Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 200 (2008) 39–54
www.elsevier.com/locate/entcs

Modelling Adaptive Systems in ForSyDe[1]

Ingo Sander [2] and Axel Jantsch [3]

School of Information and Communication Technology
Royal Institute of Technology
Stockholm, Sweden

Abstract

Emerging architectures such as partially reconfigurable FPGAs provide a huge potential for adaptivity in the area of embedded systems. Since many system functions are only executed at particular points of time they can share an adaptive component with other system functions, which can significantly reduce the design costs. However, adaptivity adds another dimension of complexity into system design since the system behaviour changes during the course of adaptation. This imposes additional requirements on the design process, in particular system verification. In this paper we illustrate how adaptivity is treated as first-class citizen inside the ForSyDe design framework. ForSyDe is a transformational system design methodology, where an initial abstract system model is refined by the application of semantic-preserving and non-semantic preserving design transformations into a detailed model that can be mapped to an implementation. Since ForSyDe is based on the functional paradigm we can model adaptivity by using functions as signal values, which we use as the base for our concept of adaptive processes. Depending on the level of adaptivity we categorise four classes of adaptive process, spanning from parameter adaptive to interface adaptive process. We illustrate our concepts by two typical examples for adaptivity, where we also show the application of design transformations.

Keywords: Adaptive systems, formal modelling of adaptivity, transformational design, embedded systens

1 Introduction

The complexity of high-end embedded systems is rapidly increasing. Broy et al. point out that today the user of a premium car interacts with about 270 functions, which are deployed over about 70 embedded platforms. The total amount of software in such a car is about 100 MB of binary code, but already in about five years upper class cars are expected to run 1 GB of software [7]. Since embedded systems interact with the physical environment and are inherently parallel and heterogeneous, their implementation consists of different domains: software, analog hardware, static digital hardware and dynamically reconfigurable digital hardware.

[1] The research has been conducted inside the project ANDRES, which is supported by the sixth framework programme of the European Commission.

[2] Email: ingo@kth.se

[3] Email: axel@kth.se

doi:10.1016/j.entcs.2008.02.011

Especially the latter is gaining in importance due to its combination of flexibility and efficiency. Today there are FPGAs [4] on the market supporting partial reconfiguration. That is, one part of the circuit can be reconfigured, while its remainder is not affected and can continue its operation [9]. Since many functions are only executed at particular points of time, dynamic adaptation to the current situation can significantly reduce the cost of embedded systems. Although the integration of adaptivity into embedded system design offers a huge potential for lower costs and more efficient implementations, it also significantly increases the complexity of the design process. The design process must ensure that the system works correctly not only before adaptation and after adaptation, but also during adaption. The design process of heterogeneous embedded systems is further complicated, since each implementation domain requires its own model of computation. Current design methodologies treat each domain independently with an own design flow and thus the integration of the different domains is a major obstacle. We are convinced that in order to cope with the complexity of adaptive embedded systems, the design process cannot rely on ad hoc approaches, but must be put on a solid formal basis.

ForSyDe [5] [20] is a transformational design methodology that targets heterogeneous embedded systems [12]. The system is modelled at an abstract level using a formal semantics and is stepwise refined by well-defined design transformations. The formal base of ForSyDe is ideally suited for the design of adaptive system. Since ForSyDe is based on the functional paradigm, functions can be used as signal values, which is the base for our concept of adaptive processes.

The rest of the paper is structured as follows. After the discussion of the related work in Section 2, the modelling concepts of ForSyDe are presented in Section 3. The core of the paper is Section 4. Here the concept of adaptive process is introduced and the modelling of adaptive systems is illustrated by two examples. Further we give examples for the application of design transformations. Finally Section 5 concludes the paper and gives also an overview of future work.

2 Related Work

The interest for adaptive computing systems has significantly increased due to a number of techniques that have become available in recent years. The March issue of IEEE Computer devotes a special section to reconfigurable computing [8]. There is a large interest on adaptive systems in several research communities, such as high-performance reconfigurable computing, autonomous systems and embedded systems. The occurrence of partially reconfigurable FPGAs has been a driver for this development, since now adaptivity cannot only be implemented in software, but even in more efficient hardware [3]. A good overview about reconfigurable architectures and related software tools can be found in [9].

Schneider and Schuele point out that the explicit modelling and analysis of dynamic adaptation in embedded systems is a young research area [22]. They stress

[4] Field Programmable Gate Array

[5] Formal System Design

that the adaptation behaviour of an embedded system does not only help to reduce costs, but also significantly complicates the design process. They use a model based on quality descriptions that models the adaptivity behaviour at an abstract level. This adaptive behaviour can then be verified using temporal logic and model checking. Zhang and Cheng [24] use a formal state-machine representation for adaptive processes and verify temporal logic specifications by model checking. This adaptation semantics has been integrated by Brown et al. into the KAOS methodology, which also provides a graphical view of the semantics [6]. Naji et al. [17] extend the multi-agent paradigm that is prevalent in distributed reactive software systems to adaptable embedded systems by implementing agents in reconfigurable hardware. Two approaches to implement adaptation for software systems are discussed in [16]. Parameter adaptation modifies program variables that determine behaviour. In contrast compositional adaptation exchanges system components with others, so that new functions can be integrated into the system during runtime. Our research targets both parameter adaptation, which is the dominating technique for analog hardware, and compositional adaptation, which is restricted to reconfigurable hardware and software.

