Development and application of design transformations in ForSyDe

I. Sander, A. Jantsch and Z. Lu

Abstract: The formal system design (ForSyDe) methodology has been developed for system level design. Starting with a formal specification model, which captures the functionality of the system at a high level of abstraction, it provides formal design transformation methods for a transparent refinement process of the specification model into an implementation model which is optimised for synthesis. The formal treatment of transformational design refinement is the central contribution of this article. Using the formal semantics of ForSyDe processes we introduce the term characteristic function to be able to define and classify transformations as either semantic preserving or design decision. We also illustrate how we can incorporate classical synthesis techniques that have traditionally been used with control/data-flow graphs as ForSyDe transformations. This approach avoids discontinuities as it moves design refinement into the domain of the specification model.

1 Introduction

Keutzer *et al.* [1] point out that 'to be effective a design methodology that addresses complex systems must start at high levels of abstraction' and underline that an 'essential component of a new system design paradigm is the orthogonalisation of concerns, i.e. the separation of various aspects of design to allow more effective exploration of alternative solutions'. In particular, a design methodology should separate (1) function (what the system is supposed to do) from architecture (how it does it) and (2) communication from computation. The authors of that work 'promote 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 believe that 'the most important point for functional specification is the underlying mathematical model of computation'. These arguments strongly support the Formal System Design (ForSyDe) methodology, which addresses the transformational design of embedded system applications. The objective of ForSyDe is to move system design to a higher level of abstraction.

ForSyDe starts with a formal specification model which captures the functionality of the system at a high abstraction level. While the high level of abstraction is suitable to system level specifications, there is a gap between the specification model and a possible implementation in a complex system architecture. ForSyDe tries to bridge this gap using the concept of process constructors. Although a system model is formulated as a function, the use of process constructors implies that the functional model can be interpreted as a network of synchronously communicating concurrent processes. Such a process structure is almost fixed in other design languages (VHDL, SDL), but in ForSyDe processes can be merged and split during the application of transformation rules during the design transformation phase [2]. As each process constructor has a hardware and software interpretation, the refined implementation model can be interpreted into a structure with hardware and software components.

In contrast to [3], where the significant potential of our approach was illustrated by two powerful transformations for clock domain and communication refinement, the emphasis of this article is on the formal treatment of design transformations in ForSyDe. Here we introduce the formal basis, define a format for transformations in ForSyDe and show how the large amount of work that exists for high-level synthesis can be used for the development of design transformations.

2 Related work

The ForSyDe system model is based on the perfect synchrony hypothesis, which also forms the basis for the family of synchronous languages. According to Benveniste and Berry 'the synchronous approach is based on a relatively small variety of concepts and methods based on deep, elegant, but simple mathematical principles' [4]. The basic synchronous assumption is that the outputs of the system are synchronised with the system inputs, while the reaction of the system takes no observable time. The synchronous assumption implies a total order of events and leads to a clean separation between computation and communication. A global clock triggers computations which are conceptually simultaneous and instantaneous. This assumption frees the designer from the modelling of complex communication mechanisms. These properties give a solid base for formal methods.

Data flow models [5], such as Kahn process networks [6] or SDF [7], are well-suited to applications which do not require the expression of time, such as DSP applications. Even though they excellently fit their application domain, we have chosen a synchronous model to enable the

IEE Proceedings online no. 20030836

doi: 10.1049/ip-cdt:20030836

Paper received 9th May 2003

The authors are with the Royal Institute of Technology, Department of Microelectronics and Information Technology, Laboratory of Electronics and Computer Systems, Isafjordsgatan 39, SE-164 40, Stockholm, Sweden

expression of timing properties and constraints on a level that abstracts from physical time.

The family of synchronous languages can be divided into two groups, one group targeting data flow applications (e.g. Lustre [8] and Signal [9]), the other targeting control oriented applications (e.g. Statecharts [10], Esterel [11] and Argos [12]). There is, however, no synchronous language covering both application domains [4]. The clean mathematical formalism has led to the development of several verification tools for these languages. The research in [13] gives an overview of the techniques and tools developed for the validation of reactive systems described in Lustre. However, the authors point out that these techniques can be adapted to any synchronous language. ForSyDe is based on the same foundation as the synchronous languages (the perfect synchrony hypothesis) but extends it to cover both control and data flow applications.

While we advocate the use of a single unified system model in the ForSyDe methodology, much work is done using mixed models of computation. This approach is advantageous in that a suitable model of computation can be used for each part of the system. As the system model is based on several computational models, however, the semantics of the interaction of fundamentally different models has to be defined, which is not a simple task. This also amplifies the verification problem, because the system model is not based on a single semantics. There is little hope that formal verification techniques can help and we are left with simulation as the only means of validation. In addition, once a heterogeneous system model is specified, it is very difficult to optimise systems between different models of computation. In summary, cross domain verification and optimisation will remain elusive for many years with respect to any heterogeneous modelling approach.

In $*$charts [14] hierarchical finite state machines (FSMs) are embedded within a variety of concurrent models of computations. The idea is to decouple the concurrency model from the hierarchical FSM semantics. An advantage is that modular components (e.g. basic FSMs) can be designed separately and composed into a system with the model of computation that best fits the application domain. It is also possible to express a state in an FSM by a process network of a specific model of computation. $*$charts has been used to describe hierarchical FSMs which are composed using data flow, discrete event and synchronous models of computation.

