ForSyDe: System Design Using a Functional Language and Models of Computation

4

Ingo Sander, Axel Jantsch, and Seyed-Hosein Attarzadeh-Niaki

Abstract

The ForSyDe methodology aims to push system design to a higher level of abstraction by combining the functional programming paradigm with the theory of Models of Computation (MoCs). A key concept of ForSyDe is the use of higher-order functions as process constructors to create processes. This leads to well-defined and well-structured ForSyDe models and gives a solid base for formal analysis. The book chapter introduces the basic concepts of the ForSyDe modeling framework and presents libraries for several MoCs and MoC interfaces for the modeling of heterogeneous systems, including support for the modeling of run-time reconfigurable processes.

The formal nature of ForSyDe enables transformational design refinement using both semantic-preserving and nonsemantic-preserving design transformations. The chapter also introduces a general synthesis concept based on process constructors, which is exemplified by means of a hardware synthesis tool for synchronous ForSyDe models. Most examples in the chapter are modeled with the Haskell version of ForSyDe. However, to illustrate that ForSyDe is language-independent, the chapter also contains a short overview of SystemC-ForSyDe.

Acronyms

ASK	Amplitude Shift Key
CPS	Cyber-Physical System

I. Sander (✉)
KTH Royal Institute of Technology, Stockholm, Sweden
e-mail: ingo@kth.se

A. Jantsch
Vienna University of Technology, Vienna, Austria
e-mail: jantsch@ict.tuwien.ac.at

S.-H. Attarzadeh-Niaki
Shahid Beheshti University (SBU), Tehran, Iran
e-mail: h_attarzadeh@sbu.ac.ir

© Springer Science+Business Media Dordrecht 2017

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_5

EDA	Electronic Design Automation
ForSyDe	Formal System Design
HSCD	Hardware/Software Codesign
MoC	Model of Computation
RTL	Register Transfer Level
SLD	System-Level Design

Contents

4.1 Introduction . . . 100
4.2 The ForSyDe Modeling Framework . . . 102
4.2.1 Signals . . . 103
4.2.2 Processes . . . 104
4.2.3 ForSyDe Models of Computation . . . 109
4.2.4 Model of Computation Interfaces . . . 115
4.2.5 Reconfigurable Processes . . . 117
4.2.6 Modeling Case Study . . . 120
4.3 Transformational Design Refinement . . . 120
4.4 Synthesis of ForSyDe Models . . . 124
4.4.1 General ForSyDe Synthesis Concepts . . . 124
4.4.2 Hardware Synthesis . . . 125
4.4.3 ForSyDe Hardware Synthesis Tool . . . 126
4.5 SystemC-ForSyDe . . . 129
4.6 Related Work . . . 131
4.7 Conclusion . . . 134
References . . . 137

4.1 Introduction

Due to the ever increasing complexity of system-on-chip platforms and the continuous need for more powerful applications, industry has to cope with enormous challenges and faces exploding verification costs when designing state-of-the-art embedded systems. Still there are no systematic methods that can guarantee correct and efficient implementations at reasonable costs, in particular for systems that have to satisfy extra-functional properties like real-time behavior.

The problem is not new and well recognized. In 2007, Sangiovanni-Vincentelli discusses the problems of System-Level Design (SLD) [51] and states that "innovation in design tools has slowed down significantly as we approach a limit in the complexity of systems we can design today satisfying increasing constraints on time-to-market and correctness. The EDA community has not succeeded as of today in establishing a new layer of abstraction universally agreed upon that could provide productivity gains similar to the ones of the traditional design flow (Register Transfer Level (RTL) to GDSII) when it was first introduced."

The Formal System Design (ForSyDe) methodology addresses these challenges and aims at pushing the design entry to a considerably higher level of abstraction,

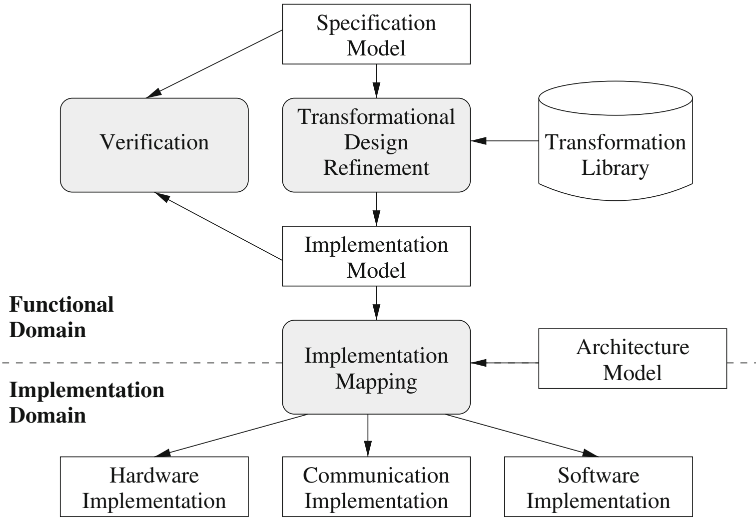

Fig. 4.1 ForSyDe system design flow [48]

by combining a formal base in form of the theory of Models of Computation (MoCs) [30] with an elegant system modeling technique based on the functional programming paradigm. This formal foundation enables the development of design transformation and synthesis techniques to convert the system model into the final implementation on a given target platform.

Figure 4.1 illustrates the ideas of the ForSyDe design flow as described in Sander's PhD thesis from 2003 [48]. The system design process starts with the development of an abstract, formal and functional *specification model* at a high abstraction level. The model is *formal* since it has a well-defined syntax and semantics. Furthermore, the model is based on well-defined models of computations, providing a clean mathematical formalism and an abstract communication model. It is *abstract* and *functional* since a system is modeled as a mathematical function of the input signals. 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 of design transformations*. Since the specification model and implementation model are based on the same semantics, the same validation and verification techniques, i.e., simulation or formal verification, 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 or Verilog for hardware and C for software running on a microcontroller. 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.

The objective of this book chapter is to give an overview of the current state of the ForSyDe design methodology, where special focus is given to the modeling framework. The whole chapter is written in a tutorial style, enabling the reader to experiment with the ForSyDe modeling and synthesis framework. Links to more detailed information are provided in the corresponding sections of the chapter. For readers not familiar with the functional programming language Haskell, a short overview of Haskell is provided in the appendix.

The chapter is structured as follows. Section 4.2 introduces the ForSyDe modeling framework and its key concepts, like signals, processes, process constructors, and MoC interfaces. Several ForSyDe MoCs are introduced and exemplified using the Haskell version of ForSyDe by concrete examples. Furthermore, the section presents the suitability of the functional paradigm to model reconfigurable processes, which in turn can be evolved to model adaptive systems. Finally, Sect. 4.2 concludes with a larger modeling case study consisting of several MoCs to illustrate the potential of the ForSyDe modeling framework. Section 4.3 introduces the ideas of transformational design refinement inside the functional domain and the use of the characteristic function to illustrate the consequences of semantic-preserving and nonsemantic-preserving design transformations to the designer. The synthesis of ForSyDe models to a target language is presented in Sect. 4.4. In particular, the section gives the general synthesis concepts that can be applied to any target architecture and exemplifies the general ideas by means of the ForSyDe hardware synthesis tool, which converts synchronous ForSyDe models to synthesizable VHDL. Section 4.5 illustrates the language independence of ForSyDe by introducing SystemC-ForSyDe, which implements the ForSyDe semantics in an industrial design language. Section 4.6 discusses related approaches, and finally Sect. 4.7 concludes the paper.

4.2 The ForSyDe Modeling Framework

In ForSyDe, a system is modeled as hierarchical concurrent process network. Processes communicate with each other only via signals. ForSyDe supports several Models of Computation (MoCs) and allows processes belonging to different models of computation to communicate via MoC interfaces as illustrated in Fig. 4.2. In order to formally describe the computational model of ForSyDe, the chapter uses a similar definition as the tagged signal model by Lee and Sangiovanni-Vincentelli [30].

The ForSyDe modeling elements are introduced and discussed in Sects. 4.2.1 and 4.2.2 using the synchronous model of computation and the Haskell implementation of the ForSyDe modeling framework, in short *Haskell-ForSyDe*. Section 4.2.3

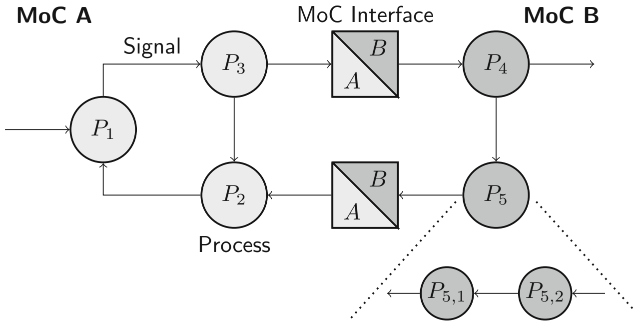

Fig. 4.2 A ForSyDe model is a hierarchical concurrent process network. Processes of different models of computation can communicate with each other via MoC interfaces. The process P_5 is created through process composition of the two processes $P_{5,1}$ and $P_{5,2}$

contains a deeper discussion of the synchronous MoC and also introduces additional ForSyDeMoCs. Heterogeneous ForSyDe models can be created by MoC interfaces, which are discussed in Sect. 4.2.4. Section 4.2.5 shows the usage of functions as signal values to model reconfigurable processes, and finally Sect. 4.2.6 concludes the discussion of the modeling framework with a larger modeling case study. All examples in this section have been modeled with the `forsyde-shallow` library, which is available on https://github.com/forsyde/forsyde-shallow, and have been run using version 7.10.3 of the Glasgow Haskell Compiler `ghc`.

It is important to point out that due to its pure functional paradigm, Haskell is a perfect match to the ForSyDe modeling framework. Still, the ForSyDe modeling formalism is language-independent and can be implemented in different languages. A good example is the SystemC implementation of ForSyDe, *SystemC-ForSyDe*, which is discussed in Sect. 4.5.

4.2.1 Signals

Processes communicate with each other by writing to and reading from signals. 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. In the ForSyDe modeling framework, a signal is modeled as a list of events, where the tag of the event is either implicitly given by the event's position in the list as in the ForSyDe synchronous MoC or can be explicitly specified as in the case of the continuous-time or discrete-time MoC. The interpretation of tags is defined by the MoC. An identical tag of two events in different signals does not necessarily imply that these events happen at the same time. All events in a signal must have values of the same type. Signals are written as $\{e_0, e_1, e_2, \ldots\}$, where $e_i = (t_i, v_i)$ denotes the tag t_i and the value v_i of the i-th event in the signal.

In general, signals can be finite or infinite sequences of events and S is the set of all signals. The type of a signal with values of type D is denoted $S(D)$.

In order to distinguish ForSyDe signals from normal lists in Haskell, there is a special data type `Signal a` for signals carrying values of data type `a`. A signal of data type `a` is modeled as

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

and

```
s1 = 1:-2:-3:-4:-NullS
```

models a signal s_1 with integer values and has the data type `Signal Int`. The `Signal` data type is isomorphic to Haskell's list data type. The Haskell version of ForSyDe outputs signals in a more readable form, i.e., s_1 will be presented as `{1,2,3,4}`. The function `signal` can be used to convert a Haskell list into a ForSyDe signal, so another signal s_2 can be created by

```
s2 = signal [10,20,30,40]
```

resulting in the signal `{10,20,30,40}`.

The signals described so far have been finite signals, but infinite signals can be modeled in Haskell as well due to Haskell's lazy evaluation mechanism. The function `constS` creates an infinite signal of constant values.

```
constS x = x :- constS x
```

A full evaluation of the signal `constS 5` would not terminate. However, finite parts of infinite signals can be evaluated due to Haskell's call-by-need evaluation mechanism. The function `takeS` can be used for this purpose and returns the first n values of a signal, e.g., `takeS 3 (constS 5)` evaluates to `{5,5,5}`.

4.2.2 Processes

Processes are defined as functions on signals

$$p : S^m \to S^n = \underbrace{S \times S \times \cdots \times S}_{m} \to (\underbrace{S \times S \times \cdots \times S}_{n}).$$

The set of all processes is P.

