process $zipWithSY_m(f)$ to hardware. The VHDL code for the template is given below ($m = 2$).

```
PACKAGE zipWithSY_f_lib IS
   TYPE type_zipWithSY_f_i1 IS to_be_defined; -- Type of i1
   TYPE type_zipWithSY_f_i2 IS to_be_defined; -- Type of i2
   TYPE type_zipWithSY_f_o IS to_be_defined; -- Type of o
   FUNCTION f(x: type_zipWithSY_f_i1; y : type_zipWithSY_f_i2)
     RETURN  type_zipWithSY_f_o;
END;
```

```
PACKAGE BODY zipWithSY_f_lib IS
   FUNCTION f(x: type_zipWithSY_f_i1; y : type_zipWithSY_f_i2)
     RETURN  type_zipWithSY_f_o IS
   BEGIN
      RETURN -- to be defined; Definition of f
   END;
END;

USE work.zipWithSY_f_lib.all;

ENTITY zipWithSY_f is
port
   (
      i1 : IN type_zipWithSY_f_i1;
      i2 : IN type_zipWithSY_f_i2;
      o  : OUT type_zipWithSY_f_o
      );
end;

architecture Comb OF zipWithSY_f is
   begin
      o <= f(i1, i2);
   end;
```

C.2 VHDL-Templates for sequential Process Constructors

C.2.1 VHDL-Template for Processes constructed by $delaySY$

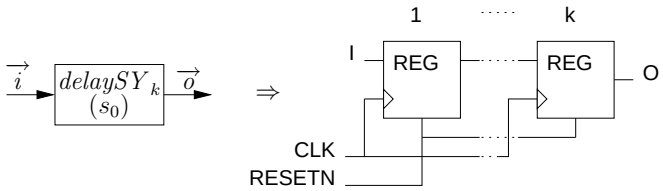


Figure C.3. Hardware implementation of $delaySY_k$

Figure C.3 illustrates the mapping of a process $delaySY_1(s_0)$ to hardware. The VHDL code for the template is given below.

```
PACKAGE delaySY_1_lib IS
   TYPE type_delaySY_1_i IS to_be_defined; -- Definition of i
   TYPE type_delaySY_1_o IS to_be_defined; -- Definition of o
   CONSTANT s0 : type_delaySY_1_o := to_be_defined;
                                  --Initial State
```

```
END;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.delaySY_1_lib.ALL;

ENTITY delaySY_1 IS
   PORT
      (
         i : IN type_delaySY_1_i;
         o : OUT type_delaySY_1_o;
         resetn : IN std_logic;
         clk : IN std_logic
         );
END;

ARCHITECTURE Seq OF delaySY_1 IS
  SIGNAL s : type_delaySY_1_o := s0;
BEGIN
   PROCESS(clk, resetn)
   BEGIN
      IF resetn = '0' THEN
         s <= s0;
      ELSIF rising_edge(clk) THEN
         s <= i;
      END IF ;
   END process;

   o <= s;
END;
```