The Ptolemy project [15] 'studies heterogeneous modelling, simulation, and design of concurrent systems'. It is implemented in the Ptolemy II software environment [16] which provides support for 'hierarchically combining a large variety of models of computation and allows hierarchical nesting of the models'.

Internal representations like the SPI model [17] and FunState [18] have been developed to integrate a heterogeneous system model into one internal representation. The SPI model 'shall enable the analysis and synthesis of mixed reactive/transformative systems described in several languages with possible differences in the underlying models of computation. All information relevant to synthesis is abstracted from the input languages and transformed into the semantics of the SPI model'.

The parallel programming community has used functional languages to derive parallel programs from a functional specification [19]. Skeletons are used to structure a problem. This formulation is then transformed, using cost measures, into an efficient implementation for a chosen computer architecture. Reekie [20] has used Haskell to model digital signal processing applications. Similarly to ourselves, he modelled streams as infinite lists and used higher-order functions to operate on them. Finally, semantic-preserving methods were applied to transform a model into a more efficient representation. Ruby [21] is a relational language that has mainly been used for hardware design. In [22] a declarative framework for hardware/software codesign based on Ruby has been proposed. Ruby also supports transformations based on equational reasoning and supports data type refinement. Lava [23] is a hardware description language based on Haskell. It focuses on the structural representation of hardware and offers a variety of powerful connection patterns. Lava descriptions can be translated into VHDL and there exist interfaces to formal method tools. Recently, Singh [24] has proposed the use of Lava for system level specifications. Hardware ML (HML) [25] is based on the functional language Standard ML and mainly an improvement of VHDL – there is a direct mapping from HML constructs into the corresponding VHDL constructs. Mycroft and Sharp have used the functional languages SAFL and SAFL+ mainly for hardware design but extended their approach in [26] to hardware/software codesign. They transform SAFL programs through meaning preserving transformations and compile the resulting program in a resource-aware manner, i.e. a function which is called more than once will be a shared resource.

A good overview of program transformation in general is given in [27] and of transformation of functional and logical programs in [28]. One of the best known transformation systems is the CIP (computer-aided, intuition-guided programming) project [29]. Inside CIP, program development is viewed as an evolutionary process which usually starts with a formal problem specification and ends with an executable program for the intended target machine. The individual transformation use semantic preserving transformation rules, which guarantees that the final version of the program still satisfies the initial specification. Such an approach has the following advantages [29]:

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

Within the CIP framework Kloos and Dosch [30] have used correctness-preserving transformations to derive different kinds of adder structures from an initial specification. In order to allow for a successful transformation of a specification into an effective implementation, a transformation framework has to provide a sufficient number of transformation rules, and there must also exist a transformation strategy to choose a suitable order of transformation rules. Ideally, this strategy interacts with an estimation tool that indicates whether one implementation is more efficient than another. Since program transformation requires a well-developed framework, it has thus far been mainly used for small programs or modules inside a larger program, where software correctness is critical.

Most of the transformational approaches are concerned with software programs where concepts of synchronous sub-domains and resource sharing, as discussed in this paper, have no relevance. Our approach allows also to use the large amount of research in high-level synthesis [31–32] by defining design decision transformations for refinement techniques like re-timing or resource sharing.

Voeten points out that each transformational design that is based on a general purpose language will suffer from fundamental incompleteness problems [33]. This means that the initial model determines to a large extent whether an effective and satisfying implementation can or cannot be obtained, since only a limited part of the design space can be explored. The same problem is known in the context of high-level synthesis as the syntactic variance problem [34], which in general is unsolvable.

3 The ForSyDe methodology

3.1 Design process

The ForSyDe design process (Fig. 1) starts with the development of a formal, abstract, functional specification model which can be executed using the functional language Haskell. This model is then refined inside the functional domain by a stepwise application of well-defined design transformations to an optimised and detailed implementation model. Since the implementation model is a refined version of the specification model, the same validation and verification methods can be applied to both models.

The next phase in the design process is implementation mapping, where the implementation model is partitioned and mapped onto the components of the target architecture. Only at this late stage of the design process does ForSyDe leave the functional domain and enter the implementation domain since the design is now described with 'implementation languages'. So far, ForSyDe defines a mapping to hardware (VHDL) and sequential software (C) as elaborated in [35].

3.2 Specification model

The specification model is based upon a synchronous computational model and uses ideal data types such as real numbers and infinite buffers. It abstracts from implementation details, such as low-level communication mechanisms, and enables the designer to focus on the functional behaviour of the system rather than structure and architecture. The specification model leaves a wide design space for further design exploration and design refinement, which is supported by our transformational refinement techniques (Section 4).

In order to formally describe our computational model, we follow the denotational framework of Lee and Sangiovanni-Vincentelli [36]. They define signals as a set of events, where each event e has a tag t and a value v, i.e. $e = (t, v) \in T \times V$. As our specification model is synchronous, T is the set of natural numbers, and all signals have the same set of tags. In order to model the absence of a