Processes are functions in the sense that for a given set of input signals, always the same set of output signals is returned. Thus $s = s' \Rightarrow p(s) = p(s')$ is valid for a process with one input signal and one output signal. Note, that this still allows processes to have an internal state. A process does not necessarily react identical to the same event applied at different times. But it will produce the same, possibly infinite, output signal when confronted with identical, possibly infinite, input signals provided it starts with the same initial state.

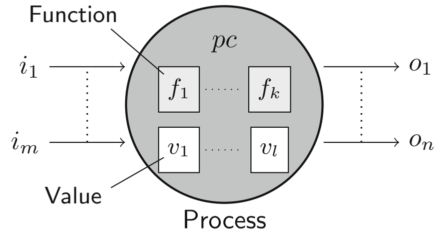

Fig. 4.3 A process is constructed by means of a process constructor *pc* that takes 0 to k side-effect-free functions and 0 to l values as argument

For processes with arbitrary number of input and output signals, the notation can become cumbersome to read. Hence, for the sake of simplicity, this chapter uses mostly processes with one input and one output only. This is not a lack of generality since it is straightforward to introduce *zip* and *unzip* processes which merge two input signals into one output signal and split one output signal into two output signals, respectively [27]. These processes together with appropriate process composition allow to express arbitrary behavior.

4.2.2.1 Process Constructors

Figure 4.3 illustrates the concept of *process constructor*, which is a key concept in ForSyDe originating from higher-order functions in functional programming languages. ForSyDe defines a set of well-defined process constructors, which are used to create processes. A process constructor *pc* takes zero or more side-effect-free functions $f_1, f_2, \ldots, f_k$ and zero or more values $v_1, v_2, \ldots, v_l$ as arguments and returns a process $p \in P$.

$$p = pc(f_1, f_2, \ldots, f_k, v_1, v_2, \ldots, v_l)$$

The functions represent the process behavior and have no notion of concurrency. They simply take arguments and produce results. The values model configuration parameters or the initial state of a process. The process constructor is responsible for establishing communication with other processes via signals. It defines the time representation, the communication interface, and the synchronization semantics. This separation of concerns leads to an elegant mathematical formalism that facilitates design analysis and design transformation. It is important to point out that most programming languages do not prevent the designer from creating functions that have side-effects, for instance by accessing a global variable inside a C-function. Since Haskell is a pure functional language, functions are side-effect free by design, but this property is not guaranteed in the SystemC version of ForSyDe (Sect. 4.5), where the designer has the responsibility not to use functions with side-effect.

A set of process constructors determines a particular MoC. The concept of process constructors ensures a systematic and clean separation of computation and communication. A function that defines the computation of a process can in principle be used to instantiate processes in different computational models. However, a computational model may impose constraints on functions. For instance,

the synchronous MoC requires a function to take exactly one event on each input and to produce exactly one event for each output (Sect. 4.2.3.1). Processes belonging to a data-flow MoC can consume and produce more than one token during each iteration, which has to be reflected in the computation functions for data-flow processes (Sect. 4.2.3.2).

The synchronous MoC is used to illustrate the usage of basic process constructors, which can be divided into combinational and sequential process constructors. A corresponding set of process constructors exists in the ForSyDe libraries for the other models of computation. Due to the total order of events in the synchronous MoC, the tag of an event is implicitly given by its position in the signal, so that synchronous ForSyDe signals do not carry an explicit tag.

A *combinational process constructor* creates combinational processes, i.e., processes that have no internal state. The basic combinational process constructor in the synchronous MoC is *mapSY*, which applies a function f to all signal values. Thus a process `twice` that doubles all input values of a synchronous signal `s` is modeled as

```
twice s = mapSY (*2) s
```

and can simulate the process *twice* with the input signal $s1$ as `twice s1`, which yields `{2,4,6,8}`. An adder can be modeled by

```
adder s1 s2 = zipWithSY (+) s1 s2
```

The process constructor *zipWithSY* applies a function f pairwise onto two synchronous signals. Hence, `adder s1 s2` yields `{11,22,33,44}` as output signal. The naming `mapSY`, `zipWithSY`, `zipWith3SY`, ... originates from functional programming, but to simplify for industrial designers the following aliases have been defined in ForSyDe for combinational process constructors: `combSY`, `comb2SY`, `comb3SY`, ...

A sequential process is a stateful process, where an output value depends not only on the current input values but also on the current state. The basic *sequential process constructor* is *delaySY*, which creates a process that delays a synchronous signal by one event cycle and where the current output value is given by the current state of the process. A register process can be modeled by

```
register s = delaySY 0 s
```

Here, the first argument to *delaySY*, in this case 0, is the initial state of the sequential process *delaySY* 0. Then `register s1` creates the output signal `{0,1,2,3,4}`, where 0, the initial state, is the value of the initial event. More powerful sequential processes and process constructors for finite state machines can be created by process composition as explained in the following section.

4.2.2.2 Process Composition

New processes can be created by composition of other processes to form a hierarchical process network. Figure 4.4 shows a model of a process *counter* that

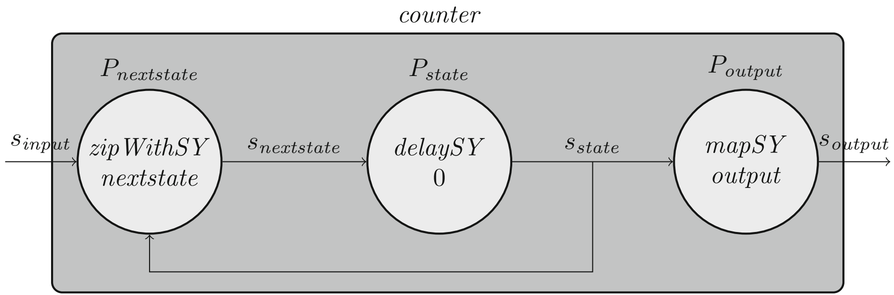

Fig. 4.4 The counter is modeled as concurrent process network. The process network is expressed as set of equations

is modeled with three processes: the combinational process $P_{nextstate}$ that calculates the next state, the sequential process P_{state} that holds the current state, and the combinational process P_{output} that models the output decoder. The counter shall output the value `TICK`, when the state is `0`, otherwise the event is absent. The possibility of the absence of an event is a special property of the synchronous MoC. It is modeled in ForSyDe by means of a special data type

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

expressing that an event can either be present, and then has a value, or absent.

Figure 4.4 illustrates how more complex stateful sequential processes can be built by process composition around a basic sequential delay process that exists in all ForSyDe MoCs. It is worth to mention, that sequential processes in ForSyDe have a *local state*. However, due to its foundation in form of the theory of models of computation [30], where processes can only share information via signals, there is nothing like a shared global state in form of a shared variable. Since ForSyDe processes require side-effect-free functions as arguments, ForSyDe processes are deterministic in the sense that they will give the same output signals for the same history of input signals.

Zero-delay feedback loops in the synchronous MoC and related MoCs can cause causality problems, where a system specification might have no solution, a unique solution, or several solutions. ForSyDe deals with zero-delay feedback loops in a pragmatic way by simply forbidding it, following the same approach as the synchronous programming language Lustre [23]. More sophisticated solutions either calculate the least fix-point, as adopted by the synchronous languages Esterel [7] or Quartz [52], or using a relational approach, as adopted by the synchronous language Signal [12]. Another practical alternative is to introduce another level in the tag system, as adopted in VHDL, which includes micro steps in form of a delta-delay.

Listing 1 System model of counter in Haskell-ForSyDe

```
1  module Counter where
2
3  import ForSyDe.Shallow
4
5  data Direction = UP
6                 | HOLD deriving (Show)
7
8  data Clock = TICK deriving (Show)
9
10 -- Step 1: Specification of process network
11 counter s_input = s_output
12    where s_output = p_output s_state
13          s_state = p_state s_nextstate
14          s_nextstate = p_nextstate s_state s_input
15
16 -- Step 2: Selection of process constructors
17 p_nextstate = zipWithSY nextstate
18 p_state = delaySY 0
19 p_output = mapSY output
20
21 -- Step 3: Specification of leaf functions
22 nextstate state HOLD = state
23 nextstate 4     UP   = 0
24 nextstate state UP   = state + 1
25
26 output 0 = Prst TICK
27 output _ = Abst
```

A top-down design of a system model in ForSyDe is conducted in three steps:

1. The designer sketches the process network including the selection of the MoC and the communication between the processes.
2. The designer selects suitable process constructors for all processes in the process network, alternatively expresses a high-level process by a composition of other processes. In Fig. 4.4, the process constructors *zipWithSY* and *mapSY* are used to form combinational processes, while the process constructor *delaySY* is used to model a process with internal state.
3. The designer formulates the arguments to the process constructors, i.e., the leaf functions (*nextstate*, *output*) and other parameters (initial state for *delaySY* is 0), to form ForSyDe processes.

Listing 1 shows the full ForSyDe model of the counter in Haskell-ForSyDe, including data types for the input (`Direction`) and output signals (`Clock`).

Haskell is a strongly typed language and although no data types are given, Haskell can infer the data type of the process `counter` to

```
counter :: Signal Direction -> Signal (AbstExt Clock)
```

This means that `counter` is a function that takes an input signal with data of type `Direction` and produces a signal with possibly absent events of type `Clock`. A simulation of the counter in ForSyDe shows the absent events as `'_'`-characters.

```
*Counter> counter (signal ([HOLD,UP,HOLD,UP,UP,UP,UP,HOLD,UP]))
{TICK,TICK,_,_,_,_,_,TICK,TICK,_}
```

4.2.3 ForSyDe Models of Computation

The Haskell-ForSyDe library supports several MoCs. The following subsections introduce the synchronous MoC (Sect. 4.2.3.1), two data-flow MoCs (Sect. 4.2.3.2), and finally the continuous-time MoC (Sect. 4.2.3.3). These three MoCs form a good base for the challenging design of Cyber-Physical Systems (CPSs), which integrate computation with physical processes [14]. The computation system consisting of software and digital hardware is naturally modeled with data-flow MoCs and synchronous MoC, while the physical process or plant is usually modeled with a continuous-time MoC. Section 4.2.6 presents a case study that integrates the presented MoCs of this section.

4.2.3.1 Synchronous Model of Computation (MoC)

The family of synchronous languages [5, 6], consisting of languages like Esterel [7], Lustre [23], Signal [12], or Quartz [52], is based on the synchronous MoC and uses the *perfect synchrony assumption*, i.e., *neither computation nor communication takes time*. Timing is entirely determined by the arriving of input events because the system processes input samples in zero time and then waits until the next input arrives. If the implementation of the system is fast enough to process all input before the next sample arrives, it will behave exactly as the specification model. ▸ Chapter 2, "Quartz: A Synchronous Language for Model-Based Design of Reactive Embedded Systems" contains a more detailed discussion about synchronous languages in general and the synchronous language Quartz in particular.

Synchronous processes are defined by the following specific characteristic. All synchronous processes consume and produce exactly one event on each input or output in each evaluation cycle, which implies a *total order* of all events in any signal inside a synchronous MoC. Events with the same tag appear at the same time instance. The set of synchronous processes is $P_{SY} \subset P$.

To model asynchronous or sporadic events like a reset signal, ForSyDe uses the special value $\perp$ to model the *absence* of an event. A value set V that is extended with the absent value $\perp$ is denoted $V_\perp = V \cup \{\perp\}$. It is often practical to abstract a non-absent value with the value $\top$. For convenience we call an event with an absent value an *absent event* and an event with a non-absent value a *present event*.

Figure 4.5 gives a formal definition of the set of basic process constructors and processes, which are needed to model a system in the synchronous MoC. In other models of computations, the set of basic process constructors is similar. Process constructors in the synchronous domain have the suffix "*SY*." Together with process composition, this set of combinational process constructors is sufficient to model systems inside the synchronous MoC.