The preceding approaches target different areas of adaptive systems and offer interesting ideas, but none of them targets adaptivity in a heterogeneous environment. Ptolemy [10] is the most prominent approach that targets the modelling of heterogeneous systems. They support models of computation spanning from continuous time to discrete time. Ptolemy uses an actor-oriented approach, where the composition semantics is based on interface automata. Although Ptolemy does not explicitly target reconfigurable systems, Neuendorffer and Lee [18] have proposed a model of parametrisation and reconfiguration for hierarchical data-flow models based on an abstract semantics. Instead ForSyDe uses the concept of process constructors to model processes in different models of computation. A process is composed by a process constructor belonging to a certain computational model and defining the interface together with functions and values, serving as process constructor arguments. ForSyDe is developed as design methodology for embedded systems, wheras Ptolemy is mainly a modelling environment. A ForSyDe process constructor establishes a separation of communication and computation, which allows a polymorphic application of design transformations [20]. Process constructors give also a structure to the model, which can be exploited in later design phases as indicated by mapping rules for synchronous ForSyDe models to hardware and software [15]. ForSyDe allows a smooth integration of different models of computation, since all models are based on the same concept of process constructors. Since ForSyDe is based on the functional paradigm, processes and functions have no side effect, which facilitates the application of formal methods. Heterogeneous ForSyDe models can be simulated in the functional language Haskell [14] using the ForSyDe library [2].

3 ForSyDe Modelling Framework

ForSyDe is a transformational system design methodology targeting heterogeneous embedded systems. In ForSyDe an initial abstract system model is refined by the application of semantic-preserving and non-semantic preserving design transformations into a detailed model that can be mapped to an implementation in hardware or software. This section presents the modelling concepts of ForSyDe. Design transformations are discussed in the context of adaptive systems in Section 4.3. A more detailed description of the ForSyDe methodology, in particular design transformations, is given in [20].

In ForSyDe a system is modelled as a hierarchical concurrent process network. Processes communicate with each other via signals. ForSyDe supports several models of computation (MoCs) and allows processes belonging to different models of computation to communicate via domain interfaces as illustrated in Figure 1.

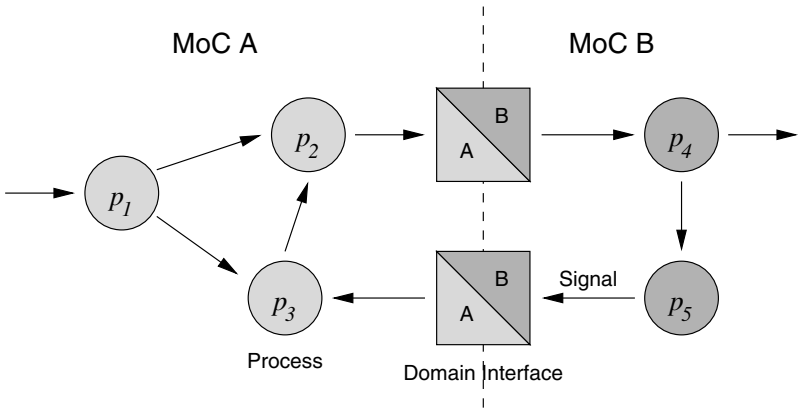


Fig. 1. A ForSyDe model is a hierarchical concurrent process network. Processes of different models of computation can communicate with each other via domain interfaces.

We start the description of the ForSyDe model with the general aspects of the modelling elements, in particular signals (Section 3.1) and processes (Section 3.2). At present ForSyDe provides a synchronous MoC, an untimed MoC and a discrete time MoC [13]. However, we will only discuss the synchronous MoC in Section 3.3, since we illustrate our concepts of adaptivity in Section 4 with that model. The concepts of adaptivity can be easily applied to other MoCs.

3.1 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 modelling framework we model a signal as a list of events, where the tag of the event is implicitly given by the event's position in the list. The interpretation of tags is defined by the model of computation, e.g. 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. Thus we write signals as $\langle e_0, e_1, e_2, \dots \rangle$, where e_i denotes the value of the i-th event of 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 where all values are of type D is denoted $S(D)$.

3.2 Processes

Processes are defined as functions on signals

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

The set of all processes is P.

Processes are functions in the sense that for a given set of input signals we always get the same set of output signals. 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. Thus, 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.

For processes with arbitrary number of input and output signals the notation can become cumbersome to read. Hence for the sake of simplicity we deal sometimes with processes with one input and one output only, which is not a lack of generality since it is straight forward to introduce "zip" and "unzip" processes which merge two input signals into one and split one output signal into two output signals, respectively [13]. These processes together with appropriate process composition allows us to express arbitrary behaviour.

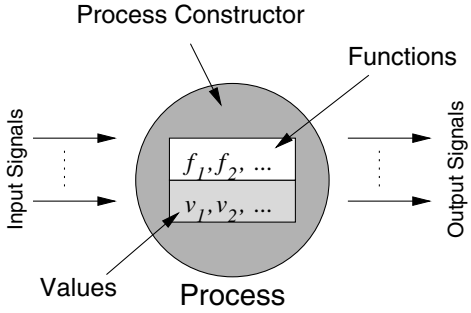


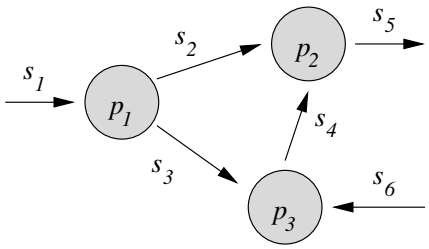
Fig. 2. A process is constructed by means of a process constructor that takes functions and values as argument.