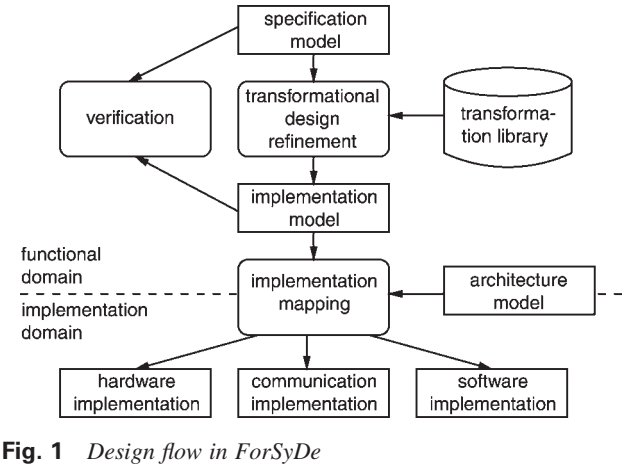

Fig. 1 *Design flow in ForSyDe*

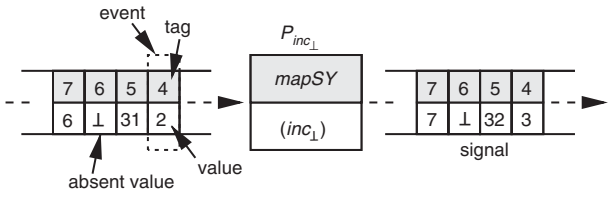

Fig. 2 *Modelling of signals and processes*

value at a certain tag, a data type D can be extended into a data type $D_\perp$ by adding the special value $\perp$ ('bottom'). Absent values are used to establish a total order of events when dealing with aperiodic signals or signals with different event rates.

Figure 2 illustrates the modelling of signals and the behaviour of processes. During the event cycle n a process processes the events of each signal with the tag n and outputs the result with the same tag n. A signal s is defined as a set of events, where each event e_i has a tag i and a value e_i. For an indexed signal s_k we denote an event as $(e_k)_i$ where k is the number of the signal and i is the tag of the event.

$$s = \langle e_0, e_1, \ldots \rangle$$
$$s_k = \langle (e_k)_0, (e_k)_1, \ldots \rangle$$

Parallel signals are described as a tuple of signals. A process P maps m input signals on n output signals.

$$P(s_1, s_2, \ldots, s_m) = (s'_1, s'_2, \ldots, s'_n) \quad m, n \in \mathbb{N}$$

Processes are executed concurrently and communicate with each other synchronously by means of signals. Process networks can be modelled as sets of equations.

As we employ the perfect synchrony hypothesis [4], all input and output signals have the same set of tags. We implement the synchronous computational model with the concept of process constructors. A process constructor is a higher-order function that takes combinational functions, i.e. functions that have no internal state, and values as input and produces a process as output. The ForSyDe methodology obliges the designer to use process constructors for the modelling of processes. This leads to a well-defined specification model where all processes are constructed by process constructors. There is a clean separation between synchronisation (process constructors) and computation (combinational function). In addition, each process constructor has a structural hardware and software semantics, which is used to translate the implementation model into a hardware/software implementation [35].

The process constructor *mapSY* takes a combinational function f and constructs a process with one input and one output signal, where f is applied on all values of the input signal. The process $P_{inc_\perp}$ of Fig. 2 is constructed with *mapSY*.

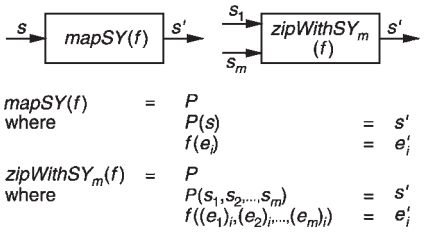

Fig. 3 *Combinational process constructors mapSY and zipWithSY*

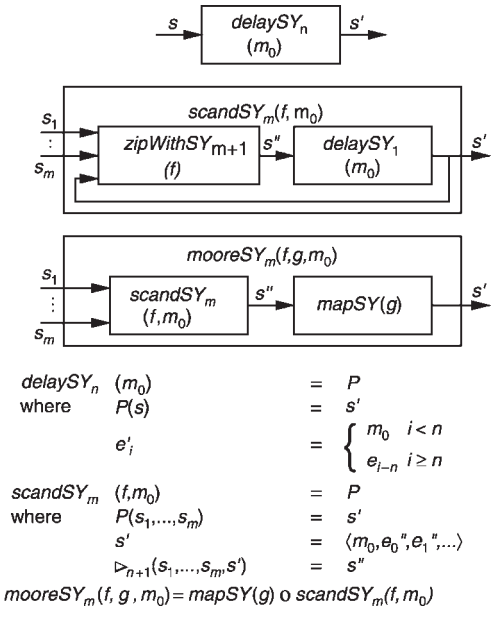

Fig. 4 *Sequential process constructors $delaySY_n$, $scandSY_m$ and $mooreSY_m$*

$$P_{inc_\perp} = mapSY(inc_\perp)$$

where

$$\mathrm{inc}_\perp(x) = \begin{cases} \perp & \text{if } x = \perp \\ x+1 & \text{otherwise} \end{cases}$$

Figure 2 also illustrates the separation of synchronisation (grey shaded and performed by *mapSY*) and computation (white shaded and performed by the function $inc_\perp$).

The process constructor $zipWithSY_m$ corresponds to *mapSY*, but creates processes with multiple input signals. The short notation $*$ is used for *mapSY* and $\triangleright_m$ for $zipWithSY_m$. Both process constructors are defined in Fig. 3.

The basic sequential process constructor $delaySY_n$ (short notation Δ_n) constructs a process which delays a signal by n cycles. We define the m-input process constructor $scandSY_m$ which takes a function f and a value m_0 for the initial state to construct the basic FSM process. Using the function composition operator $\circ$, where

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

we can define *mooreSY* to model a Moore FSM. The discussed sequential process constructors are defined in Fig. 4.