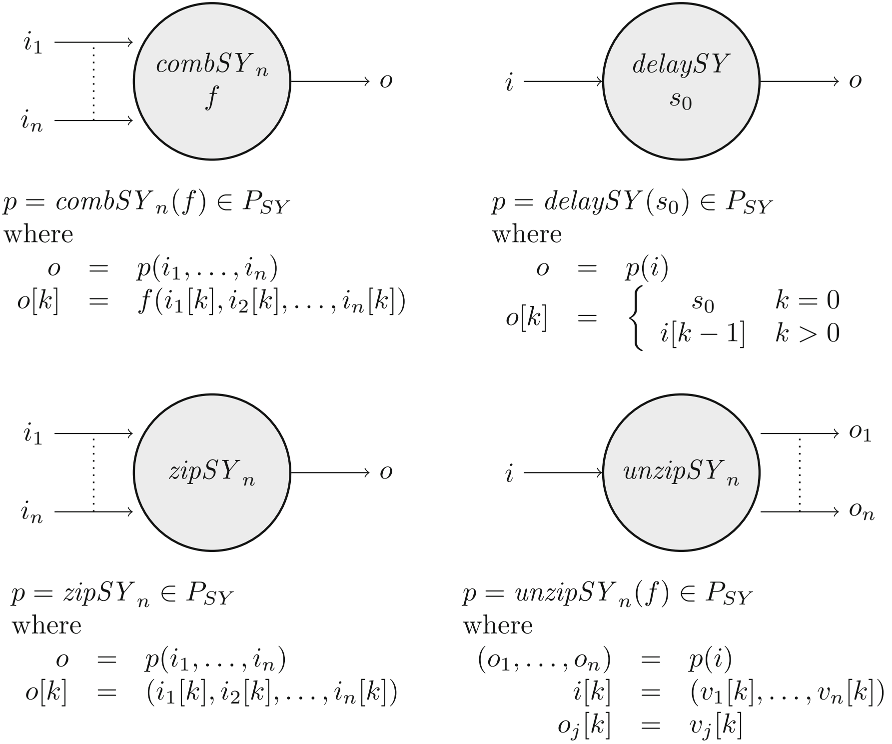

Fig. 4.5 Formal definition of the basic process constructors $combSY_n$, *delaySY*, and the basic processes *zipSY* and *unzipSY*

A combinational process constructor $combSY_n$ takes a function $f : D_1 \times \cdots \times D_n \rightarrow E$ as argument and returns a process $p : S(D_1) \times \cdots \times S(D_n) \rightarrow S(E)$ with no internal state. The delay process constructor *delaySY* takes only one value $s_0 : D$ as argument and produces a process $p : S(D) \rightarrow S(D)$ that delays the input signal one cycle. The supplied value is the initial value of the output signal. The basic processes $zipSY_n$ and $unzipSY_n$ are required because a ForSyDe process is a mathematical function, which can only have a single signal as output. However, it is possible to model a process that has a signal of tuples as output and convert it with the process $unzipSY_n$ into a tuple of n signals. The process $zipSY_n$ converts a tuple of signals into a signal of tuples. Other process constructors are defined for convenience, such as the state machine constructor $mooreSY_n$, which is used to model a finite state machine, where the output depends only on the current state.

Figure 4.6 illustrates the process constructor *mooreSY*, which takes two functions, *ns* and o, and a value s_0 as arguments. The function *ns* calculates the next state, the function o calculates the output, and the value s_0 gives the initial state. Thus instead of specifying the counter of Fig. 4.4 with an explicit process network,

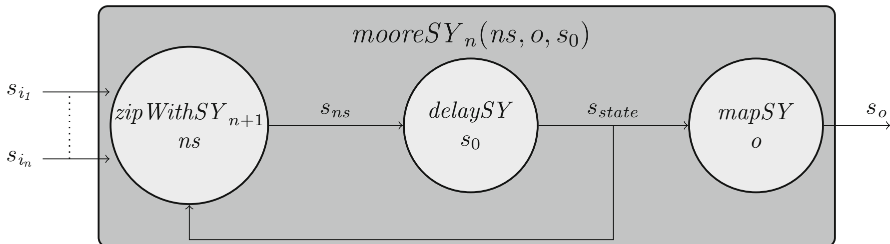

Fig. 4.6 The process constructor $mooreSY_n$ creates a synchronous process that models a state machine, where the output depends only on the state

the designer can use the process constructor *mooreSY* together with the arguments for *ns*, *o*, and s_0 to model the counter, i.e.,

```
counter = mooreSY nextstate output 0
```

This model has exactly the same behavior as the counter of Listing 1. The ForSyDe library also includes the *mealySY* process constructor to model synchronous Mealy FSMs, where the output depends not only on the state but on the input signal as well.

4.2.3.2 Data-Flow Models of Computation

ForSyDe provides several libraries for data-flow models of computation. These MoCs are untimed and there is no total order between events in two different signals, only a partial order exists. Many data-flow models exist in the literature ranging from models providing a high degree of expressiveness at the cost of low analyzability, like dynamic data flow, to models providing high analyzability at the cost of limited expressiveness, like synchronous data flow. ▶ Chapter 3, "SysteMoC: A Data-Flow Programming Language for Codesign" gives a very detailed overview about the most common data-flow Model of Computation (MoC) and introduces SysteMoC, a language to model and design data-flow systems based on SystemC.

This section will introduce two of ForSyDe's untimed data-flow MoCs: the *untimed MoC* provides a high level of expressiveness, while the *synchronous data flow (SDF) MoC* provides a high level of analyzability.

The ForSyDe SDF MoC follows the definition of synchronous data flow [29]. An SDF actor is created by process constructors, which take consumption rates, production rates, and a function as arguments and produce a process (actor) as result. Internally, an actor process $actorSDF_{m,n}$ with *m* input and *n* output signals is created by a composition of a $zipWithSDF_m$ process constructor and an $unzipSDF_n$ process.

Figure 4.7 illustrates the usage of the SDF MoC library by creating a system that repetitively takes two tokens from an input signal and compares them with the current maximum. The system model consists of two processes, which are created using an $actorSDF_{2,2}$ and a *delaySDF* process constructor. The process constructor $actorSDF_{2,2}$ takes three arguments. The first one, (2,1), gives the consumption rate for the input signals. The second one, (2,1), gives the production rate for the output

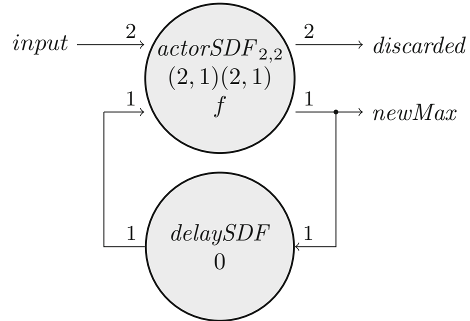

Fig. 4.7 Process network of an SDF model that calculates the current maximum and discards the other values

signals, and the third one f gives the computation function that operates on the input tokens. The initial value for the maximum is set to 0, given by the argument to the *delaySDF* process constructor. In each iteration the current maximum is compared with two new input values to determine a new maximum. The new maximum is fed back to the system, while the two other values are discarded.

Listing 2 An SDF model that calculates the current maximum and discards the other values

```
1 system input = (discarded, newMax)
2   where (discarded, newMax)
3                 = actor22SDF (2,1) (2,1) f input curMax
4         curMax = delaySDF 0 newMax
5
6 f [a, b] [c] = [(delete newMax [a,b,c], [newMax])]
7   where newMax = maximum [a,b,c]
```

The corresponding ForSyDe code is given in Listing 2. The function arguments in the SDF-MoC operate on lists and return lists as output. This can be seen in Line 6 of Listing 2, where the function f takes lists of different size as input and returns another list with tuples of lists as output values. The standard Haskell function `delete` removes an element from a list and outputs the list in reversed order. A simulation of the process network returns the expected result in the form of a tuple of signals. The first signal consists of the discarded values, with a changed order due to the reversed output of the function `delete`, while the second signal consists of the current maximum values.

```
*SDF> system (signal [1..10])
({1,0,3,2,5,4,7,6,9,8},{2,4,6,8,10})
```

The usage of the SDF MoC library ensures a well-defined and analyzable SDF model, where all processes behave according to the rules of the SDF-MoC.

In contrast to the SDF MoC, the untimed MoC of ForSyDe gives a high grade of expressiveness at the cost of losing analyzability. The reason is that

processes in the untimed MoC base the decision on how many tokens to be consumed and produced during an iteration on the current state of the process. This enables to model processes that vary the consumption and production rates during their run time. The expressiveness of the untimed MoC is best illustrated using the process constructor *mealyU*, which creates a state machine of Mealy type. The first argument is a function γ that operates on the state of the process and returns the number of tokens to be consumed in the next iteration. Listing 3 illustrates the use of the *mealyU* process constructor and the γ-function by a tutorial example.

Listing 3 An untimed model based on the process constructor *mealyU* consuming a varying number of input tokens in each iteration

```
1 system = mealyU gamma nextstate output 0
2     where gamma state = state + 1
3           nextstate state xs = length xs
4           output state xs = [state]
```

The initial state of the system is 0 given by the last argument of the *mealyU* process constructor (Line 1). Thus due to the γ-function (Line 2), a single token will be consumed in the first iteration. The next state is determined by the function *nextstate* (Line 3), which is the number of the consumed tokens during an iteration, i.e., one token in the first iteration. Thus the result of the γ-function will be 2 in the second iteration and consequently an increasing number of tokens is consumed in following iterations. The function *output* outputs the current state of the process (Line 4). The simulation below shows the expected results and stops when there are not enough tokens in the input signal.

```
*Untimed> system (signal [1..100])
{0,1,2,3,4,5,6,7,8,9,10,11,12}
```

4.2.3.3 Continuous Time Model of Computation

The time base, i.e., the tag, for the continuous-time MoC is given by the set of the positive real numbers, $t \in \mathbb{R}_{+}$, allowing to model physical time. To model continuous-time systems, ForSyDe exploits one key property of functional programming languages: functions are first-class citizens and can be treated as normal values. A continuous-time signal is defined as a set of sub-signals, where each sub-signal is defined by its time interval and the function that is executed during this time interval. A signal s_1 that has the constant value 1 during the time interval between 0 and 0.4 and the constant value -0.5 during the time interval between 0.4 and 1.0 is modeled as

```
s1 = signal [SubsigCT ((\t -> 1.0), (0,0.4)),
             SubsigCT ((\t -> -0.5), (0.4,1.0))]
```

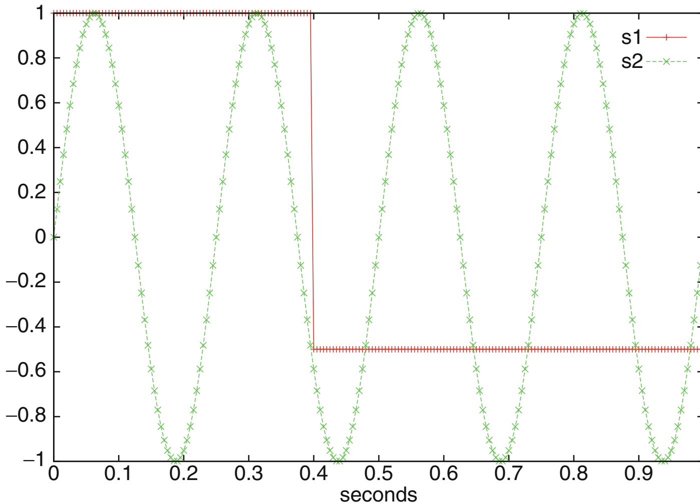

Fig. 4.8 The continuous-time signals s_1 and s_2 plotted (Resolution: 5 ms)

where `t-> 1.0` and `t-> -0.5` are the functions yielding a constant 1 or a constant 0.5, respectively. In a similar way, a continuous-time signal for a sine wave can be constructed, and the ForSyDe library allows to model sine waves, with the function `sineWave`, which takes the frequency in Hz of the sine wave as argument. The signal s_2 models a sine wave with the frequency 4 Hz during the time interval between 0 and 1.0.

```
s2 = sineWave 4 (0,1.0)
```

The signals can be plotted using the ForSyDe command `plotCT'` with the desired resolution as illustrated in Fig. 4.8. To generate the plots, `plotCT'` requires an installation of `gnuplot`.

```
plotCT' 5e-3 [(s1, "s1"), (s2, "s2")]
```

Please note that due to the lazy evaluation of Haskell, ForSyDe only calculates the results of the functions when needed, for instance to plot the graph with the given resolution. Otherwise, functions are treated as normal values.