Figure 2 illustrates the concept of *process constructor*, which is a key element in ForSyDe. ForSyDe defines a set of well-defined process constructors, which are used to create processes. A process constructor pc takes zero or more functions $f_1, f_2, \dots, f_n$ and zero or more values $v_1, v_2, \dots, v_n$ as arguments and returns a process $p \in P$.

$$p = pc(f_1, f_2, \dots, f_n, v_1, v_2, \dots, v_n)$$

The functions represent the process behaviour 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. It defines the time representation, the communication and synchronisation semantics. This separation of concerns leads to an elegant mathematical formalism that facilitates design analysis and transformation.

A set of process constructors determines a particular model of computation. This leads to 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 put constraints on functions. For instance, the synchronous MoC requires a function to take exactly one event on each input and produce exactly one event for each output. The untimed MoC does not have a similar requirement.



$$
\begin{aligned}
pn(s_1, s_6) = s_5 \quad & \text{where} \\
pn \in P & \\
(s_2, s_3) &= p_1(s_1) \\
s_5 &= p_2(s_2, s_4) \\
s_4 &= p_3(s_3, s_6)
\end{aligned}
$$

Fig. 3. A process network can be expressed by a set of equations.

New processes can be created by composition of other processes. Figure 3 shows a process network and the corresponding set of equations. The process network is in itself a process.

3.3 Synchronous Model of Computation

The family of synchronous languages [4] [5] is based on the synchronous model of computation, which uses the perfect synchrony assumption. Among others the languages Esterel, Lustre, Signal and StateCharts are all based on the synchronous model of computation.

Perfect synchrony hypothesis: *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 in the synchronous language.

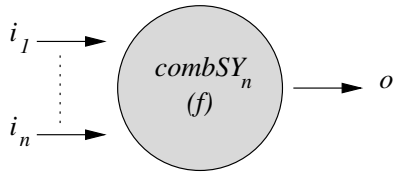
Synchronous processes are defined by the following specific characteristic. All synchronous processes consume and produce exactly one event on each input or

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, there is 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*.

In the following we give a set of basic process constructors, which are needed to model a system in the synchronous model of computation. 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 the set of combinational process constructors $combSY_n$ [6] and the delay process constructor $delaySY$ are sufficient to model a system inside the synchronous MoC.

A combinational process constructors $combSY_n$ takes a function $f : D_1 \times \cdots \times D_n \to E$ as argument and returns a process $p : S(D_1) \times \cdots \times S(D_n) \to S(E)$ with no internal state. Figure 4 shows the combinational process constructor $combSY_n$, which takes a function f as argument and constructs a combinational process with n input signals.



$$p = combSY_n(f) \in P_{SY}$$

where

$$o = p(i_1, \ldots, i_n)$$

$$o[k] = f(i_1[k], i_2[k], \ldots, i_n[k])$$

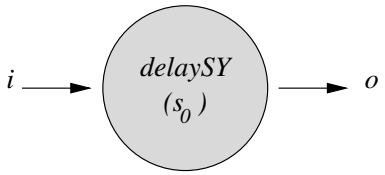
Fig. 4. A process constructor $combSY_n$ creates a combinational synchronous process.

The delay process constructor $delaySY$ takes only one value $s_0 : D$ as argument and produces a process $p : S(D) \to S(D)$ that delays the input signal one cycle. The supplied value is the initial value of the output signal. Figure 5 shows the process constructor $delaySY$ that creates a process, which delays the input signal one cycle.

Other process constructors can be defined for convenience, such as the state machine constructor $stateSY_n$, which is used to model a state machine. Though all state machines can be composed by a net-list of combinational and delay processes, state machine process constructors prove to be very useful as designers are used to the concept of state machines.

Figure 6 shows the process constructor $stateSY_n$ that takes a function $f : D_1 \times \cdots \times D_n \times E \to E$ and an initial value $s_0 : E$ and returns a synchronous process $p : S(D_1) \times \cdots \times S(D_n) \to S(E)$ that models a state machine without output

[6] Previous ForSyDe versions defined the process constructors $mapSY$ and $zipWithSY_n$ instead of the process constructor $combSY_n$. Only the naming has changed, there has not been a change in the formal definition. $mapSY$ corresponds to $combSY_1$ and $zipWithSY_n$ to $combSY_n$ (for $n > 1$).



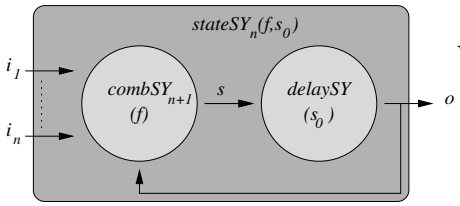
$$p = delaySY(s_0) \in P_{SY}$$

where

$$o = p(i)$$

$$o[k] = \begin{cases} s_0 & k = 0 \\ i[k-1] & k > 0 \end{cases}$$

Fig. 5. The process constructor *delaySY* creates a synchronous process that delays the input signal one cycle.



$$p = stateSY_n(f, s_0) \in P_{SY}$$

where

$$o = p(i_1, i_2, \ldots, i_n)$$

$$o = (delaySY(s_0))(s)$$

$$s = (combSY_{n+1}(f))(i_1, \ldots, i_n, o)$$

Fig. 6. The process constructor $stateSY_n$ creates a synchronous process that models a state machine without output decoder.

decoder. Other state machines process constructors, e.g. for Moore and Mealy machines, can be based on this process constructor.