3.3 Implementation model

The implementation model is the result of the refinement process (Section 4). In contrast to the specification model, which is a network of concurrent synchronous processes, it may also include domain interfaces in order to establish

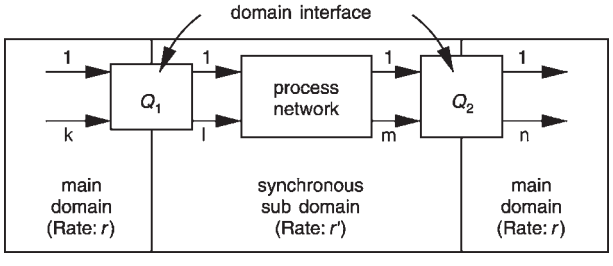

Fig. 5 *Synchronous sub-domains*

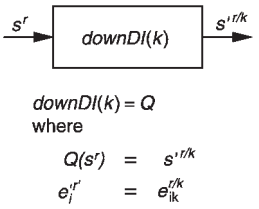

Fig. 6 *Domain interface downDI(k)*

synchronous sub-domains which comprise local synchronous process networks with signals having different events rates as illustrated in Fig. 5.

In order to formally describe implementation models with synchronous sub-domains we extend our notation for signals by including the event rate $r \in \mathbb{Q}$ to the form $s^r = \langle e_0^r, e_1^r, \ldots \rangle$ where the tag of a signal of e_i^r is given by the position i and the event rate r and has the absolute value i/r. A domain interface consumes m input signals with event rate r and produces n output signals with another event rate r'. In the specification model the event rate of all signals is one.

Figure 6 shows the domain interface *downDI*(k) which consumes an input signal s with event rate r and produces an output signal s' that includes only each k-th input event (starting with $k = 0$) and has the event rate $r' = r/k$. Synchronous sub-domains violate the synchronous assumption, as not all signals share the same set of tags. Thus, they are not allowed in the specification model, but are introduced by well-defined transformations during the refinement process. Inside a synchronous sub-domain the synchronous assumption is still valid and the same formal techniques can be used as for the specification model. Due to the formal definition of domain interfaces we may also reason about a refined model with synchronous sub-domains as further elaborated in the following Section.

4 Refinement

One central idea of the ForSyDe methodology is to move large parts of the synthesis, which traditionally is part of the implementation domain, into the functional domain. This is done in the refinement phase where the specification model M_0 is stepwise refined by well-defined design transformations T_i into a final implementation model M_n (Fig. 7). Only at this late stage is the implementation model translated using the ForSyDe hardware and software semantics into a synthesisable implementation description.

A transformation (Definition 2) is the application of a transformation rule (Definition 1) to a process network which is part of a system model as illustrated in Fig. 8.

Definition 1 (transformation rule): A transformation rule is a functional mapping of a process network PN onto another process network PN' with the same input signals and the same number of output signals. A transformation rule is denoted by $R(PN) = PN'$ or $PN \xrightarrow{R} PN'$.

Definition 2 (transformation): A transformation $T(M, PN, R)$ is a functional mapping of a system model M onto another system model M' with the same input signals and the same number of output signals. Using the transformation rule R

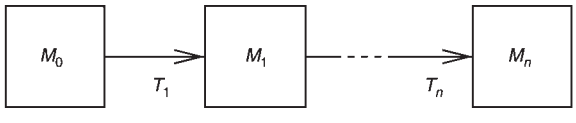

Fig. 7 *Transformational refinement*

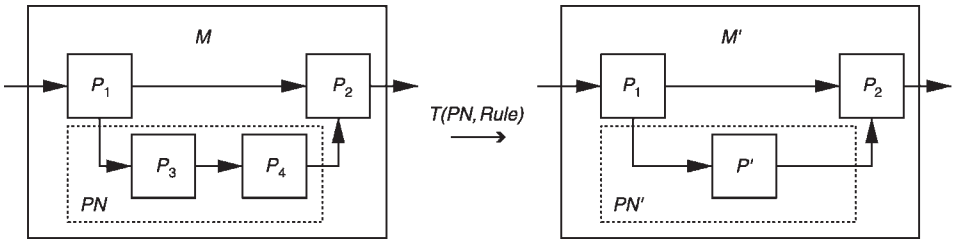

Fig. 8 *Design transformation*

the internal process network PN in M is replaced by $R(PN)$ to yield M'. A transformation is denoted by $T(M, PN, R) = M' = M[R(PN)/PN]$ or $M \xrightarrow{T(PN,R)} M' = M[R(PN)/PN]$, where $[x/y]$ reads as y is replaced by x.

In order to classify transformations and to compare process networks we introduce the term 'characteristic function' which characterises the functional behaviour of a process network.