C.2.2 VHDL-Template for Processes constructed by $scanlSY$

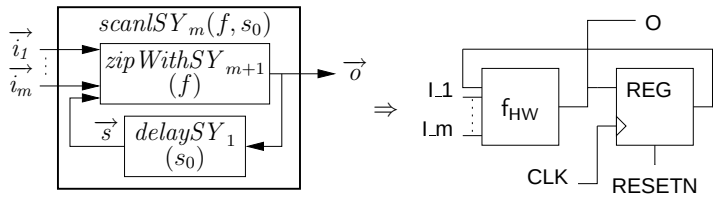


Figure C.4. Hardware implementation of $scanlSY_m$

Figure C.4 illustrates the mapping of a process $scanlSY_m(f, s_0)$ to hardware. The VHDL code for the template is given below ($m = 1$).

```
PACKAGE scanlSY_f_lib IS
   TYPE type_scanlSY_f_i IS to_be_defined; -- Type of i
   TYPE type_scanlSY_f_o IS to_be_defined; -- Type of o
   SUBTYPE type_scanlSY_f_state IS type_scanlSY_f_o;
   CONSTANT s0 : type_scanlSY_f_state := to_be_defined;
                                      -- initial State
   FUNCTION f (
      i       : type_scanlSY_f_i;
      state   : type_scanlSY_f_state
      ) RETURN type_scanlSY_f_o;

END;

PACKAGE BODY scanlSY_f_lib IS

   FUNCTION f (
      i       : type_scanlSY_f_i;
      state   : type_scanlSY_f_state
      ) RETURN type_scanlSY_f_o IS
   BEGIN
      RETURN to_be_defined; -- Definition of f
   END;

END scanlSY_f_lib;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.scanlSY_f_lib.ALL;
LIBRARY synopsys;
USE synopsys.attributes.ALL;

ENTITY scanlSY_f IS

   PORT (
      i     : IN  type_scanlSY_f_i;
      o      : OUT type_scanlSY_f_o;
      clk    : IN  std_logic;
      resetn : IN  std_logic);

END scanlSY_f;

ARCHITECTURE Seq OF scanlSY_f IS
   SIGNAL state, nextstate : type_scanlSY_f_state;
   ATTRIBUTE state_vector        : string;
   ATTRIBUTE state_vector OF Seq : ARCHITECTURE IS "state";
BEGIN  -- Seq
```

```
   PROCESS (clk, resetn)
   BEGIN  -- PROCESS
      IF resetn = '0' THEN -- asynchronous reset (active low)
         state <= s0;
      ELSIF clk'event AND clk = '1' THEN  -- rising clock edge
         state <= nextstate;
      END IF;
   END PROCESS;

   PROCESS (i,state)
   BEGIN  -- PROCESS
      nextstate <= f(i,state);
   END PROCESS;

   o <= nextstate;
END Seq;
```

C.2.3 VHDL-Template for Processes constructed by $scanldSY$

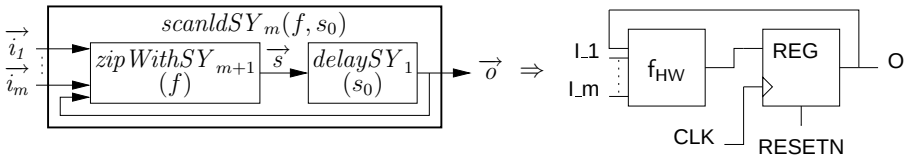


Figure C.5. Hardware implementation of $scanldSY_m$

Figure C.5 illustrates the mapping of a process $scanldSY_m(fp, s_0)$ to hardware. The VHDL code for the template is given below ($m = 1$).

```
PACKAGE scanldSY_f_lib IS
   TYPE type_scanldSY_f_i IS to_be_defined; -- Type of i
   TYPE type_scanldSY_f_o IS to_be_defined; -- Type of o;
   SUBTYPE type_scanldSY_f_state IS type_scanldSY_f_o;
   CONSTANT s0 : type_scanldSY_f_state := to_be_defined;
                                          -- Initial State
   FUNCTION f (
      i       : type_scanldSY_f_i;
      state   : type_scanldSY_f_state
      ) RETURN type_scanldSY_f_o;

END;

PACKAGE BODY scanldSY_f_lib IS
```

```
   FUNCTION f (
      i       : type_scanldSY_f_i;
      state   : type_scanldSY_f_state
      ) RETURN type_scanldSY_f_o IS
   BEGIN
      RETURN i and state;
   END;

END scanldSY_f_lib;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.scanldSY_f_lib.ALL;
LIBRARY synopsys;
USE synopsys.attributes.ALL;

ENTITY scanldSY_f IS

   PORT (
      i     : IN  type_scanldSY_f_i;
      o      : OUT type_scanldSY_f_o;
      clk    : IN  std_logic;
      resetn : IN  std_logic);

END scanldSY_f;

ARCHITECTURE Seq OF scanldSY_f IS
   SIGNAL state, nextstate : type_scanldSY_f_state;
   ATTRIBUTE state_vector        : string;
   ATTRIBUTE state_vector OF Seq : ARCHITECTURE IS "state";
BEGIN  -- Seq

   PROCESS (clk, resetn)
   BEGIN  -- PROCESS
      IF resetn = '0' THEN  -- asynchronous reset (active low)
         state <= s0;
      ELSIF clk'event AND clk = '1' THEN  -- rising clock edge
         state <= nextstate;
      END IF;
   END PROCESS;

   PROCESS (i,state)
   BEGIN  -- PROCESS
      nextstate <= f(i,state);
      o <= state;
   END PROCESS;
```

```
END Seq;
```