4 Modelling of Adaptivity

We extend the ForSyDe framework to model adaptivity on several levels of abstraction. Here the key concept is to use functions in the same way as variables of normal data types, in particular we introduce signals that carry functions. Thus we can define the following signal

$$s_f = \langle (+), (-), (+), \ldots \rangle$$

where the signal values are functions on numbers.

4.1 Adaptive Processes

We introduce the concept of *adaptive process*, which can be used to model adaptivity in different ways. Adaptivity is achieved by an additional signal that is used to change the behaviour of the adaptive process. We divide adaptive processes into the following categories that are illustrated in Figure 7.

Parameter Adaptivity In Figure 7a the functionality of the process $p_{pa} : S(D_1) \times \cdots \times S(D_n) \rightarrow S(E) \rightarrow S(F)$ is changed by the input signal $s_{pa} : S(E)$, which

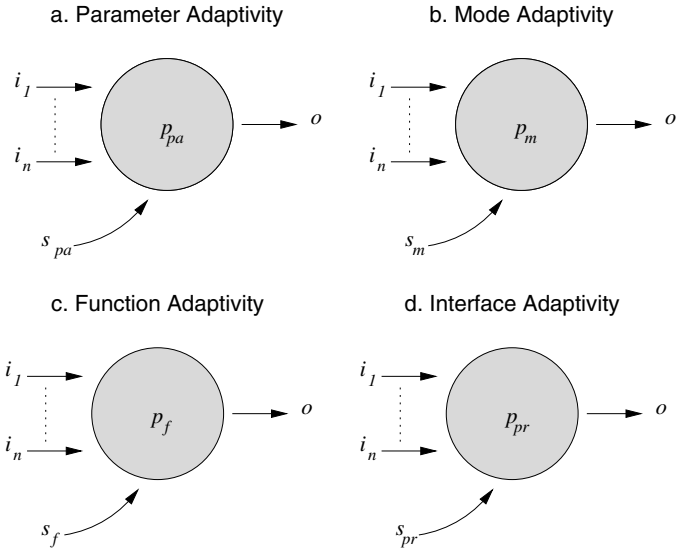


Fig. 7. Four classes of adaptive processes.

supplies current parameters to change the functionality of a system. A typical example is the parameterisation of an analog circuit that executes a transfer function, such as $H(s) = \frac{1}{(s+k_1)(s+k_2)}$. Here the signal s_{pa} is used to adapt the function $H(s)$ by supplying new values for the parameters k_1 and k_2.

Mode Adaptivity In Figure 7b the functionality of the process $p_m : S(D_1) \times \cdots \times S(D_n) \to S(E) \to S(F)$ is changed by the input signal $s_m : S(E)$, which determines the current mode of the system. The adaptive process must contain the different functionalities corresponding to each possible mode. If a new mode is demanded indicated by the value of the signal s_m, the adaptive process has to adapt by selection of the corresponding function. Mode adaptivity is current design practice and is implemented by means of a multiplexer in hardware or an if/case statement in software.

Function Adaptivity Figure 7c uses a signal $s_f : S(D_1 \times \cdots \times D_n \to E)$, where the values of the signal are functions. The adaptive process $p_f : S(D_1) \times \cdots \times S(D_n) \times S(D_1 \times \cdots \times D_n \to E) \to S(E)$ executes always the current value, i.e. a function, of the signal s_f. This means that the adaptive process does not need to store functions, since they are supplied from the outside. The concept of figure 7c can be extended in such a way that the signal s_f does not only carry functions, but also state information. Function adaptivity can be implemented by reconfigurable hardware or software, where the new functions can be loaded into a reconfigurable area, such as an FPGA or memory block, during operation. We will exemplify function adaptivity in Section 4.2.1.

Interface Adaptivity Figure 7d uses a signal $s_{pr} : S(S(D_1) \times \cdots \times S(D_n) \to S(E))$, where the values of the signal are processes. The adaptive process $p_{pr} : S(D_1) \times \cdots \times S(D_n) \times S(S(D_1) \times \cdots \times S(D_n) \to S(E)) \to S(E)$ executes always the current value, i.e. a process, of the process signal s_{pr}. This extends the

concept of Figure 7c, since processes do not only include the function, but also the process constructor. It is therefore possible to not only change functionality, but also the complete interface given by the process constructor. The adaptive process can thus change from a process p_1 of a particular computational model A to another process p_2 of another computational model B. Interface adaptivity can be realised with the same implementation techniques as function adaptivity, but is considerably more complex, since the process interface is changing. We exemplify interface adaptivity in Section 4.2.2.

Using the classification introduced by McKinley in [16] parameter and mode adaptive processes belong to their category of parameter adaptation, while function and interface adaptive processes belong to compositional adaptation. We could even refine our classification further. Especially the class of interface adaptive processes could be divided into several subcategories, since there is a big difference, if only the number of input or output signals is changed, or if there is a change of the model of computation.

All four concepts of adaptivity can be modelled with existing ForSyDe process constructors $combXX_n$ or $stateXX_n$, where "XX" is replaced by the suffix for the computational model of the process.