Definition 3 (characteristic function): The characteristic function

$$\mathcal{F}_{PN}((s_1)^r, \ldots, (s_m)^r, i)$$

of a process network PN with the input signals $(s_1)^r, \ldots, (s_m)^r$ and the output signals $(s'_1)^{u_1}, \ldots, (s'_n)^{u_n}$ expresses the dependence of the output events at index i on the input signals.

$$\mathcal{F}_{PN}(s_1^r, \ldots, s_m^r, i) = ((e'_1)_i^{u_1}, \ldots, (e'_n)_i^{u_n})$$

The characteristic function can be derived for any process network including domain interfaces. Processes based only on combinational process constructors have characteristic functions which only depend on current input events. Here, we give the characteristic function for the basic combinational processes *mapSY* and $zipWithSY_m$.

$$(e')_i^u = \mathcal{F}_{*(f)}(s^r, i) = (f(e_i))_i^r$$
$$(e')_i^u = \mathcal{F}_{\triangleright_m(f)}((s_1)^r, \ldots, (s_m)^r, i) = (f(e_1)_i, \ldots, (e_m)_i)_i^r$$

Sequential processes have a characteristic function which also depends on past input values. A process constructed with $delaySY_n$ has the following characteristic function.

$$(e')_i^u = \mathcal{F}_{\Delta_n(m_0)}(s^r, i) = \begin{cases} (m_0)_i^r & i < n \\ e_{i-n}^r & i \geq n \end{cases}$$

The characteristic functions for FSM processes like $mooreSY_m$ are more complex as they depend on past values and include an internal feedback loop. The characteristic function on $mooreSY_m$ is recursively given as

$$(e')_i^u = \mathcal{F}_{mooreSY_m(f,g,m_0)}((s_0)^r, \ldots, (s_m)^r, i) = (g(m_i))_i^r$$

where

$$m_i = \begin{cases} m_0 & \text{if } i = 0 \\ f((e_1)_{i-1}^r, \ldots, (e_n)_{i-1}^r, m_{i-1}) & \text{otherwise} \end{cases}$$

We can classify transformations as *semantic preserving* or *design decision* according to the following definitions.

Definition 4 (semantic preserving transformation): A transformation rule

$$PN \xrightarrow{R} PN'$$

is semantic preserving, if and only if

$$\forall i \in \mathbb{N}_0 . \mathcal{F}_{PN}(s_1, \ldots, s_m, i) = \mathcal{F}_{PN'}(s_1, \ldots, s_m, i)$$

Definition 5 (design decision): A transformation rule

$$PN \xrightarrow{R} PN'$$

is a design decision, if and only if

$$\exists i \in \mathbb{N}_0 . \mathcal{F}_{PN}(s_1, \ldots, s_m, i) \neq \mathcal{F}_{PN'}(s_1, \ldots, s_m, i)$$

Semantic preserving transformation do not change the meaning of the model and are mainly used to optimise the model for synthesis. In contrast, design decisions change the meaning of a model and are mainly used to introduce the necessary low-level details required to yield an efficient implementation. Definition 5 gives only the formal definition of a design decision. For a meaningful design decision it is also necessary that the transformed process network PN' and the original process network PN are closely related. A typical design decision is the refinement of an infinite buffer into a fixed-size buffer with n elements. While such a design decision clearly modifies the semantics, the transformed model may still behave in the same way as the original model. For instance, if it is possible to prove that a certain buffer will never contain more than n elements, the ideal buffer can be replaced by a finite one of size n.

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

We exemplify transformation rules by a combinational process with n inputs. If the process has a regular structure such as an N-input adder or multiplier, where $N = 4, 8, 16, \ldots$ the process can be transformed into a balanced network of $N - 1$ 2-input processes. The transformation rule *BalancedTree*(*PN*) is defined in the transformation library as

`Transformation Rule`: *BalancedTree*(*PN*)
`Original Process Network`:

$$PN(s_1, \ldots, s_N) = \triangleright_N(f)(s_1, \ldots, s_N)$$
$$N = 2^k; k > 1$$
$$f(x_1, \ldots, x_N) = x_1 \otimes \cdots \otimes x_N; \otimes \text{ is associative}$$

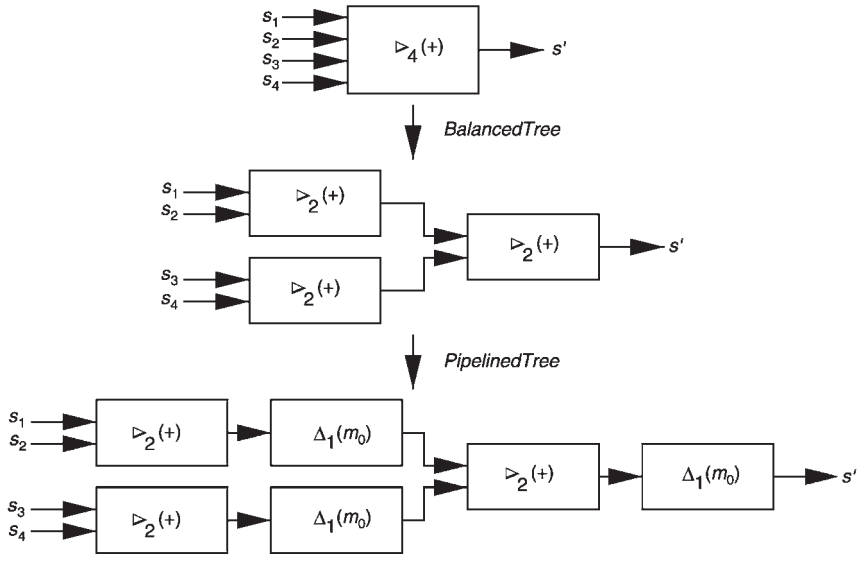

Fig. 9 *Transformation into balanced pipelined structure*

`Transformed Process Network`:

$$PN'(s_1,\ldots,s_N) = \triangleright_2(g)(\ldots(\triangleright_2(g)(s_1,s_2),\triangleright_2(g)(s_3,s_4)),\ \ldots(\triangleright_2(g)(s_{N-3},s_{N-2}),\triangleright_2(g)(s_{N-1},s_N)))$$
$$g(x,y) = x \otimes y$$

`Implication`:

$$\mathcal{F}_{PN}(s_1,\ldots,s_N,i) = \mathcal{F}_{PN'}(s_1,\ldots,s_N,i); \forall i \in \mathbb{N}_0$$

This transformation can be used for all processes that comply with the format and constraints given in `Original Process Network`, here multiple 2^k-input processes, where the operator $\otimes$ is associative. From the `Implication` we can see that *BalancedTree*(*PN*) is semantic preserving, as the characteristic functions of the original and transformed process network are identical.