C.2.4 VHDL-Template for Processes constructed by $mooreSY$

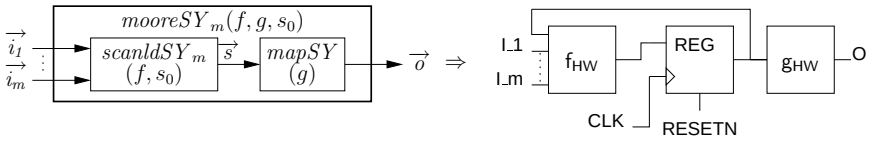


Figure C.6. Hardware implementation of $mooreSY_m$

Figure C.6 illustrates the mapping of a process $mooreSY_m(f, g, s_0)$ to hardware. The VHDL code for the template is given below ($m = 1$).

```
PACKAGE mooreSY_f_g_lib IS
   TYPE type_mooreSY_f_g_i IS to_be_defined; -- Type of i
   TYPE type_mooreSY_f_g_o IS to_be_defined; -- Type of o
   TYPE type_mooreSY_f_g_state IS to_be_defined; -- Type of state
   CONSTANT s0 : type_mooreSY_f_g_state := to_be_defined;
                                        -- Initial State
   FUNCTION f (
      i       : type_mooreSY_f_g_i;
      state   : type_mooreSY_f_g_state
      ) RETURN type_mooreSY_f_g_state;

   FUNCTION g (
      state    : type_mooreSY_f_g_state
      ) RETURN type_mooreSY_f_g_o;
END;

PACKAGE BODY mooreSY_f_g_lib IS

   FUNCTION f (
      i       : type_mooreSY_f_g_i;
      state    : type_mooreSY_f_g_state
      ) RETURN type_mooreSY_f_g_state IS
   BEGIN
      RETURN to_be_defined; --Definition of f
   END;

   FUNCTION g(
      state    : type_mooreSY_f_g_state
      ) RETURN type_mooreSY_f_g_o IS
   BEGIN
      RETURN to_be_defined; -- Definition of g
```

```
   END;

END mooreSY_f_g_lib;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.mooreSY_f_g_lib.ALL;
LIBRARY synopsys;
USE synopsys.attributes.ALL;

ENTITY mooreSY_f_g IS

   PORT (
      i     : IN  type_mooreSY_f_g_i;
      o      : OUT type_mooreSY_f_g_o;
      clk    : IN  std_logic;
      resetn : IN  std_logic);

END mooreSY_f_g;

ARCHITECTURE Seq OF mooreSY_f_g IS
   SIGNAL state, nextstate : type_mooreSY_f_g_state;
   ATTRIBUTE state_vector        : string;
   ATTRIBUTE state_vector OF Seq : ARCHITECTURE IS "state";
BEGIN  -- Seq

   PROCESS (clk, resetn)
   BEGIN  -- PROCESS
      IF resetn = '0' THEN  -- asynchronous reset (active low)
         state <= s0;
      ELSIF clk'event AND clk = '1' THEN  -- rising clock edge
         state <= nextstate;
      END IF;
   END PROCESS;

   PROCESS (i,state)
   BEGIN  -- PROCESS
      nextstate <= f(i,state);
      o <= g(state);
   END PROCESS;

END Seq;
```