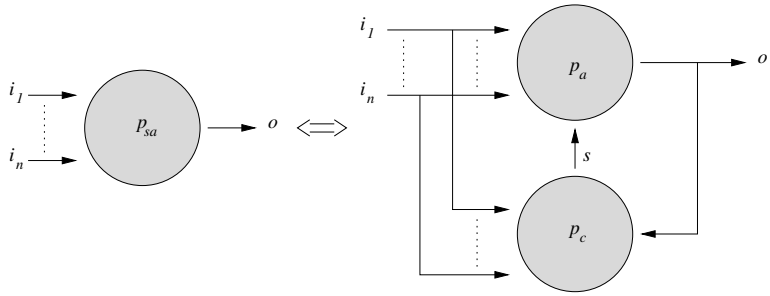
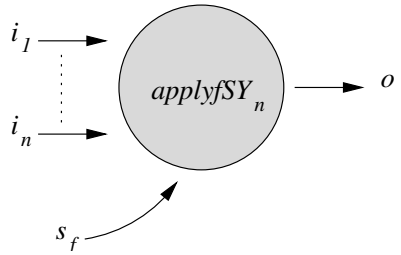


Fig. 8. A self-adaptive process is modelled as a process network of an adaptive process and an additional process. The signal s can either carry modes, functions or entire processes as illustrated in Figure 7.

A special case is the *self-adaptive process* shown in Figure 8, 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 an adaptive process p_a and another process p_c that controls the functionality of the adaptive process p_a. The process p_a can be of any of the four forms given in Figure 7. 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 time instant as the input or output values that trigger the change of the functionality of the adaptive process.

In order to show that adaptivity can be treated as a first-class citizen in ForSyDe, we illustrate how the existing synchronous ForSyDe process constructor $combSY_n$ can be used to model a synchronous function adaptive process. First we define the adaptive process $applyfSY_n$ which models a synchronous version of the function adaptive process p_f as shown in Figure 7c. The definition of the adaptive process $applyfSY_n$ is given in Figure 9.



$$p_f = applyfSY_n$$

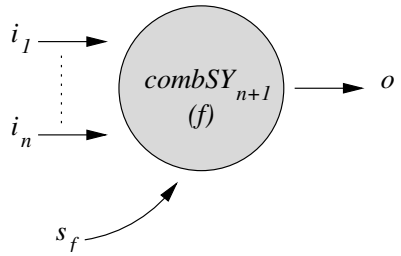
where

$$o = p_f(s_f, (i_1, \ldots, i_n))$$

$$o[k] = s_f[k](i_1[k], i_2[k], \ldots, i_n[k])$$

Fig. 9. The process $applyfSY_n$ models an adaptive process, where adaptivity is controlled by an input signal carrying functions.

Figure 10 shows how the process constructor $applyfSY_n$ can be created be means of the process constructor $combSY_{n+1}$. The argument to the process constructor $combSY_{n+1}$ is a higher-order function f that takes in each event cycle the current value of the signal s_f, the function $s_f[k]$, and applies it to the input values of the other input signals.



$$p_f = combSY_{n+1}(f)$$

where

$$o = p_f(s_f, (i_1, \ldots, i_n))$$

$$o[k] = f(s_f[k], i_1[k], i_2[k], \ldots, i_n[k])$$

$$f(s_f[k], i_1[k], i_2[k], \ldots, i_n[k]) = s_f[k](i_1[k], i_2[k], \ldots, i_n[k])$$

Fig. 10. The process $applyfSY_n$ can be created be means of the process constructor $combSY_{n+1}$.

4.2 Case Studies

4.2.1 Function Adaptivity

Figure 11 shows a tutorial example for function adaptivity using a synchronous system model with two function adaptive processes. A signal is encoded with an encoding function and later the encoded signal is decoded with a decoding function. The signal Key is an input to both the generateEncoder and generateDecoder processes. The processes Encoder and Decoder are examples for function adaptive processes and have signals carrying functions as inputs. Figure 11 models adaptivity at a very abstract level, where the adaptation of the adaptive process is assumed to be instantaneous and does not consume any time.

During design refinement the ideal property of an instantaneous adaptation process will be replaced by an adaptation process with a finite adaptation time. Figure 12 shows a refined model of Figure 11, where the adaptation time is expressed as a number of cycles in the synchronous model of computation. Since this example shall only illustrate how a non-instantaneous adaptation affects the system model,

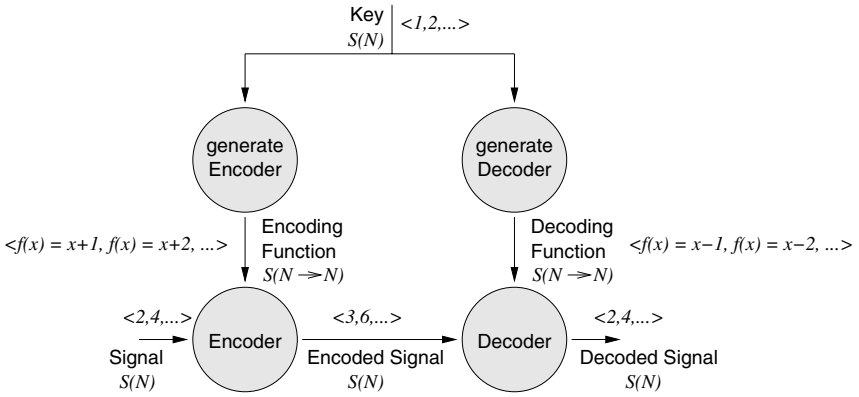


Fig. 11. The Encoder/Decoder is a typical example for function adaptivity. The processes Encoder and Decoder are both function adaptive processes and are fed with signals carrying functions. The types of all signals are shown in the figure.

a very simple model is used. Here it is assumed that the adaptation process takes a fixed amount of time independent of the function that is loaded. In reality these times may vary or even be unknown, which requires a more elaborate modelling approach.