The process constructors in the continuous-time MoC correspond to the process constructors in the synchronous, i.e., *mapCT*, *zipWithCT*, or *delayCT*. Processes are created and composed in the same way as in the synchronous MoC. The process p_1 that adds two continuous-time signals and the process p_2 that multiplies two signals are modeled as follows:

```
p1 = zipWithCT (+)
p2 = zipWithCT (*)
```

Figure 4.9 shows the plot of the operations `p1 s1 s2` and `p2 s1 s2`.

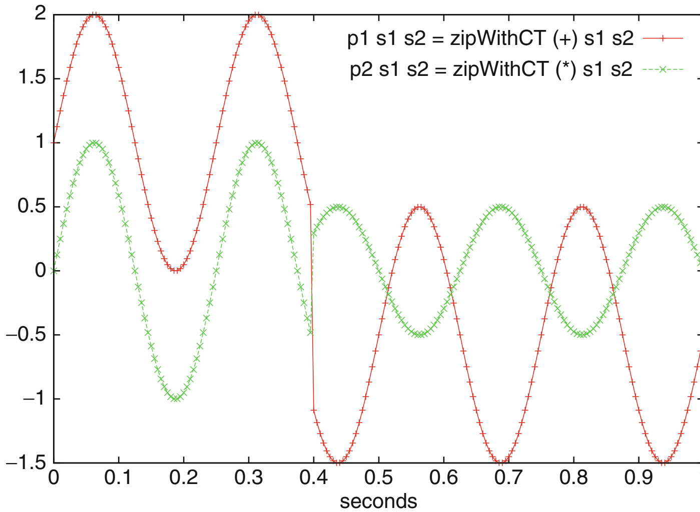

Fig. 4.9 Operations on the continuous-time signals s_1 and s_2 (Resolution: 5 ms)

4.2.4 Model of Computation Interfaces

ForSyDe supports heterogeneity through MoC interfaces, which are special types of processes used to connect signals belonging to different MoCs. Classical examples for practical MoC interfaces are analog-to-digital and digital-to-analog converters. Corresponding MoC interfaces also exist in ForSyDe for the connection of the continuous-time MoC and the synchronous MoC, in the form of `ct2sy` and `sy2ct`. Both interfaces are inspired by practical A/D and D/A converters.

The MoC interface `ct2sy` corresponds to an A/D converter and is an ideal process in the sense that is does not perform quantization of the input signal; it only samples the input signals according to the given sample period. To model a real A/D converter, an additional synchronous ForSyDe process `quantizer` is required that takes the minimal and maximal signal values and the number of bits as input and produces a quantized signal. There are two modes for the MoC interface `sy2ct`, which corresponds to a D/A converter. In `DAhold`-mode, the continuous-time output follows directly the synchronous input value for the whole sampling period, while in `DAlinear`-mode, a smooth transition between two adjacent synchronous values is done.

Listing 4 and the plot of the output signals in Fig. 4.10 illustrate the use of the MoC interfaces between the continuous-time MoC and synchronous MoC.

The example shows the effects of both an ideal A/D converter `adc_ideal` on the output signal s_5 and a nonideal A/D converter `adc_non_ideal` with a quantization stage by means of the output signal s_6. ForSyDe provides a few standard MoC interfaces but allows designers to write their own MoC interfaces. These interfaces can be on all abstraction levels and might be ideal, as in the case

Listing 4 Illustration of the use of MoC interfaces by means of the `ct2sy` and `sy2ct`, which are used to model ideal and nonideal A/D converters and a D/A converter. The output signals are plotted in Fig. 4.10

```
1  -- Ideal A/D-converter
2  adc_ideal = ct2sy 0.02
3  -- Non-ideal A/D-converter with quantizer
4  quantizer = mapSY (quantize (-1.0, 1.0) 4)
5  adc_non_ideal = quantizer . (ct2sy 0.02)
6  -- D/A-converter using DAhold-mode
7  dac = sy2ct DAhold 0.02
8
9  -- Sine wave as input signals
10 s4 = sineWave 1 (0,1.0)
11
12 -- Output signals : s6 is inverted for illustration purposes
13 s5 = (dac . adc_ideal) s4
14 negator = mapSY (* (-1.0))
15 s6 = (dac . negator . adc_non_ideal) s4
```

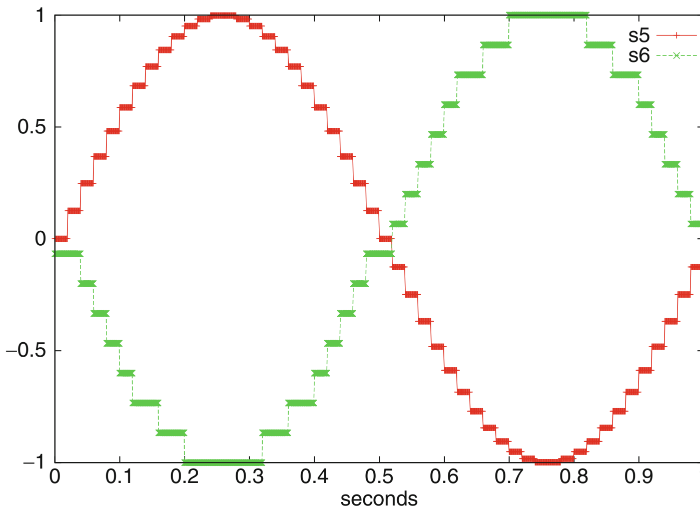

Fig. 4.10 The signal s_5 shows the effect of the sampling period on the input sine wave, but does not use any bit-level quantization. The signal s_6 shows also the effect of quantization with a resolution of 4 bits on a negated signal

for the standard `ct2sy` MoC interface, or can be nonideal as in the case of the `adc_non_ideal`, which in itself is a MoC interface created by the composition of `ct2sy` and the synchronous quantizer.

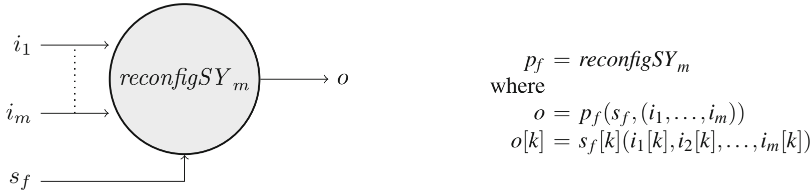

Fig. 4.11 The process $reconfigSY_m$ models a reconfigurable process, where the function that the process executes is controlled by an input signal carrying functions

4.2.5 Reconfigurable Processes

The property of functional languages, that functions are regarded as first-class citizens, has already been used for the continuous-time MoC library. ForSyDe exploits this key property also to model *reconfigurable processes*, i.e., processes that change their behavior over time, by introducing signals carrying functions as event values. An example is the synchronous signal s_f, which has functions on numbers as signal values.

$$s_f = \{(+), (-), (*), (+)\}$$

Figure 4.11 illustrates how a signal $s_f : S(D_1 \times \cdots \times D_m \rightarrow E)$, where the values of the signal are functions, serves as input signal for a reconfigurable process. The reconfigurable process $p_f : S(D_1) \times \cdots \times S(D_m) \times S(D_1 \times \cdots \times D_m \rightarrow E) \rightarrow S(E)$ executes always the current value, i.e., a function, of the signal s_f. This means that the reconfigurable process does not need to provide the code for different modes of functions, because they are supplied from the outside. Reconfigurable processes can be implemented by run-time reconfigurable hardware or software, where the new functions can be loaded into a reconfigurable area, such as an FPGA or memory block, during operation.

Using the classification introduced by McKinley in [36], reconfigurable ForSyDe processes belong to the category of *compositional adaptation*. In contrast, most modeling frameworks offer only *parameter adaptation*, where adaptivity is changed by parameter settings or the existence of different system modes usually implemented by *if-then-else* statements.

Reconfigurable processes can be used to create a *self-adaptive process*, as illustrated in Fig. 4.12, where the executed function of the process is triggered by the change of the values of the input or output signals. The self-adaptive process p_{sa} is constructed as a process network consisting of a reconfigurable $p_{reconfig}$ and another process $p_{control}$ that controls the functionality of the reconfigurable process $p_{reconfig}$. At the highest level of abstraction, we assume adaptation to be instantaneous. Thus the change of functionality indicated by a new value of the signal s occurs at the same instant as the input or output values that trigger the change of the functionality of the adaptive process.

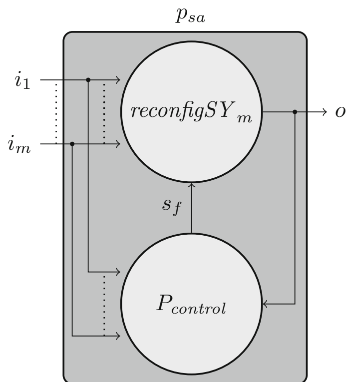

Fig. 4.12 A self-adaptive process p_{sa} is modeled as a process network of an adaptive process and an additional process. The signal s_f carries functions as values as illustrated in Fig. 4.11

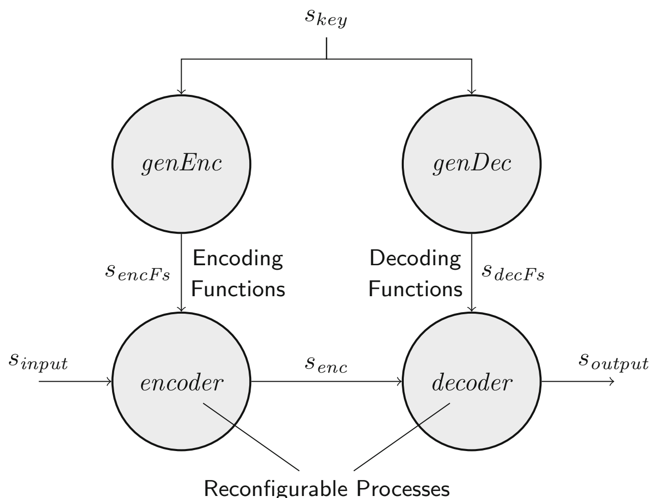

Fig. 4.13 Encoder-decoder

In order to show that reconfigurability can be treated as a first-class citizen in ForSyDe, we illustrate how the existing synchronous ForSyDe process constructor $combSY_n$ in conjunction with the function application operator (\$) can be used to model a synchronous reconfigurable process $reconfigSY_n$. The function application operator is defined as `f $ x = f x` and enables the application of functions on signal values.

```
reconfigSY = combSY ($)
reconfig2SY = comb3SY ($)
```

Figure 4.13 shows a tutorial example to model run-time reconfigurability using a synchronous system model with two reconfigurable processes. A signal is

encoded with an encoding function and later the encoded signal is decoded with a decoding function. The signal s_{key} is an input to both the *genEnc* and *genDec* processes. The processes *encoder* and *decoder* are reconfigurable processes and have signals carrying functions as inputs. Figure 4.13 models reconfigurability at a high abstraction level, where the reconfiguration of the reconfigurable process is assumed to be instantaneous and does not consume any time. The corresponding ForSyDe source code is given in Listing 5.

Listing 5 ForSyDe source code for the encoder-decoder example

```
1  module EncoderDecoder where
2
3  import ForSyDe.Shallow
4
5  reconfigSY fs xs = zipWithSY ($) fs xs
6
7  genEnc s_key = mapSY f s_key
8                where f x y = y + x
9
10 genDec s_key = mapSY f s_key
11               where f x y = y - x
12
13 encoder s_encFs xs = reconfigSY s_encFs xs
14
15 decoder s_decFs xs = reconfigSY s_decFs xs
16
17 system s_key s_input  = (s_enc, s_output)
18        where s_output = decoder s_decFs s_enc
19              s_enc    = encoder s_encFs s_input
20              s_encFs  = genEnc s_key
21              s_decFs  = genDec s_key\vspace*{5pt}
```

The simulation with the signal `s_key = signal [1,4,6,1,1]`, which carries the encoding keys, and an input signal `s_input = signal [1,2,3,4,5]` yields the expected output

```
*EncoderDecoder> system s_key s_input
({2,6,9,5,6},{1,2,3,4,5})
```

where the output is a tuple of two signals. The first signal `{2,6,9,5,6}` is the encoded signal s_{enc} and the second signal `{1,2,3,4,5}` is the decoded output signal s_{output}.