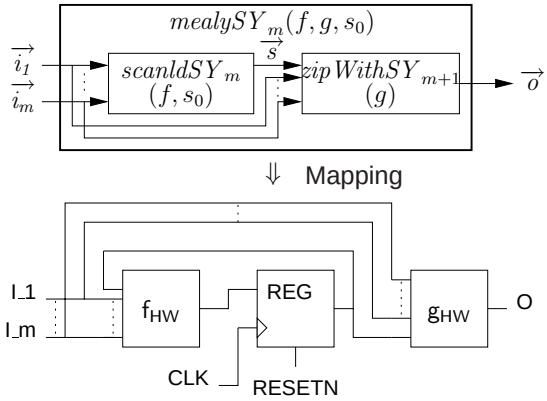


Figure C.7. Hardware implementation of $mealySY_m$

C.2.5 VHDL-Template for Processes constructed by $mealySY$

Figure C.7 illustrates the mapping of a process $mealySY_m(f, g, s_0)$ to hardware. The VHDL code for the template is given below ($m = 1$).

```
PACKAGE mealySY_f_g_lib IS
   TYPE type_mealySY_f_g_i IS to_be_defined; -- Type of i
   TYPE type_mealySY_f_g_o is to_be_defined; -- Type of o
   TYPE type_mealySY_f_g_state IS to_be_defined; -- Type of state
   CONSTANT s0 : type_mealySY_f_g_state := to_be_defined;
                                          -- Initial State
   FUNCTION f (
      i     : type_mealySY_f_g_i;
      state : type_mealySY_f_g_state
      ) RETURN type_mealySY_f_g_state;

   FUNCTION g (
      i     : type_mealySY_f_g_i;
      state : type_mealySY_f_g_state
      ) RETURN type_mealySY_f_g_o;
END;

PACKAGE BODY mealySY_f_g_lib IS

   FUNCTION f (
      i     : type_mealySY_f_g_i;
      state : type_mealySY_f_g_state
      ) RETURN type_mealySY_f_g_state IS
   BEGIN
      RETURN to_be_defined; -- Definition of f;
   END;
```

```
   FUNCTION g(
      i     : type_mealySY_f_g_i;
      state : type_mealySY_f_g_state
      ) RETURN type_mealySY_f_g_o IS
   BEGIN
      RETURN to_be_defined; -- Definition of g;
   END;

END mealySY_f_g_lib;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.mealySY_f_g_lib.ALL;
LIBRARY synopsys;
USE synopsys.attributes.ALL;

ENTITY mealySY_f_g IS

   PORT (
      i     : IN  type_mealySY_f_g_i;
      o      : OUT type_mealySY_f_g_o;
      clk    : IN  std_logic;
      resetn : IN  std_logic);

END mealySY_f_g;

ARCHITECTURE Seq OF mealySY_f_g IS
   SIGNAL state, nextstate : type_mealySY_f_g_state;
   ATTRIBUTE state_vector        : string;
   ATTRIBUTE state_vector OF Seq : ARCHITECTURE IS "state";
BEGIN  -- Seq

   PROCESS (clk, resetn)
   BEGIN  -- PROCESS
      IF resetn = '0' THEN   -- asynchronous reset (active low)
         state <= s0;
      ELSIF clk'event AND clk = '1' THEN  -- rising clock edge
         state <= nextstate;
      END IF;
   END PROCESS;

   PROCESS (i,state)
   BEGIN  -- PROCESS
      nextstate <= f(i,state);
   END PROCESS;
```

```
   o <= g(i,state);
END Seq;
```

C.3 VHDL-Templates for Domain Interfaces

C.3.1 VHDL-Template for Processes constructed by $downDI$

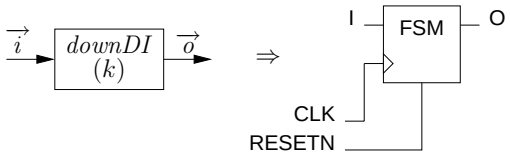


Figure C.8. Hardware implementation of $downDI(k)$

A process $downDI(k)$ is implemented with a counter that counts from 1 to k (Figure C.8). Only when the counter has the value 1, the input value is send to the output. In practice this means that the output clock is divided by k.

The VHDL-template for $k = 4$ is given below.

```
PACKAGE downDI_4_lib IS

   TYPE Type_downDI_4 IS to_be_defined; -- Type for i and o

END downDI_4_lib;

LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE work.downDI_4_lib.ALL;

ENTITY downDI_4 IS

   PORT (
      i : IN  type_downDI_4;
      clk : IN std_logic;
      resetn : IN std_logic;
      o : OUT type_downDI_4);

END downDI_4;

ARCHITECTURE Seq OF downDI_4 IS

   SIGNAL div_clk : std_logic;
   SIGNAL cnt : integer RANGE 0 TO 4;
```

```
   CONSTANT n : integer  := 4;
BEGIN  -- Seq

   PROCESS (clk, resetn)
   BEGIN  -- PROCESS
      IF resetn = '0' THEN
         cnt <= 1;
      ELSE
         IF rising_edge(clk) THEN
            CASE cnt IS
               WHEN n => cnt <= 1;
                         div_clk <= '1';
               WHEN OTHERS => cnt <= cnt + 1;
                              div_clk <= '0';
            END CASE;
         END IF;
      END IF;
   END PROCESS;

   PROCESS (i,div_clk)
   BEGIN  -- PROCESS
      IF rising_edge(div_clk) THEN
         o <= i;
      END IF;
   END PROCESS;

END Seq;
```