There is another transformation *PipelinedTree* that pipelines a balanced tree structure of possibly different two-input processes into a pipelined tree structure.

`Transformation Rule`: *PipelinedTree*(*PN*)
`Original Process Network`:

$$PN(s_1,\ldots,s_N) = \triangleright_2(g_{N-1})(\ldots(\triangleright_2(g_1)(s_1,s_2),\ldots),\ \ldots(\triangleright_2(\ldots,\triangleright_2(g_{N/2})(s_{N-1},s_N)))$$
$$N = 2^k; k > 1$$

`Transformed Process Network`:

$$PN'(s_1,\ldots,s_N) = \Delta_1(m_0) \circ \triangleright_2(g_{N-1})\ (\ldots(\Delta_1(m_0) \circ \triangleright_2(g_1)(s_1,s_2),\ldots),\ \ldots(\ldots,\Delta_1(m_0) \circ \triangleright_2(g_{N/2})(s_{N-1},s_N)))$$

`Implication`:

$$\mathcal{F}_{\Delta_k(m_0)\circ PN}(s_1,\ldots,s_N,i) = \mathcal{F}_{PN'}(s_1,\ldots,s_N,i); \forall i \geq k$$

As expressed in the `Implication`, *PipelinedTree* is a design decision, as it introduces a delay of k cycles. Since such implications are part of the transformation rule, the designer is always aware of the consequences of a transformation. During the refinement process s/he chooses transformations from the library and applies them successively as visualised in Figure 9, where a four-input addition process is transformed into a pipelined structure.

A direct translation of a computationally intensive algorithm such as an nth-order FIR filter results in an implementation with a large amount of multipliers and adders. Using the concept of synchronous sub-domains we have developed a transformation *SerialClockDomain* which transforms a combinational process of a regular structure into a structure with two clock domains using an FSM process to schedule the operations into several clock cycles. This transformation (as illustrated in Fig. 10 and formally

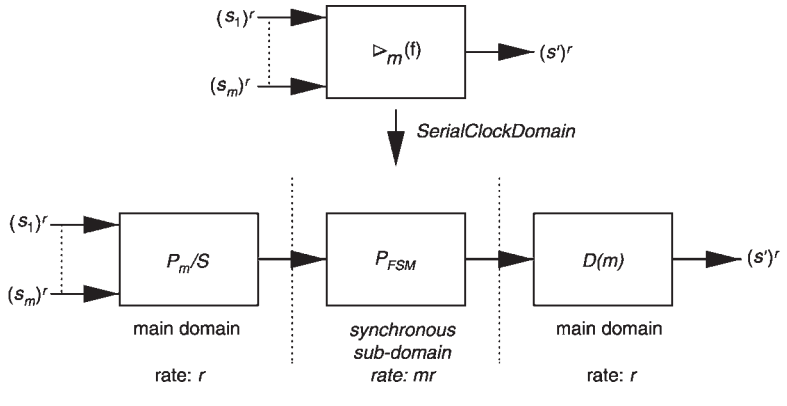

Fig. 10 *Transformation into a serialised form with two clock domains*

given below) is very efficient where if there are identical operations that can be shared.

`Transformation Rule`: *SerialClockDomain*(*PN*)
`Original Process Network`:

$$PN(s_1,\ldots,s_m) = \triangleright_m(f)(s_1,\ldots,s_m)$$
$$f(x_0,\ldots,x_{m-1}) = g_{m-1}(h_{m-1}(x_{m-1}), (\ldots,(g_1(h_1(x_1),h_0(x_0)))\ldots))$$

`Transformed Process Network`:

$$PN'(s_1,\ldots,s_m) = (D(m)\circ P_{FSM}\circ P/S_m)(s_1,\ldots,s_m)$$
$$P_{FSM} = mooreSY(f',g',\nu_0)$$
$$f'(x,(i,\nu)) = \begin{cases}(1,h_i(x))i=0\\(i+1,g_i(h_i(x),\nu))0<i<m-1\\(0,g_i(h_i(x)+\nu))i=m-1\end{cases}$$
$$g'(i,\nu) = \begin{cases}\nu & i=0\\ \perp & 0<i<m\end{cases}$$

`Implication`:

$$\mathcal{F}_{\Delta_1(\nu_0)\circ PN}(s_1,\ldots,s_m,i) = \mathcal{F}_{PN'}(s_1,\ldots,s_m,i)$$

The transformed process network works as follows. During an input event cycle the domain interface P/S_m (parallel to serial) reads all input values at rate r and outputs them at rate mr one by one in the corresponding m output cycles. The process P_{FSM} is based on *mooreSY* and executes the combinational function f of the original process in m cycles. In state zero the first input value (operand x_0) is stored as intermediate value v. In the $n-1$ following states a function g_i is applied to the new input value (x_i) and to the intermediate value. At tag $0, m, 2m, \ldots$ the process produces the output value, otherwise the output has the value $\perp$. The domain interface *downDI*(m) ($D(m)$) down-samples the input signal to rate r and outputs only each mth input value starting with tag zero, thus suppressing the absent values from the output of P_{FSM}.