The processes *encoder* and *decoder* in Fig. 4.13 can be further refined in consecutive design steps to take reconfiguration time and the need for buffers into account. This would reflect the nature of partial and run-time reconfigurable FPGAs. For a more detailed discussion about the modeling and refinement of reconfigurable and adaptive systems in ForSyDe, see [50].

4.2.6 Modeling Case Study

To illustrate the capability of the ForSyDe modeling approach, a case study from the European FP6 ANDRES project [25] will be used. The case study models an Amplitude Shift Key (ASK) transceiver and combines three different MoCs: the continuous-time MoC, the synchronous MoC, and the untimed MoC.

The structure of the system is illustrated in Fig. 4.14. A synchronous signal of integers enters the system in (1) and is converted to a signal of bit vectors, which are then encoded. The encoded signal is converted into a serial bit stream, before it is converted into a continuous-time signal in (2). This signal is then modulated using an amplitude shift key modulation and amplified before it leaves the sender of the transceiver in (3). In order to test the system, a Gaussian noise (4) is added to the signal resulting in a noisy signal in (5). This signal is received by the transceiver and is converted from the continuous-time MoC to the untimed MoC. Then the serialized signal is converted into a signal of bit vectors (6), before it is decoded and converted to an output signal of integers (7).

The system has an inbuilt mechanism to deal with noise during transmission. In case bit errors are detected after (6), the adaptive power controller (8) increases the gain and the amplification of the transmitted signal will be increased in the process *adaptGain* (9).

The operation of the system is visible in the simulation in Fig. 4.15. At the start of the simulation, all input signals are either fully available to the simulator as in the case of the synchronous input signal (1) or are defined as source processes, like in the case of the Gaussian noise generator (4), which can produce an infinite signal thanks to Haskell's lazy evaluation mechanism. The simulation is data-driven and ends when the final event in the synchronous input signal (1) has been processed. The model contains several MoC interfaces, which define the relation between the tag systems of the different MoCs as discussed in Sect. 4.2.4. The synchronous input signal (1) is represented as a signal of integers. The Gaussian noise increases between 2 and 3 ms (4). This causes a bit error in the third event of the output signal (7). The error is detected and causes to amplify the encoded signal to be transmitted in the following event cycle from 3 to 4 ms (3). The following event is received correctly, which can also be seen from the noisy ASK signal in (5) and the amplification power is lowered again to normal level.

4.3 Transformational Design Refinement

The ForSyDe design process starts with the development of an initial abstract *specification model* that defines the behavior of the system as a function between system inputs and system outputs based on the tagged signal model [30]. A central idea of ForSyDe is to exploit the formal nature of this functional system model for design transformation and to refine the *specification model* by the application of well-defined design transformations into a lower-level *implementation model*, which

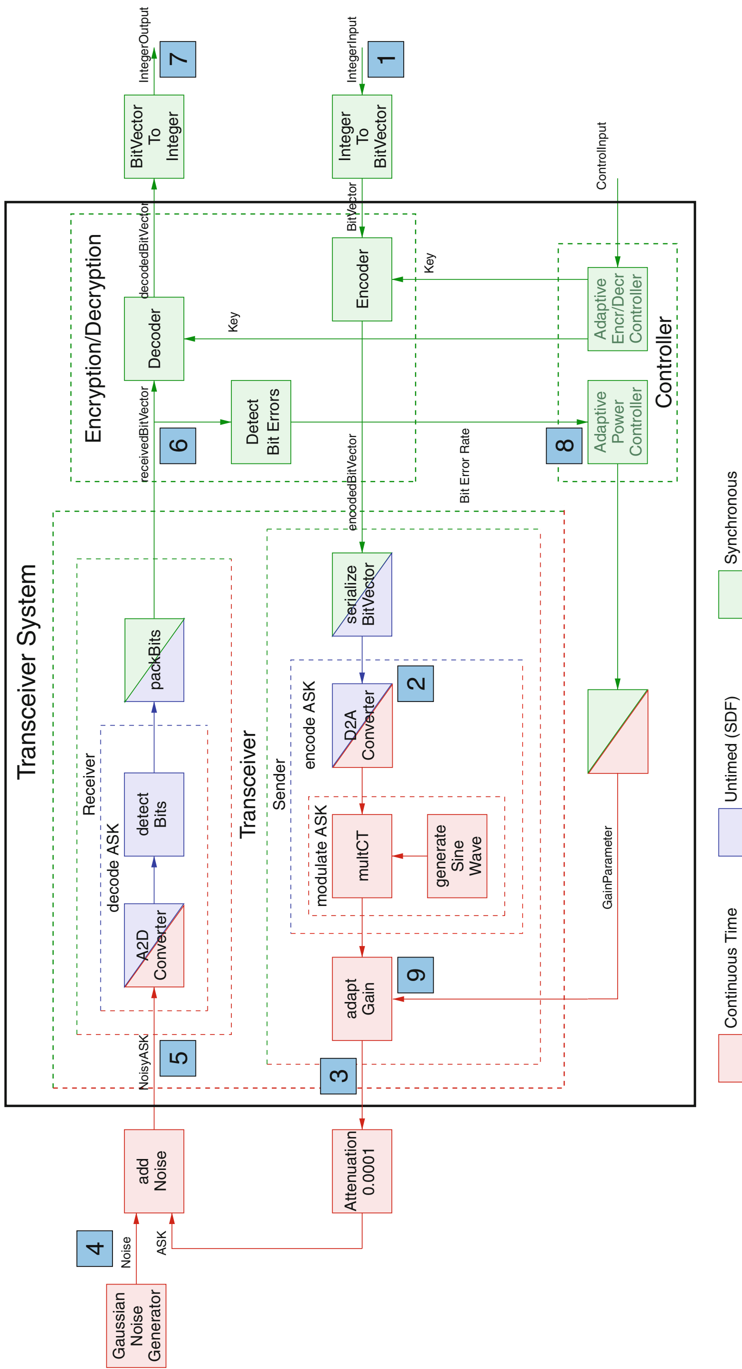

Fig. 4.14 Structure of the ASK transceiver example

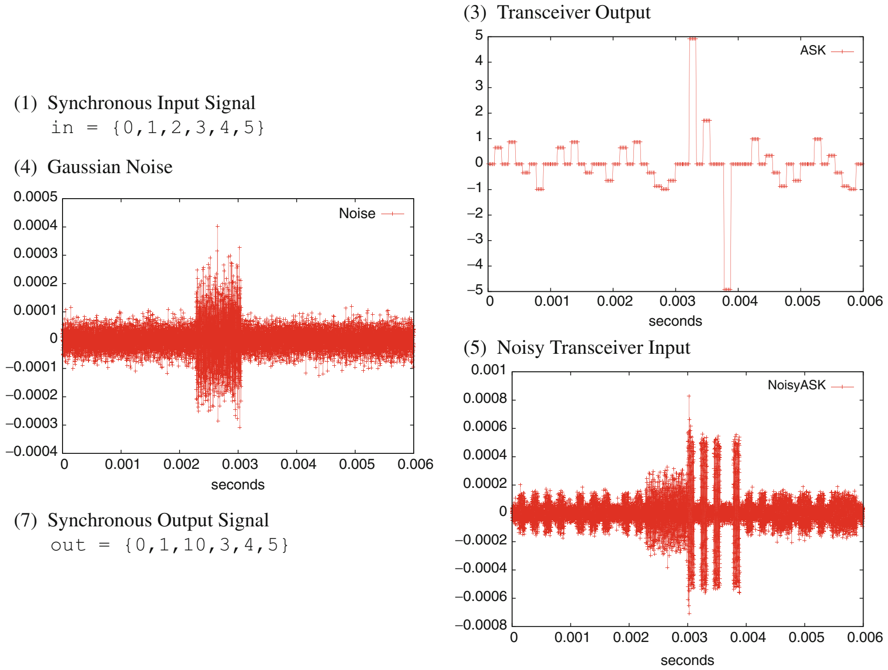

Fig. 4.15 Simulation of the ASK transceiver case study

is then synthesized into the target platform (Sect. 4.4). Since both the specification and the implementation model are based on the same ForSyDe semantics, as described in Sect. 4.2, they can be simulated using the same testbench as long as the specification of the inputs and outputs do not change during the refinement process.

The transformational design refinement process requires both *semantic-preserving* transformations, which do not change the meaning, i.e., the timely and functional behavior, of the model, and *nonsemantic-preserving* transformations or *design decisions*, which change the meaning of the model. Nonsemantic-preserving design decisions are required to improve the efficiency of the model, for instance, to refine an infinite buffer into a finite buffer or for data type refinement, but require an additional verification effort.

In order to give the designer information about the implications on the behavior of a refined process due to the application of a design transformation rule, ForSyDe has introduced the concept of *characteristic function* and exemplified it for the synchronous MoC [49]. The characteristic function F_{PN} of a process network *PN*, where

$$PN(i_1, \ldots, i_m) = (o_1, \ldots, o_n)$$

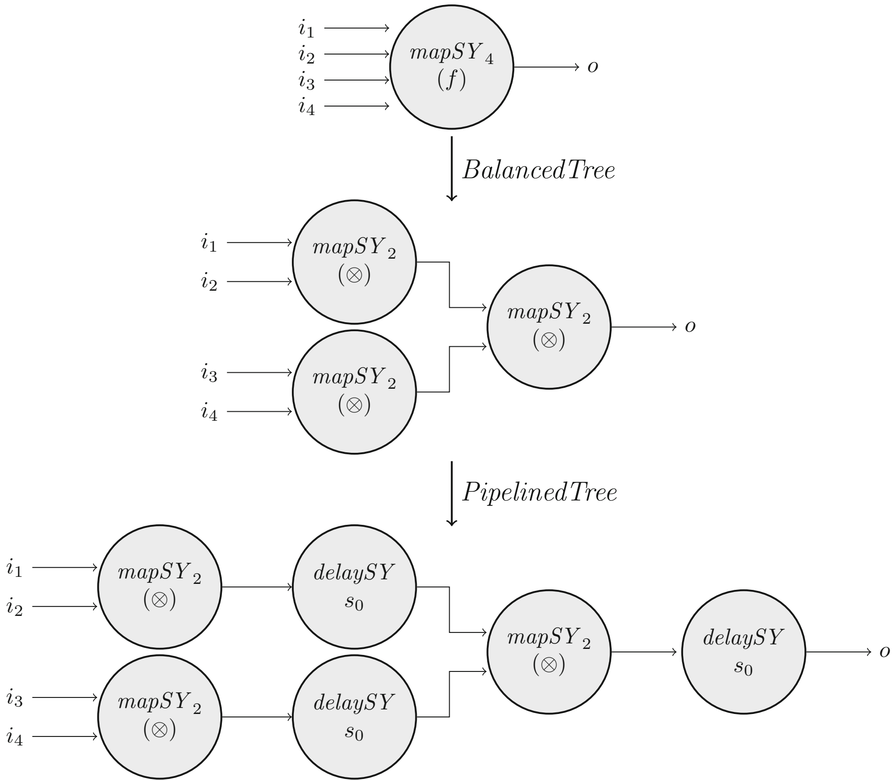

Fig. 4.16 Transformation of a combinational function into a balanced, pipelined tree structure ($m = 4$)

expresses the functional behavior of a process network as the dependence of the output events at tag j on the input signals, i.e.,

$$F_{PN}(i_1, \ldots, i_m, j) = ((T(o_1[j]), V(o_1[j]), \ldots, (T(o_n[j]), V(o_n[j])))$$

where $T(e)$ denotes the tag of the event e, $V(e)$ denotes the value of the event e, and $s[j]$ denotes the j-th event of a signal s. Thus $T(o_i[j])$ and $V(o_i[j])$ give the tag and the value of the j-th event $o_i[j]$ of the output signal o_i. The characteristic function utilizes the property of the tagged signal model that divides an event into tag and value. This enables to compare the behavior of two different process networks with respect to both timing (tag) and computation function (value).