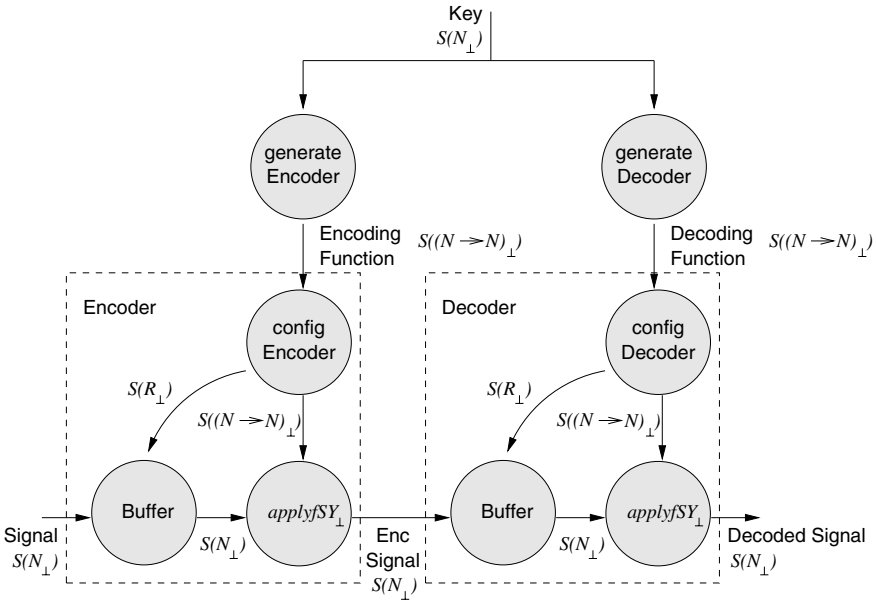


Fig. 12. The refined model of the Encoder/Decoder example takes adaptation time into account. Buffers are introduced and signal types are extended to contain the absent value, since the adaptive process cannot process any data during adaptation. The types of all signals are shown in the figure.

During adaptation it is assumed that the adaptive process cannot produce any meaningful result, which is modelled with the absent value $\perp$. The occurrence of absent events requires the introduction of a buffer that stores the values of the input signal during adaptation. The processes configEncoder and configDecoder control the adaptation process and request new values from the buffers only when the adaptive

process $applyfSY_\perp$ [7] is fed with a valid function.

4.2.2 Interface Adaptivity

The example of Figure 13 shows how processes with different interfaces can be implemented within the same adaptive process. In this case the adaptive process acts as a shared resource.

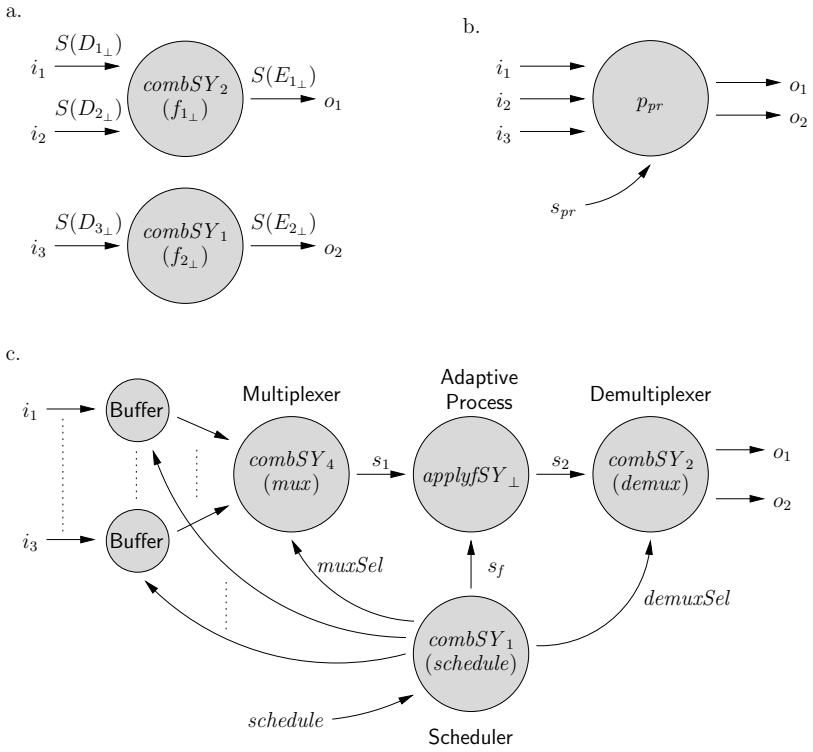


Fig. 13. Figure 13 illustrates the use of an interfaces adaptive process for resource sharing. The processes $combSY_2(f_{1_\perp})$ and $combSY_1(f_{2_\perp})$ shall use the adaptive process as shared resource (Figure 13a). Figure 13b shows an interface adaptive process that is configured by a signal carrying processes as signal values. Figure 13c shows a refined model using a function adaptive process, buffer processes, scheduler, multiplexer and demultiplexer processes.