C.3.2 VHDL-Template for Processes constructed by $upDI$

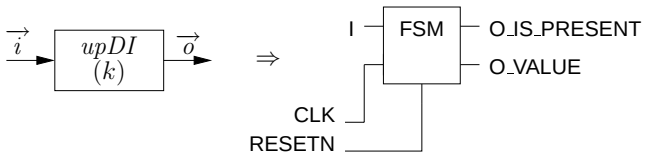


Figure C.9. Hardware implementation of $upDI(k)$

A process $upDI(k)$ is implemented with a counter that counts from 1 to k (Figure C.9). When the counter has the value 1, the output value `o.is_present` is `True` and the output value `o.value` gets the input value. Otherwise the output value is absent (`o.is_present` has the value `False`). The clock `clk` has a frequency that is k-times higher than the corresponding input frequency and determines the clock of the output signal.

The VHDL-template for $k = 4$ is given below.

```
PACKAGE upDI_4_lib IS

   TYPE type_upDI_4 is to_be_defined; -- Type for o and value of i
   TYPE type_upDI_4_o IS RECORD
                           value      : type_upDI_4;
                           is_present : boolean;
                         END RECORD;
   TYPE type_upDI_4_i IS type_upDI_4;

END upDI_4_lib;

LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE work.upDI_4_lib.ALL;

ENTITY upDI_4 IS

   PORT (
      i      : IN  type_upDI_4_i;
      o      : OUT type_upDI_4_o;
      clk    : IN  std_logic;
      resetn : IN  std_logic);

END upDI_4;

ARCHITECTURE Seq OF upDI_4 IS
   CONSTANT n : integer := 4;
   SIGNAL cnt : integer RANGE 1 TO n := 1;
BEGIN  -- Seq

   PROCESS (clk, resetn)
   BEGIN  -- PROCESS
      IF resetn = '0' THEN  -- asynchronous reset (active low)
         cnt <= 1;
      ELSIF clk'event AND clk = '1' THEN  -- rising clock edge
         CASE cnt  IS
            WHEN n => cnt <= 1;
            WHEN others => cnt <= cnt + 1;
         END CASE;
      END IF;
   END PROCESS;

   PROCESS (clk,cnt,i)
   BEGIN  -- PROCESS
      o.value <= i;
      CASE cnt  IS
```

```
         WHEN 1      => o.is_present <= True;
         WHEN others => o.is_present <= False;
      END CASE;
   END PROCESS;

END Seq;
```

C.3.3 VHDL-Template for Processes constructed by $p2sDI$

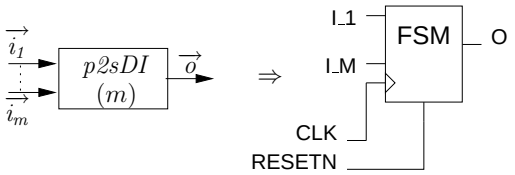


Figure C.10. Hardware implementation of $p2sDI(m)$

The process $p2sDI(m)$ is implemented as a FSM where counter counts to k. Whenever the counter is in state k, the input value I_k is forwarded to the output. The clock `clk` determines the output frequency and is k times faster than the frequency of the input signal.

The VHDL template for $k = 2$ is given below.

```
PACKAGE par2serDI_2_lib IS

   TYPE type_par2serDI_2 IS to_be_defined;
        -- Type for i1, i2, and o

END par2serDI_2_lib;

LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE work.par2serDI_2_lib.ALL;

ENTITY par2serDI_2 IS

   PORT (
      i1  : IN type_par2serDI_2;
      i2  : IN type_par2SerDI_2;
      o   : out type_par2serDI_2;
      clk : IN std_logic;
      reset_n  : IN std_logic);

END par2serDI_2;
```

```
ARCHITECTURE Seq OF par2serDI_2 IS
   SIGNAL cnt : integer RANGE 0 TO 1 := 0;
BEGIN  -- Seq

   PROCESS (clk, reset_n)
   BEGIN  -- PROCESS
      IF reset_n = '0' THEN  -- asynchronous reset (active low)
         cnt <= 0;
      ELSIF clk'event AND clk = '1' THEN  -- rising clock edge
         CASE cnt IS
            WHEN 0 => cnt <= 1;
            WHEN 1 => cnt <= 0;
            WHEN OTHERS => NULL;
         END CASE;
      END IF;
   END PROCESS;

   PROCESS (cnt, i1, i2)
   BEGIN  -- PROCESS
      CASE cnt IS
         WHEN 0 => o <= i1;
         WHEN 1 => o <= i2;
         WHEN OTHERS => NULL;
      END CASE;
   END PROCESS;

END Seq;
```