Figure 4.16 illustrates transformational design refinement using the semantic-preserving transformation rule *BalancedTree* and the design decision *PipelinedTree* in order to convert a process network *PN* performing the mathematical function $f(x_1, x_2, \ldots, x_m) = x_1 \otimes x_2 \otimes \cdots \otimes x_m$, where $m = 2^k; k \in \mathbb{N}_+$, into a pipelined

representation PN' that requires two input functions. The characteristic function informs the designer about the implication of the design transformation, which in this case is a delay of the output compared to the original process network PN by k event cycles, i.e.,

$$F_{PN'}(i_1, \ldots, i_m, j) = F_{delaySY_k(s_0) \circ PN}(i_1, \ldots, i_m, j); \quad \forall j \geq k$$

In other words, after this transformation, the refined process network will yield the same output values as the original process network after a delay of k event cycles. Thus during transformational design refinement, the characteristic function is used to inform the designer on both changes in the timely behavior and changes of the output values for a given design transformation rule.

The implication can be used by the designer to verify the local consequences of the design transformation. However, a transformation like *PipelinedTree* also affects the global timing of a larger process network and has to be compensated in feedback loops or parallel paths in the global process network. This section can only give an overview about transformational design in ForSyDe. For a more detailed discussion, see [49] as main reference but also [45], which focuses mainly on *nonsemantic-preserving transformations* and discusses the verification of design decisions at the local level and gives a method for restoring time correctness at the global level. So far only the concepts for design transformations have been proposed [45, 49], the automation of the design transformation is still a topic for future work.

4.4 Synthesis of ForSyDe Models

Thanks to the well-defined structure of ForSyDe models, it is possible to give a general scheme for the synthesis of a ForSyDe implementation model into an implementation on a given target platform. The general synthesis flow is described in Sect. 4.4.1 and then exemplified by hardware synthesis of ForSyDe models belonging to the synchronous MoC in Sect. 4.4.2. The general synthesis concepts have also been applied for software synthesis toward a single processor as part of a case study on Hardware/Software Codesign (HSCD) [32]. Furthermore, there exists a first version of a synthesis tool `f2cc` targeting GPGPUs from abstract ForSyDe specifications where the functions are expressed in C-code [11].

Please note that this section deals with the translation of ForSyDe models into the target implementation. The refinement of the specification model through design transformations into the implementation model has been the topic of Sect. 4.3.

4.4.1 General ForSyDe Synthesis Concepts

ForSyDe models are structured as hierarchical concurrent process networks, where each process is either (1) composed of other processes communicating via signals, is (2) constructed by means of a process constructor, or is (3) a basic process. In order

to synthesize a system model into a target implementation, synthesis rules have to be developed for these different cases.

1. **Synthesis of concurrent process networks.** ForSyDe process networks communicate via signals according to a well-defined model of computation. The process network needs to be translated into a corresponding implementation in the target language that obeys the properties of the model of computation.
2. **Synthesis of processes created by process constructors.** Each process constructor has to be implemented into the corresponding pattern in the target implementation. As a second step, the arguments of the process constructors, i.e., pure functions and values, have to be translated into the target implementation.
3. **Synthesis of basic processes.** Basic processes like *zipSY* and *unzipSY* need to be implemented in the target implementation.

4.4.2 Hardware Synthesis

The general synthesis concept is illustrated by means of the synthesis of ForSyDe system models belonging to the synchronous MoC into digital hardware, more precisely into the corresponding VHDL code. The translation of synchronous ForSyDe models is straightforward. The synchronous MoC can be faithfully implemented in synchronous hardware, where the total order of events is preserved by the use of hardware clocks. Furthermore, both ForSyDe system models and VHDL models are based on concurrent processes communicating via signals.

Figure 4.17 shows how ForSyDe processes are translated to VHDL components, while ForSyDe signals are translated to VHDL signals.

In order to synthesize ForSyDe processes based on process constructors to the corresponding hardware, the corresponding pattern for each process constructor has to be identified in VHDL, and the arguments of the process constructors, pure functions and variables have to be translated to synthesizable VHDL. Figure 4.18 shows the translation of the ForSyDe process constructors *mapSY*, *delaySY*, and

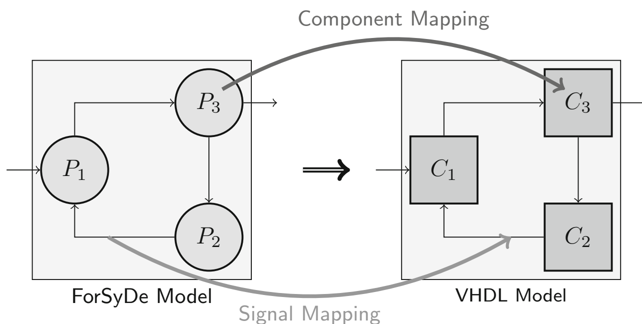

Fig. 4.17 Hardware synthesis of process networks

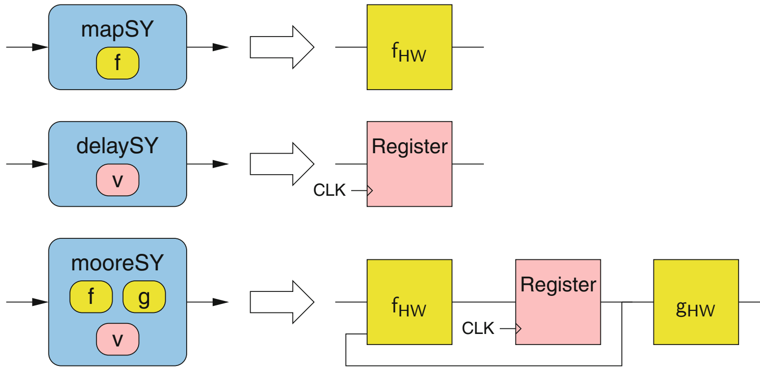

Fig. 4.18 Hardware synthesis of process constructors

mooreSY into the corresponding hardware patterns. Finally, the side-effect-free ForSyDe functions, which are the arguments of the process constructors, are translated into the corresponding VHDL functions.

4.4.3 ForSyDe Hardware Synthesis Tool

Based on the concepts presented in the previous sections, a hardware synthesis back end has been developed for ForSyDe [1]. The hardware synthesis tool has been implemented as deep-embedded domain-specific language in contrast to the shallow-embedded version of ForSyDe discussed in Sect. 4.2, which has been developed for simulation. In the deep-embedded version of ForSyDe, the system knows about its own structure, and ForSyDe's embedded compiler can operate on the abstract syntax tree to perform different analysis and transformation activities, such as simulation of the system or translation into a target language, e.g., VHDL in the case of the hardware synthesis tool. Thus there is no need for external parsers to compile a deep-embedded ForSyDe model. So far the deep-embedded version mainly supports hardware synthesis from the synchronous MoC, but using the same technique different back ends, like SW synthesis to C, can be developed within the embedded compiler.

In order to get access to the internal structure of the system model, the deep-embedded version of ForSyDe relies on several advanced Haskell techniques, which also affects the syntax of ForSyDe system models. In particular the impact of Template Haskell [55] is clearly visible in the definition of ForSyDe functions, but enables that all details of the system model are known to the embedded compiler at compile time. All examples in this section have been modeled with the `forsyde-deep` library, which is available on https://github.com/forsyde/forsyde-deep, and have been run using version 7.10.3 of the Glasgow Haskell Compiler `ghc`.

Listing 6 shows the synthesizable model of a counter in deep-embedded ForSyDe. The counter consists of a single process `counterProc`, which is created

Listing 6 A synthesizable counter in the deep-embedded version of Haskell-ForSyDe

```
1  -- Enable Language Extension: Template Haskell
2  {-# LANGUAGE TemplateHaskell #-}
3
4  module CounterHW (Direction, counterSys) where
5
6  import ForSyDe.Deep
7  import Data.Int
8
9  type Direction = Bit
10
11 nextStateFun :: ProcFun (Int8 -> Direction -> Int8)
12 nextStateFun = $(newProcFun
13    [d| nextState state dir
14           = if dir == H then
15               if state < 9 then state + 1
16               else 0
17             else
18               if state == 0 then 9
19               else state - 1
20       |])
21
22 counterProc :: Signal Direction -> Signal Int8
23 counterProc = scanldSY "counterProc" nextStateFun 0
24
25 counterSys :: SysDef (Signal Direction -> Signal Int8)
26 counterSys = newSysDef counterProc "Counter" ["direction"]
27               ["number"]
```

as finite state machine without output decoder by means of the process constructor `scanldSY` and the function argument `nextStateFun`. The reason for the special syntax inside `nextStateFun`, i.e., `$(newProcFun [d| ... |])`, is the use of Template Haskell to get access to the internal structure of the system model. Deep-embedded ForSyDe introduces the concept of *system* as a new hierarchical level. A system has a name, input and output ports, and there is full information about its internal structure, so that functions operating on the internal structure of the system can be defined for analysis, simulation or synthesis. An example is the simulation command `simulate` that enables the simulation of a system.

```
*CounterHW> simulate counterSys [L,H,H,H,H,L,L,L,L]
[0,9,0,1,2,3,2,1,0]
```

New systems can be created by instantiation of system components and their composition into a new system as illustrated in Listing 7, where a new system is created through composition of the counter and a seven-segment decoder. Please note that in order to define vectors of fixed size, which is critical for hardware

systems, a new synthesizable data type for fixed-sized vectors `FSVec` has been defined for ForSyDe. The size of the vector is a part of the type and given by a data type constructor `D`x, where x is the size of the vector.

Listing 7 Larger systems can be composed of instantiated components using a set of equations

```
1  module CounterSystemHW where
2
3  import ForSyDe.Deep
4  import CounterHW
5  import SevenSegmentDecoderHW
6  -- omitted additional import declarations
7
8  systemProc :: Signal Direction -> Signal (FSVec D7 Bit)
9  systemProc dir = sevenSeg
10    where
11       sevenSeg   = (instantiate "sevenSegDec" sevenSegDecSys)
12                    counterOut
13       counterOut = (instantiate "counter" counterSys) dir
14
15 system :: SysDef (Signal Direction -> Signal (FSVec D7 Bit))
16 system = newSysDef systemProc "system" ["in"] ["out"]
```

Listing 8 The ForSyDe synthesis tool interacts directly with the Altera Quartus tool

```
1  compileQuartus_CounterSystem :: IO ()
2  compileQuartus_CounterSystem = writeVHDLOps vhdlOps system
3   where
4     vhdlOps = defaultVHDLOps{execQuartus=Just quartusOps}
5     quartusOps
6       = QuartusOps{action=FullCompilation,
7                    fMax=Just 50, -- in MHz
8                    fpgaFamiliyDevice=Just ("CycloneII",
9                                            Just "EP2C35F672C6"),
10                   pinAssigs=[("in", "PIN_N25"),      -- SW0
11                              ("resetn", "PIN_N26"), -- SW1
12                              ("clock","PIN_G26"),    -- KEY[0]
13                              ("out[6]","PIN_AF10"), -- HEX0[0]
14                               ...
15                              ("out[0]","PIN_V13") ] -- HEX0[6]
16                  }
```

The deep-embedded ForSyDe tool provides a direct link to the Altera Quartus synthesis tool. The ForSyDe compiler generates the VHDL code and passes it together with optional design constraints and pin assignments to Quartus, which

generates the netlist for the circuit. Listing 8 shows the ForSyDe code for the synthesis of the counter to an Altera DE2/35 University board.

4.5 SystemC-ForSyDe

The formal definition of ForSyDe is based on the functional programming paradigm, so a pure functional language like Haskell is a perfect fit for ForSyDe. Nevertheless, the ForSyDe modeling framework is language independent. To make ForSyDe attractive for industrial designers, a SystemC version of the ForSyDe modeling framework has been created, which follows the spirit and semantics of the formal ForSyDe framework. This section gives a short overview about the nature of SystemC-ForSyDe by using the synchronous counter example from Fig. 4.4. A more detailed discussion on SystemC-ForSyDe is given in [2].