Figure 13a shows two combinational processes $combSY_2(f_{1_\perp})$ and $combSY_1(f_{2_\perp})$. The processes are independent of each other and do not have valid data at all time instances as indicated by the absent symbol $\perp$ used as index for the functions $f_{1_\perp}$ and $f_{2_\perp}$. If we further assume that the "executions" of $f_{1_\perp}$ and $f_{2_\perp}$ do not overlap, we can introduce an interface adaptive process as shared resource. This is illustrated in Figure 13b, where the adaptive process uses interface adaptivity. Thus the signal s_{pr} carries the processes $combSY_2(f_{1_\perp})$ and $combSY_1(f_{2_\perp})$ as signal values and the adaptive process p_{pr} will always perform the functionality of the current process value given by the signal s_{pr}.

Figure 13c illustrates the next step in design refinement, where the interface adaptive process p_{pr} is replaced by a function adaptive process $p_f = applyfSY_\perp$ and

[7] The process $applyfSY_\perp$ is a variant of the adaptive process $applyfSY$ that can deal with absent values.

additional components such as multiplexer, demultiplexer and scheduler. Buffers have to be introduced, if adaptation time is non-instantaneous. In this solution the signal s_1 has to be able to carry values of both data type $D_{1_\perp} \times D_{2_\perp}$ and $D_{3_\perp}$ at different time instants depending on the functionality of the adaptive process. Thus s_1 has the type $S((D_{1_\perp} \times D_{2_\perp}) + D_{3_\perp})$. The signal s_2 has the type $S(E_{1_\perp} + E_{2_\perp})$. The process Scheduler controls the processes Multiplexer and Demultiplexer and the buffers and schedules when the functions $f_{1_\perp}$ and $f_{2_\perp}$ are executed on the adaptive process.

4.3 Transformation of Adaptive Processes

Although the example of Figure 13 was idealised in the sense that we stayed in the same model of computation and that the execution of functions was assumed to be non-overlapping, it already indicated the possible complexity of adaptive systems. Thus to exploit the full potential of adaptivity, the design process has to support the designer with methods that ensure design correctness.

In the ForSyDe methodology the designer refines a system by the stepwise application of design transformations. The formal concepts developed in ForSyDe for design transformation [20] can also be used for adaptive systems. ForSyDe defines not only *semantic-preserving transformations*, which do not change the semantics of the model, but also *non-semantic transformations*, which change the meaning of a model. While semantic preserving transformations have the nice property that they are correct-by-construction, they are not sufficient to yield an efficient implementation of a system model. Non-semantic preserving transformations are needed to increase the efficiency of the model, e.g. to introduce shared resources or to constrain the size of a buffer. In order to ensure design correctness the ForSyDe project proposes also a verification method for non-semantic preserving transformations [19].

We have already discussed possible design refinements of adaptive system in Section 4.2. Section 4.2.1 showed the refinement of the encoder/decoder system model, where in the initial model adaptation was assumed to be instantaneous (Figure 11). However, if adaptation time is taken into account, buffers have to be introduced into the model and the semantics of the model is changed (Figure 12). The refinements of this example and the example of Figure 13 are typical candidates for introduction into the ForSyDe library as non-semantic preserving design transformations. Each non-semantic design transformation in ForSyDe is accompanied by a formal description of its implication, which informs the designer of the consequences of the transformation and helps to formulate proper verification tasks. This is in contrast to an ad hoc refinement, which is an error-prone activity, since the consequences of the refinement may not be fully understood by the designer.

Also semantic-preserving transformations have their place inside a design flow for adaptive systems. Figure 14 shows an example for a semantic-preserving transformation that merges two function adaptive processes into a single function adaptive process, which can lead to lower design costs. Semantic preserving design transformations are not accompanied by implications since they are correct by construction.

In this article we have only indicated potential design transformations for adap-

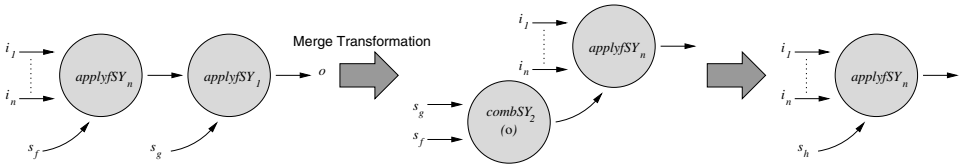


Fig. 14. Two adaptive processes can be merged by means of a design transformation, where $s_h[i] = s_g[i] \circ s_f[i]$.

tive systems. In order to further develop the design refinements of Section 4.2 the formal framework in form of implications and possible verification techniques has to be developed. For more information about these activities we refer to [20] and [19].

5 Conclusion

Since adaptivity adds another dimension of complexity to the design process, design methodologies have to give the designer additional support to ensure the correct of the system during the course of adaptation. We have presented how adaptivity can be treated as first-class citizen in the transformational system design methodology ForSyDe. A main concept is the adaptive process, which is based on the use of functions as signal values and allows to model different classes of adaptive processes. Since adaptivity is fully embedded into the ForSyDe framework, the formal concepts that have been developed for ForSyDe can also be used for adaptive systems. Especially important is the possibility to define semantic and non-semantic preserving design transformations for adaptive systems.