As domain interfaces can be characterised by a characteristic function, it means that, while not shown here, the characteristic function for the whole transformed process network can be developed. It follows that *SerialClockDomain* delays the output of the transformed process network one event cycle compared to the original process network, which is given as `Implication`.

This transformation can, for example, be applied to the four-input adder of Fig. 9, where $h_i(x)$ is the identity function and $g_i(x,y)$ is an add operation, resulting in a circuit with two clock domains using a single adder.

5 Refinement of FIR-filter

We now use the developed transformation *SerialClockDomain* for the refinement of a FIR-filter which is part of the specification model of a digital equaliser [3]. A FIR-Filter is described by the following equation:

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

We model the FIR-filter as shown in Fig. 11. The process $sipoSY_k$ models a serial-in-parallel-out shift register with $k+1$ output signals. The process $zipWithSY_{k+1}(ipV(c))$ computes the inner product of the coefficient vector, $c_0,\ldots,c_k$, and the outputs of the process $sipoSY_k, x_n,\ldots,x_{n-k}$. Since the process $ipV(c)$ is defined as

$$(ipV(c))(x_0,\ldots,x_n) = c_0x_0+\cdots+c_nx_n$$

it complies with the `Input Process Network` format of the transformation rule *SerialClockDomain*, where

$$g_i(x,y) = x+y$$
$$h_i(x) = c_ix$$

We can use this rule to apply the transformation

$$T(M_1, zipWithSY_{k+1}(ipV(c)), SerialClockDomain)$$

on the FIR-filter model M_1 in order to receive a model M_2, where $sipoSY_k$ remains unchanged and the FIR-filter is realised with two clock domains and only one multiplier, and one adder (Fig. 11).

We have used the ForSyDe hardware semantics to translate both the original model and the transformed model for an eighth-order FIR-filter with sample and coefficient size of 10-bit into VHDL, and synthesised it for the CLA90K standard cell library. The results (for $f = 8$ MHz) show that the area for the transformed model (4030 gates) is as expected, clearly less than for the original model (10482 gates).

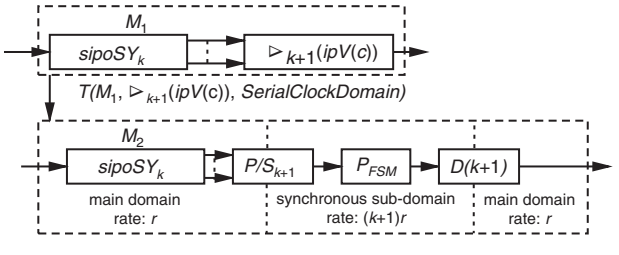

Fig. 11 *Transformation of a FIR-Filter*

6 Conclusions

The focus of this article is the formal basis of transformational refinement in the functional domain in ForSyDe. Using the formal definition of process constructors and domain interfaces we can develop characteristic functions for process networks in order to define transformations which can be classified as either semantic preserving or as design decisions. Each transformation rule is well defined by format and constraints on the original process network and the resulting transformed process. In addition, each rule also shows the consequences for the design through an implication part, expressed with the characteristic function.

As illustrated, powerful synthesis techniques can now be formulated as transformation rules and applied inside the functional domain. This is contrast to traditional methods, where the initial model is first translated into a control/data-flow graph before it is transformed, which leads to discontinuities in the design process. By selecting transformation rules from the transformation library, the designer is now able to perform a transparent and documented refinement process inside the functional domain.

As part of our future work, we plan to analyse how a design decision on a local process network effects the overall system model. Further future work covers the incorporation of formal verification techniques and the development of tool support for transformational design in ForSyDe.

7 Acknowledgment

The research presented in this paper was supported by the Swedish Foundation for Strategic Research (SSF) and is

part of the Integrated Electronic Systems(INTELECT) program.

8 References

1 Keutzer, K., Malik, S., Newton, A.R., Rabaey, J.M., and Sangiovanni–Vincentelli, A.: 'System-level design: Orthogonolization of concerns and platform-based design', *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, 2000, **19**, (12), pp. 1523–1543

2 Wu, W., Sander, I., and Jantsch, A.: 'Transformational system design based on a formal computational model and skeletons'. Presented at the Forum on Design Languages 2000, Tübingen, Germany, September 2000

3 Sander, I., and Jantsch, A.: 'Transformation based communication and clock domain refinement for system design'. Presented at 39th Design automation Conf (DAC), New Orleans, USA, June 2002

4 Benveniste, A., and Berry, G.: 'The synchronous approach to reactive and real-time system', *Proc. IEEE*, 1991, **79**, (9), pp. 1270–1282

5 Lee, E.A., and Parks, T.M.: 'Dataflow process networks', *Proc. IEEE*, 1995, pp. 773–801

6 Kahn, G.: 'The semantics of a simple language for parallel programming'. Presented at the IFIP Congress, North-Holland, 5–10 August 1974

7 Lee, E.A., and Messerschmitt, D.G.: 'Synchronous data flow', *Proc. IEEE*, 1987, **75**, (9), pp. 1235–1245

8 Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D.: 'The synchronous data flow programming language LUSTRE', *Proc. IEEE*, 1991, **79**, (9), pp. 1305–1320