Listing 9 shows the code for the counter in SystemC-ForSyDe, which has the same structure as the corresponding Haskell model of Listing 1. The functions for the next state and the output need to be declared as side-effect-free functions, and are arguments for the combinational process constructors `scomb2` and `scomb` to create the processes `nextstate` and `outputdecode`. Here, `scomb2` and `scomb` are special versions of `comb2` and `comb`, requiring present events as inputs. Process constructors are implemented as C++ template classes where the template parameters determine the input/output types, and the constructor arguments are either values or functions. The process `del1` is created with the `delay`-process constructor. Finally, process networks can be expressed in SystemC-ForSyDe as *composite processes*, which can be regarded as netlists created by binding signals to processes through the newly introduced concept of *ports*.

A SystemC-ForSyDe model is also aware of its internal structure, which means that introspection can be used to operate on the system structure. Hence, it is possible to extract graph representations, which can be used for subsequent phases in the design flow, such as design space exploration or synthesis. Figure 4.19 shows an automatically generated graphical representation of the SystemC model from Listing 9 to illustrate the capabilities of introspection in SystemC-ForSyDe.

SystemC-ForSyDe supports a similar set of MoCs as Haskell-ForSyDe. All the MoCs discussed in the Sect. 4.2.3 are also supported by SystemC-ForSyDe libraries, which are publicly available from the ForSyDe web page [19].

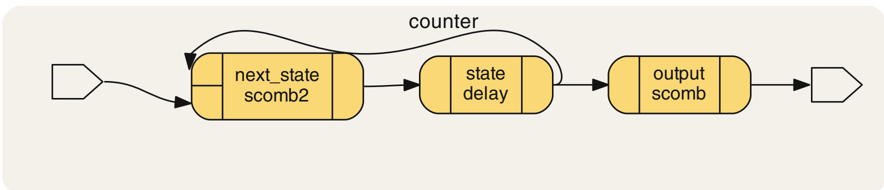

Fig. 4.19 Generated graphical representation of the counter in SystemC using the tool `f2dot`

Listing 9 The counter example in the SystemC version of ForSyDe

```
1  #include <forsyde.hpp>
2
3  using namespace ForSyDe;
4
5  typedef enum Direction { up, hold } Direction;
6  typedef enum Clock     { tick }     Clock;
7
8  void nextstate_f(int& ns, const int& s, const Direction& in) {
9      switch (in) { case up   : ns = (s + 1)
10                   case hold : ns = s; break; }
11 }
12
13 void output_f(abst_ext<Clock>& out, const int& s) {
14     switch (s) { case 0  : out = abst_ext<Clock>(tick); break;
15                  default : out = abst_ext<Clock>(); }
16 }
17
18 SC_MODULE(counter) {
19    // Declaration of inputs, outputs and intermediate signals
20    SY::in_port<Direction>         input;
21    SY::out_port<abst_ext<Clock>> output;
22    SY::signal<int>                next_state, state1, state2;
23
24    // Module architecture: constructing processes and binding
25    signals SC_CTOR(counter) {
26       // Process that computes the next state
27       auto nextstate = new SY::scomb2<int, int, Direction>
28                                     ("next_state", nextstate_f);
29       nextstate->iport1(state1);
30       nextstate->iport2(input);
31       nextstate->oport1(next_state);
32
33       // Sequential process that stores the state value
34       auto del1 = new SY::delay<int>("state", 0);
35       del1->iport1(next_state);
36       del1->oport1(state1);
37       del1->oport1(state2);
38
39       // Process that decodes the output
40       auto outputdecode = new SY::scomb<abst_ext<Clock>, int>
41                                         ("output", output_f);
42       outputdecode->iport1(state2);
43       outputdecode->oport1(output);
44    }
45 };
```

4.6 Related Work

Researchers have for many years promoted methods that push the design entry to a higher level of abstraction by the usage of formal models and transformations in the design process. In 1997, Edwards et al. [16] expressed their beliefs “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.” Furthermore, they stated that “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.’” In 1998, Skillicorn and Talia discussed models of computation for parallel architectures in [58]. They argued that “a model must hide most of the details from programmers if they are to be able to manage, intellectually, the creation of software” and that “as much as possible of the exact structure of the executing program should be inserted by the translation mechanism (compiler and run-time system) rather than by the programmer.” Furthermore, they pointed out that “models ought to be as abstract and simple as possible.” Also Keutzer et al. [28] stressed that “to be effective a design methodology that addresses complex systems must start at high levels of abstraction.” They promoted “the use of formal models and transformations in system design so that verification and synthesis can be applied to advantage in the design methodology” and argued that “the most important point for functional specification is the underlying mathematical model of computation.”

Although the arguments for a more formal and disciplined design approach have been known for a long time, there still exists no formal and systematic design methodology that can be employed in an industrial setting. The IEEE standard language SystemC [26] has been inspired by SpecC [15] and is implemented as C++ class library. SystemC provides a discrete-event simulation kernel, and has been the base for several approaches towards a more formal design process. SystemC-AMS [59], an extension of SystemC based on a timed data-flow MoC, enables the modelling of analog and hybrid systems. HetSC [24] is based on the standard SystemC kernel and targets heterogeneous systems. HetSC supports the designer by a set of modeling primitives with additional supporting rules and guidelines for different MoCs. In comparison to ForSyDe, this approach is less formal, since it is difficult to enforce well-defined models. SysteMoC [18], which is described in more detail in ▸ Chap. 3, “SysteMoC: A Data-Flow Programming Language for Codesign”, is based on SystemC and addresses dynamic data-flow applications. SysteMoC’s modeling technique enables to describe both statically analyzable actors and expressive dynamic actors. An actor is modeled as a state machine that separates the computational part of the actor from the processing of tokens, which is modeled as actor state machine. SysteMoC provides analysis methods operating on the actor state machine that can detect the analyzable portion of the system model, for which

powerful data-flow analysis methods exist. The Ptolemy project [17, 44] studies the usage of well-defined MoCs for the design of heterogeneous, concurrent embedded and cyber-physical systems. Ptolemy uses an actor-oriented model, where actors communicate with each other by sending messages. Ptolemy introduces the concept of director, where the director defines the semantics of the underlying set of actors and thus specifies the MoC. Heterogeneous models can be created due to the concept of hierarchy, where each director defines the interaction of the actors on its own level. ForSyDe supports a generic mechanism, where each process is associated with a MoC and where the execution of the process network is only based on data dependences. Thus no central synchronization is required, which is a prerequisite for efficient simulation on parallel and distributed architectures.

Several researchers have used declarative languages to address system design from a more formal perspective. Reekie used Haskell for the modeling purpose of digital signal processing applications, where streams are modeled as infinite lists and processes are created by higher-order functions [46]. Reekie also proposed semantic-preserving transformations based on equational reasoning to convert a model into a more efficient form. The relational language Ruby has been developed for the design of hardware circuits using a structural representation of basic hardware components and connection patterns. This structured concept has been extended for software to formulate a vision on HSCD [33] based on Ruby. Lava [10] borrows many concepts from Ruby, but is embedded in the functional language Haskell. It provides a variety of powerful connection patterns, access to formal methods and translation to VHDL. There exist several versions of Lava: Chalmers Lava [10], Xilinx Lava [56], and most recently Kansas Lava [22]. The goal of Kansas Lava is to scale up the ideas in Lava to operate on larger circuits and to use larger basic components. In contrast to the structural approach of Lava and Ruby, Mycroft and Sharpe have taken a behavioral approach as base for the development of the languages SAFL (statically allocated functional language) and SAFL+ [54]. Although these languages have been primarily designed for hardware design, they have been used in [40] for HSCD. SAFL offers the application of semantic-preserving transformations and can synthesize programs in a resource-aware style, where functions that are called several times result in shared resources. The Hawk [35] language, based on Haskell, addresses the modeling, simulation and verification of microprocessors, where it exploits the formal base of Haskell. Hardware ML (HML) [31] is based on the functional programming language Standard ML [39]. It has been designed as an alternative to VHDL with a direct mapping of the HML constructs to the corresponding VHDL code. Cλash [3] is a functional hardware description language that uses a subset of Haskell for the purpose of describing hardware. Haskell functions denote components and the Cλash-compiler converts Cλash-descriptions into the corresponding hardware implementation. There are several approaches to generate target code for GPUs from Haskell-based domain-specific languages. Examples are Nikola [34] and Obsidian [60].

ForSyDe has been inspired by the work of Reekie and is based on the same modeling approach for signals and processes. The work of Mycroft and Sharp

shares the same ambition as ForSyDe to move system design to a higher level of abstraction, but followed a different modeling approach. Furthermore, refinement is restricted to semantic-preserving transformations. Cλash, Lava, Ruby, and HML are developed as languages for hardware design and operate on a lower abstraction level than ForSyDe. Their main objective is to provide a formally sound alternative to VHDL and Verilog. In contrast, ForSyDe addresses the modeling and design of embedded and cyber-physical design, and views VHDL merely as a target language. Hawk focuses on processor design with a focus on modeling and verification of instruction set and architecture, and does not support hardware synthesis. Furthermore, ForSyDe is the only declarative approach that is based on model of computation theory and that provides several models of computation and MoC interfaces.

The ForSyDe concept of process constructors is heavily influenced by the work of Skillicorn on *homomorphic skeletons* [57]. The term skeleton, coined by Cole [13] in his seminal work on *algorithmic skeletons*, 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, because 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. Using the Bird-Meertens formalism, Bird demonstrates how to derive programs from specifications by equational reasoning using lists [8], arrays, and trees [9] 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 influential CIP (computer-aided, intuition-guided programming) project investigated transformational program construction [4]. CIP follows a top-down approach that starts with a formulation of the formal specification which is then converted via well-defined semantic-preserving transformations into a final program. The authors stated the following advantages of this approach in [4]: (a) the final program is correct by construction; (b) the transitions can be described by schematic rules and thus be reused for a whole class of problems; (c) due to formality the whole process can be supported by the computer; (d) the overall structure is no longer fixed throughout the development process, so that the approach is quite flexible.

The development of a successful practical design transformation system is a huge challenge. The transformation framework does not only need to provide a sufficient number of transformation rules, but has also to derive a sequence of transformations steps that yield a correct and efficient implementation. So far transformational approaches have mainly been used for small general purpose programming modules with a high demand on formal correctness. The problem is aggravated in the embedded systems domain, because of extra-functional

design constraints and restricted resources. Most approaches for transformational hardware design are restricted to semantic-preserving transformations [43, 53], while ForSyDe's support of nonsemantic transformations enables to integrate the refinement techniques in the area of high-level synthesis [21,37] as design decisions. The result of transformational design refinement from a high-level general purpose language is largely dependent on the initial specification due to the very large design space that can only partly explored [62]. The problem is of fundamental character and also known as syntactic variance problem in high-level synthesis [20]. The problem is naturally smaller in domain-specific languages like ForSyDe with a smaller set of building blocks, but it cannot be eliminated still exists. A more detailed overview on program transformation is given in [41], while [42] concentrates on transformation techniques for functional and logical programs.

4.7 Conclusion

The ForSyDe design methodology aims at pushing system design to a higher level of abstraction and provides means to enable a correct-by-construction design flow. ForSyDe is based on a solid formal foundation in form of a well-defined functional system model and a theoretical base in form of model of computation theory. The chapter gave an overview about the key concepts in ForSyDe and illustrated its heterogeneous system modeling using its Haskell version, which can be viewed as a perfect match with the underlying formal framework. Furthermore, the chapter also discussed how to convert an abstract system model into a final implementation by transformational design refinement followed by system synthesis. The ForSyDe framework is language-independent and can even be realized in other languages, which has been demonstrated by the SystemC version of ForSyDe that has been developed for industrial designers.