C.3.4 VHDL-Template for Processes constructed by $s2pDI$

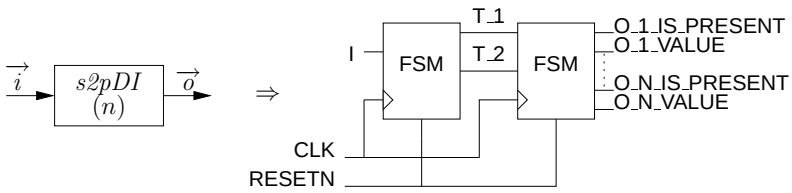


Figure C.11. Hardware implementation of $s2pDI(n)$

The process $s2pDI(n)$ is implemented with two FSMs. The first FSM directs the output in a round-robin style, while the second FSM is used for clock division to guarantee that the outputs are synchronized with the output clock. In the reset state the output values are absent.

The VHDL template for $n = 2$ is given below.

```
PACKAGE ser2parDI_2_lib IS

   SUBTYPE type_ser2parDI_2 IS to_be_defined;
   TYPE type_ser2parDI_2_o IS RECORD
                                 value      : type_ser2parDI_2;
                                 is_present : boolean;
                              END RECORD;
END ser2parDI_2_lib;

LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE work.ser2parDI_2_lib.ALL;

ENTITY ser2parDI_2 IS

   PORT (
      i      : IN  type_ser2parDI_2;
      o1     : OUT type_ser2parDI_2_o;
      o2     : OUT type_ser2parDI_2_o;
      clk    : IN  std_logic;
      resetn : IN  std_logic);

END ser2parDI_2;

ARCHITECTURE Seq OF ser2parDI_2 IS
   SIGNAL cnt    : integer RANGE 0 TO 2 := 0;
   SIGNAL cnt2   : integer RANGE 0 TO 1 := 0;
   SIGNAL t1, t2 : type_ser2parDI_2_o;
BEGIN  -- Seq

   PROCESS (clk, resetn)
   BEGIN  -- PROCESS
      IF resetn = '0' THEN  -- asynchronous reset (active low)
         cnt <= 0;
      ELSIF clk'event AND clk = '1' THEN  -- rising clock edge
         CASE cnt IS
            WHEN 0      => cnt <= 1;
            WHEN 1      => cnt <= 2;
            WHEN 2      => cnt <= 1;
            WHEN OTHERS => NULL;
         END CASE;
      END IF;
   END PROCESS;

   PROCESS (cnt, i)
   BEGIN  -- PROCESS
```

```
      t1.is_present <= true;
      t2.is_present <= true;
      CASE cnt IS
         WHEN 0 => t1.value <= i;
                   t1.is_present <= false;
                   t2.is_present <= false;
         WHEN 1      => t2.value <= i;
         WHEN 2      => t1.value <= i;
         WHEN OTHERS => NULL;
      END CASE;
   END PROCESS;

   PROCESS (clk, resetn)
   BEGIN  -- PROCESS
      IF resetn = '0' THEN  -- asynchronous reset (active low)
         cnt2 <= 0;
      ELSIF clk'event AND clk = '1' THEN  -- rising clock edge
         CASE cnt2 IS
            WHEN 0      => cnt2 <= 1;
            WHEN 1      => cnt2 <= 0;
            WHEN OTHERS => NULL;
         END CASE;
      END IF;
   END PROCESS;

   PROCESS (cnt2, t1, t2)
   BEGIN  -- PROCESS
      CASE cnt2 IS
         WHEN 0 => o1 <= t1;
                   o2 <= t2;
         WHEN OTHERS => NULL;
      END CASE;
   END PROCESS;
END Seq;
```