We have focused our work on systems with a static number of processes. Systems where processes are created and deleted dynamically have been so far beyond the scope of our research. Future work will analyse the course of adaptation in more detail and develop design transformations for typical patterns in the design of adaptive systems. In order to be able to verify the system during the course of adaptation, a transformation should not only cover the initial and the transformed process network, but also visualise the intermediate steps of a transformation. Particularly challenging is the development of transformations for the class of interface adaptive processes, since it is not obvious how to design a correctly working system during complex cases of interface adaptation, like a change of the model of computation.

We will continue our work on adaptive systems inside the European project ANDRES [1], where we are extending the ForSyDe modelling framework with a continuous time model of computation and where ForSyDe is used to provide a formal base for the design of adaptive systems in SystemC-based methodologies like SystemC-AMS [23], HetSC [11] and OSSS+R [21].

References

[1] *ANDRES (Analysis and Design of run-time Reconfigurable, Heterogeneous Systems) Project Description*, http://andres.offis.de.

[2] *The ForSyDe webpage*, http://www.imit.kth.se/info/FOFU/ForSyDe/.

[3] Becker, J., M. Hübner, G. Hettich, R. Constapel, J. Eisenmann and J. Luka, *Dynamic and partial FPGA exploitation*, Proceedings of the IEEE **95** (2007), pp. 438–452.

[4] Benveniste, A. and G. Berry, *The synchronous approach to reactive and real-time systems*, Proceedings of the IEEE **79** (1991), pp. 1270–1282.

[5] Benveniste, A., P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic and R. D. Simone, *The synchronous languages 12 years later*, Proceedings of the IEEE **91** (2003), pp. 64–83.

[6] Brown, G., H. C. Cheng, H. Goldsby and J. Zhang, *Goal-oriented specification of adaptation requirements engineering in adaptive systems*, in: *Proceedings of the Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'06)*, Shanghai, China, 2006.

[7] Broy, M., I. H. Krüger, A. Pretschner and C. Salzmann, *Engineering automotive software*, Proceedings of the IEEE **95** (2007), pp. 356–373.

[8] Buell, D., T. El-Ghazawi, K. Gaj and V. Kindratenko, *High-performance reconfigurable computing*, IEEE Computer **40** (2007), pp. 23–27.

[9] Compton, K. and S. Hauck, *Reconfigurable computing: A survey of systems and software*, ACM Computing Surveys **34** (2002), pp. 171–210.

[10] Eker, J., J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs and Y. Xiong, *Taming heterogeneity—the Ptolemy approach*, Proceedings of the IEEE **91** (2003), pp. 127–144.

[11] Herrera, F. and E. Villar, *A framework for embedded system specification under different models of computation in systemc*, in: *Proceedings of the 43rd Design Automation Conference (DAC 2006)*, San Diego, California, USA, 2006.

[12] Jantsch, A., "Modeling Embedded Systems and SoCs," Morgan Kaufmann, 2004.

[13] Jantsch, A., *Models of embedded computation*, in: R. Zurawski, editor, *Embedded Systems Handbook*, CRC Press, 2005 Invited contribution.

[14] Jones, S. P., editor, "Haskell 98 Language and Libraries: The Revised Report," Cambridge University Press, 2003.

[15] Lu, Z., I. Sander and A. Jantsch, *A case study of hardware and software synthesis in ForSyDe*, in: *Proceedings of the 15th International Symposium on System Synthesis*, Kyoto, Japan, 2002, pp. 86–91.

[16] McKinley, P. K., S. M. Sadjadi, E. P. Kasten and B. H. Cheng, *Composing adaptive software*, IEEE Computer **37** (2004), pp. 56–64.

[17] Naji, H. R., B. E. Wells and L. Etzkorn, *Creating an adpative embedded system by applying multi-agent techniques to reconfigurable hardware*, Future Generation Computer Systems **20** (2004), pp. 1055–1081.

[18] Neuendorffer, S. and E. Lee, *Hierarchical reconfiguration of dataflow models*, in: *Conference on Formal Methods and Models for Codesign (MEMOCODE)*, San Diego, 2004.

[19] Raudvere, T., I. Sander, A. K. Singh and A. Jantsch, *Verification of design decisions in ForSyDe*, in: *Proceedings of the 1st International Conference on Hardware - Software Codesign and System Synthesis (CODES+ISSS)*, Newport Beach, California, USA, 2003.

[20] Sander, I. and A. Jantsch, *System modeling and transformational design refinement in ForSyDe*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **23** (2004), pp. 17–32.

[21] Schallenberg, A., W. Nebel and F. Oppenheimer, *OSSS+R modelling and simulating self-reconfigurable systems*, in: *Proceedings of the 16th International Conference on Field Programmable Logic and Applications (FPL 2006)*, Madrid, Spain, 2006.

[22] Schneider, K., T. Schuele and M. Trapp, *Verifying the adaptation behavior of embedded systems*, in: *Proceedings of the Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'06)*, Shanghai, China, 2006.

[23] Vachoux, A., C. Grimm and K. Einwich, *Analog and mixed signal modelling with SystemC-AMS*, in: *Proceedings of the International Symposium on Circuits and Systems (ISCAS 2003)*, Bangkok, Thailand, 2003.

[24] Zhang, J. and B. H. Cheng, *Model-based development of dynamically adaptive software*, in: *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*, Shanghai, China, 2006.