ForSyDe is an active research project that covers the whole design flow. In addition to system modeling, design refinement and system synthesis, current research focuses also on the development of a design flow for mixed-criticality multi-processor applications sharing the same platform. Here, the underlying formal models of computation and the concept of process constructors enable to create analysis models from the ForSyDe system models. These analysis models can then be used in the critical design space exploration activity to find efficient implementations on shared multi-processor platforms. A first implementation of the design space exploration (DSE) tool, which uses constraint programming and takes communication on shared resources into account, is presented in [47]. Once an efficient mapping with the corresponding schedules has been calculated by the DSE tool, the schedule and the function arguments of the process constructors need to be synthesized to the processors on the target platform. Future research will develop the synthesis concepts for multi-processor platforms based on the synthesis concepts presented in Sect. 4.4.

Appendix: Introduction to Haskell

This section gives a short introduction to functional languages and Haskell to give readers who are not familiar with functional programming additional background information. For more information on Haskell, visit the Haskell home page [61].

A functional program is a function that receives the program's input as argument and delivers the program's output as result. The main function of a program 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. Haskell is a pure functional language, where each function is free from side-effects. This means given the same inputs, which in case of ForSyDe could be a set of input signals, a Haskell function will always produce identical outputs. Thus the whole functional program is free from side-effects and thus behaves totally deterministic. 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

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

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.

In addition to common data types, such as `Bool`, `Int`, and `Double`, Haskell 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 [38], which is not only strongly typed but also uses type inference to determine the type of every expression instead of relying on explicit-type declarations.

Haskell is based on the lambda-calculus and allows to write functions in *curried* form, where the arguments are written by juxtaposition. The following Haskell 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 type of `add` can also be read as

```
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 enables *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
```

Another powerful concept in functional languages is the *higher-order function*, which is adopted in ForSyDe for process constructors. 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 `map` 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 `map` uses an additional feature of Haskell, 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 : 3+1 : 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]
```

Another important 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 allows to merge processes in a structured way.

Haskell allows to define own data types using a `data` declaration. It allows for recursive and polymorphic 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, the list data type is predefined in Haskell. Here `[]` corresponds to `Empty` and `:` to `Cons`. `[a]` corresponds to `AList a`. The ForSyDe `Signal` is defined in the same way as the data type `AList`, see Sect. 4.2.1.

References

1. Acosta A (2007) Hardware synthesis in ForSyDe. Master's thesis, School for Information and Communication Technology, Royal Institute of Technology (KTH), Stockholm. KTH/ICT/ECS-2007-81
2. Attarzadeh Niaki S, Jakobsen M, Sulonen T, Sander I (2012) Formal heterogeneous system modeling with SystemC. In: Forum on specification and design languages (FDL 2012), Vienna, pp 160–167
3. Baaij C, Kooijman M, Kuper J, Boeijink A, Gerards M (2010) CλaSH: structural descriptions of synchronous hardware using Haskell. In: 2010 13th Euromicro conference on digital system design: architectures, methods and tools (DSD), pp 714–721
4. Bauer FL, Möller B, Partsch H, Pepper P (1989) Formal program construction by transformations – computer-aided, intuition guided programming. IEEE Trans Softw Eng 15(2):165–180
5. Benveniste A, Berry G (1991) The synchronous approach to reactive and real-time systems. Proc IEEE 79(9):1270–1282
6. Benveniste A, Caspi P, Edwards SA, Halbwachs N, Le Guernic P, Simone RD (2003) The synchronous languages 12 years later. Proc IEEE 91(1):64–83

7. Berry G, Gonthier G: The Esterel synchronous programming language: design, semantics, implementation. Sci Comput Program 19(2):87–152 (1992)
8. Bird RS (1986) An introduction to the theory of lists. Technical monograph PRG-56 edn. Oxford University Computing Laboratory
9. Bird RS (1988) Lectures on constructive functional programming. Technical Monograph PRG-69 edn. Oxford University Computing Laboratory
10. Bjesse P, Claessen K, Sheeran M, Singh S (1998) Lava: hardware design in Haskell. In: International conference on functional programming, Baltimore, pp 174–184
11. Blindell GH, Menne C, Sander I (2014) Synthesizing code for GPGPUs from abstract formal models. In: Forum on specification and design languages (FDL 2014), Munich
12. Boussinot F, De Simone R (1991) The Esterel language. Proc IEEE 79(9):1293–1304
13. Cole M (1989) Algorithmic skeletons: structured management of parallel computation. Research monographs in parallel and distributed computing. Pitman, London
14. Derler P, Lee E, Sangiovanni-Vincentelli A (2012) Modeling cyber-physical systems. Proc IEEE 100(1):13–28
15. Dömer R, Gerstlauer A, Gajski D (2002) SpecC language reference manual, version 2.0
16. Edwards S, Lavagno L, Lee EA, Sangiovanni-Vincentelli A (1997) Design of embedded systems: formal models, validation, and synthesis. Proc IEEE 85(3):366–390
17. Eker J, Janneck J, Lee E, Liu J, Liu X, Ludvig J, Neuendorffer S, Sachs S, Xiong Y: Taming heterogeneity – the Ptolemy approach. Proc IEEE 91(1):127–144 (2003)
18. Falk J, Haubelt C, Teich J (2006) Efficient representation and simulation of model-based designs in SystemC. In: Proceedings of the forum on specification and design languages (FDL), vol 6, pp 129–134
19. ForSyDe: Formal system design. https://forsyde.ict.kth.se/
20. Gajski DD, Ramachandran L (1994) Introduction to high-level synthesis. IEEE Des Test Comput 11(4):44–54
21. Gajski DD, Dutt ND, Wu ACH, Lin SYL (1992) High-level synthesis. Kluwer Academic, Boston
22. Gill A, Bull T, Kimmell G, Perrins E, Komp E, Werling B (2010) Introducing kansas Lava. In: Morazán M, Scholz SB (eds) Implementation and application of functional languages. Lecture notes in computer science, vol 6041. Springer, Berlin/Heidelberg, pp 18–35
23. Halbwachs N, Caspi P, Raymond P, Pilaud D (1991) The synchronous data flow programming language Lustre. Proc IEEE 79(9):1305–1320
24. Herrera F, Villar E (2008) A framework for heterogeneous specification and design of electronic embedded systems in SystemC. ACM Trans Des Autom Electron Syst 12(3): 22:1–22:31
25. Herrholz A, Oppenheimer F, Hartmann PA, Schallenberg A, Nebel W, Grimm C, Damm M, Haase J, Brame J, Herrera F, Villar E, Sander I, Jantsch A, Fouilliart AM, Martinez M (2007) The ANDRES project: analysis and design of run-time reconfigurable, heterogeneous systems. In: International conference on field programmable logic and applications (FPL'07), pp 396–401
26. IEEE Standard for Standard SystemC Language Reference Manual. IEEE Std 1666–2011 (Revision of IEEE Std 1666–2005), pp. 1–638. http://ieeexplore.ieee.org/document/6134619/
27. Jantsch A (2005) Models of embedded computation. In: Zurawski R (ed) Embedded systems handbook. CRC Press, Boca Raton. Invited contribution
28. Keutzer K, Malik S, Newton AR, Rabaey JM, Sangiovanni-Vincentelli A (2000) System-level design: orthogonolization of concerns and platform-based design. IEEE Trans Comput-Aided Des Integr Circuits Syst 19(12):1523–1543
29. Lee EA, Messerschmitt DG (1987) Synchronous data flow. Proc IEEE 75(9):1235–1245
30. Lee EA, Sangiovanni-Vincentelli A (1998) A framework for comparing models of computation. IEEE Trans Comput-Aided Des Integr Circuits Syst 17(12):1217–1229
31. Li Y, Leeser M (2000) HML, a novel hardware description language and its translation to VHDL. IEEE Trans VLSI 8(1):1–8

32. Lu Z, Sander I, Jantsch A (2002) A case study of hardware and software synthesis in ForSyDe. In: Proceedings of the 15th international symposium on system synthesis, Kyoto, pp 86–91
33. Luk W, Wu T (1994) Towards a declarative framework for hardware-software codesign. In: Proceedings of the third international workshop on hardware/software codesign, Grenoble, pp 181–188
34. Mainland G, Morrisett G (2010) Nikola: embedding compiled GPU functions in Haskell. In: Proceedings of the third ACM Haskell symposium on Haskell, Haskell '10. ACM, New York
35. Matthews J, Cook B, Launchbury J (1998) Microprocessor specification in HAWK. In: International conference on computer languages (ICCL'98), pp 90–101
36. McKinley PK, Sadjadi SM, Kasten EP, Cheng BH (2004) Composing adaptive software. IEEE Comput 37(7):56–64
37. Micheli GD (1994) Synthesis and optimization of digital circuits. McGraw-Hill, New York
38. Milner R (1978) A theory of type polymorphism in programming. J Comput Syst Sci 17: 348–375
39. Milner R, Tofte M, Harper R, MacQueen D (1997) The definition of standard ML – revised. MIT, Cambridge
40. Mycroft A, Sharp R (2000) Hardware/software co-design using functional languages. In: Proceedings of tools and algorithms for the construction and analysis of systems (TACAS). LNCS, vol 2031. Springer, pp 236–251
41. Partsch HA (1990) Specification and transformation of programs. Springer, New York
42. Pettorossi A, Proietti M (1996) Rules and strategies for transforming functional and logic programs. ACM Comput Surv 28(2):361–414
43. Plosila J (1999) Self-timed circuit design – the action systems approach. PhD thesis, University of Turku, Turku
44. Ptolemaeus C (ed) (2014) System design, modeling, and simulation using Ptolemy II. Ptolemy.org. http://ptolemy.org/books/Systems
45. Raudvere T, Sander I, Jantsch A (2008) Application and verification of local non-semantic-preserving transformations in system design. IEEE Trans Comput-Aided Des Integr Circuits Syst 27(6):1091–1103
46. Reekie H (1995) Realtime signal processing. PhD thesis, School of Electrical Engineering, University of Technology at Sydney
47. Rosvall K, Sander I (2014) A constraint-based design space exploration framework for real-time applications on MPSoCs. In: Design automation and test in Europe (DATE '14), Dresden
48. Sander I (2003) System modeling and design refinement in ForSyDe. PhD thesis, Royal Institute of Technology, Stockholm
49. Sander I, Jantsch A (2004) System modeling and transformational design refinement in ForSyDe. IEEE Trans Comput-Aided Des Integr Circuits Syst 23(1):17–32
50. Sander I, Jantsch A (2008) Modelling adaptive systems in ForSyDe. Electron Not Theor Comput Sci (ENTCS) 200(2):39–54
51. Sangiovanni-Vincentelli A (2007) Quo vadis, SLD? Reasoning about the trends and challenges of system level design. Proc IEEE 95(3):467–506
52. Schneider K (2009) The synchronous programming language Quartz. Internal report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern
53. Seceleanu T (2001) Systematic design of synchronous digital circuits. PhD thesis, University of Turku, Turku
54. Sharp R, Mycroft A (2001) A higher level language for hardware synthesis. In: Proceedings of 11th advanced research working conference on correct hardware design and verification methods (CHARME). LNCS, vol 2144. Springer, pp 228–243
55. Sheard T, Jones SP (2002) Template meta-programming for Haskell. ACM SIGPLAN Not 37(12):60–75
56. Singh S, James-Roxby P (2001) Lava and JBits: from HDL to bitstream in seconds. In: Proceedings of the 9th annual IEEE symposium on field-programmable custom computing machines, FCCM '01, pp 91–100

57. Skillicorn DB (1994) Foundations of parallel programming. Cambridge international series on parallel computation. Cambridge University Press, Cambridge/New York
58. Skillicorn DB, Talia D (1998) Models and languages for parallel computation. ACM Comput Surv 30(2):123–169
59. Standard SystemC AMS extensions 2.0 language reference manual (2013)
60. Svensson J, Sheeran M, Claessen K (2011) Obsidian: a domain specific embedded language for parallel programming of graphics processors. In: Scholz SB, Chitil O (eds) Implementation and application of functional languages. Lecture notes in computer science, vol 5836. Springer, Berlin/Heidelberg, pp 156–173
61. The Haskell language. http://www.haskell.org
62. Voeten J (2001) On the fundamental limitataions of transformational design. ACM Trans Des Autom Electron Syst 6(4):533–552