9 Le Guernic, P., Gautier, T., Le Borgne, M., and Le Marie, C.: 'Programming real-time applications with SIGNAL', *Proc. IEEE*, 1991, **79**, (9), pp. 1321–1335

10 Harel, D.: 'Statecharts: a visual formalism for complex systems', *Sci. Comput. Program.*, 1987, **8**, (3), pp. 231–274

11 Boussinot, F., and De Simone, R.: 'The ESTEREL language', *Proc. IEEE*, 1991, **79**, (9), pp. 1293–1304

12 Maraninchi, F.: 'The Agros language: Graphical representation of automata and description of reactive systems'. Presented at IEEE Workshop on Visual languages, Kobe, Japan, October 1991

13 Halbwachs, N. and Raymond, P.: 'Validation of synchronous reactive systems: from formal verification to automatic testing', *Lect. Notes Comput. Sci.*, 1999, **1742**, pp. 1–12

14 Girault, A., Lee, B., and Lee, E.A.: 'Hierarchical finite state machines with multiple concurrency models', *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 1999, **18**, (6), pp. 742–760

15 Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., and Xiong, Y.: 'Taming heterogeneity—the Ptolemy approach', *Proc. IEEE*, 2003, **91**, (1), pp. 127–144

16 Davis II, J., Goel, M., Hylands, C., Kienhuis, B., Lee, E.A., Liu, J., Liu, X., Muliadi, L., Neuendorffer, S., Reekie, J., Smyth, N., Tsay, J., and Xiong, Y.: 'Overview of the Ptolemy project'. Technical Report UCB/ERL No. M99/37, Dept. EECS, University of California, Berkeley, CA 94720, 1999

17 Ernst, R., Ziegenbein, D., Richter, K., Thiele, L., and Teich, J.: 'Hardware/software codesign of embedded systems - the SPI workbench'. Presented at the IEEE Workshop on VLSI, Orlando, USA, 1999

18 Thiele, L., Strehl, K., Ziegenbein, D., Ernst, R., and Teich, J.: 'Funstate - an internal design representation for codesign'. Proc. IEEE/ACM Int. Conf. on Computer-Aided Design (ICCAD), 7–11 November 1999, pp. 558–565

19 Skillicorn, D.B.: 'Foundations of Parallel Programming' (Cambridge University Press, Cambridge, 1994)

20 Reekie, H.J.: 'Realtime Signal Processing'. PhD thesis, University of Technology at Sydney, Australia, 1995

21 Jones, G., and Sheeran, M.: Circuit design in Ruby. Formal Methods for VLSI Design, North-Holland, 1990, pp. 13–70

22 Luk, W., and Wu, T.: 'Towards a declarative framework for hardware - software design'. Proc. 3rd Int. Workshop on Hardware/Software Codesign, Grenoble, France, 22–24 September 1994, pp. 181–188

23 Bjesse, P., Claessen, K., Sheeran, M., and Singh, S.: 'Lava: Hardware design in Haskell'. Presented at the Int. Conf. on Functional programming, Baltimore, MD, 27–29 September 1998

24 Singh, S.: 'System level specification in Lava'. Proc. Design, Automation and Test in Europe Conf. (DATE), Munich, Germany, March 2003, pp. 370–375

25 Li, Y., and Leeser, M.: 'HML, a novel hardware description language and its translation to VHDL', *IEEE Trans VLSI*, 2000, **8**, (1), pp. 1–8

26 Mycroft, A., and Sharp, R.: 'Hardware/software co-design using functional languages', *Lect. Notes Comput. Sci.*, 2000, **2031**, pp. 236–251

27 Partsch, H.A.: 'Specification and Transformation of Programs' (Springer-Verlag, New York, 1990)

28 Pettorossi, A., and Proietti, M.: 'Rules and strategies for transforming functional and logic programs', *ACM Comput. Surv.*, 1996, **28**, (2), pp. 361–414

29 Bauer, F.L., Möller, B., Partsch, H., and Pepper, P.: 'Formal program construction by transformation – computer-aided, intuition guided programming', *IEEE Trans. Softw. Eng.*, 1989, **15**, (2), pp. 165–180

30 Kloos, C.D., and Dosch, W.: 'Transformation development of circuit descriptions for binary adders', *Lect. Notes Comput. Sci.*, 1999, **544**, pp. 217–237

31 Gajski, D.D., Dutt, N.D., Wu, A.C-H., and Lin, S.Y-L.: 'High-Level Synthesis' (Kluwer Academic Publishers, Boston, 1992)

32 De Micheli, G.: 'Synthesis and Optimization of Digital Circuits' (McGraw-Hill, New York, 1994)

33 Voeten, J.: 'On the fundamental limitations of transformational design', *ACM Trans. Des. Autom. Electron. Syst.*, 2001, **6**, (4), pp. 533–552

34 Gajski, D.D., and Ramachandran, L.: 'Introduction to high-level synthesis', *IEEE Des. Test Comput.*, 1994, **11**, (4), pp. 44–54

35 Lu, Z., Sander, I., and Jantsch, A.: 'A case study of hardware and software synthesis in ForSyDe'. Presented at the 15th Int. Symp. on System synthesis, Kyoto, Japan, October 2002

36 Lee, E.A., and Sangiovanni–Vincentelli, A.: 'A framework for comparing models of computation', *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, 1998, **17**, (12), pp. 1217–1229