Development and Application of Design Transformations in ForSyDe*

Ingo Sander, Axel Jantsch, and Zhonghai Lu
Royal Institute of Technology
Stockholm, Sweden
{ingo,axel,zhonghai}@imit.kth.se

Abstract

The ForSyDe methodology has been developed for system level design. Starting with a formal specification model, that captures the functionality of the system at a high abstraction level, it provides formal design transformation methods for a transparent refinement process of the system model into an implementation model that is optimized for synthesis. The main contribution of this paper is the formal treatment of transformational design refinement. 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. Thus, our approach avoids discontinuities since it moves design refinement into the domain of the specification model.

1 Introduction

Keutzer et al. [5] 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 orthogonalization 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. They "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".

*This research was supported by the Swedish Foundation for Strategic Research within the INTELECT program.

These arguments strongly support the ForSyDe (Formal System Design) methodology which addresses the transformational design of SoC applications. In contrast to [12], where we have illustrated the large potential of our approach by two powerful transformations for clock domain and communication refinement, the emphasis of this paper is on the formal treatment of design transformations in ForSyDe. 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

According to the tagged signal model developed by Lee and Sangiovanni-Vincentelli [6] our system model can be classified as synchronous computational model. It is based on the synchrony hypothesis, that also forms the base for the 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" [1]. The synchronous assumption implies a total order of events and leads to a clean separation between computation and communication and gives a solid base for formal methods. Hsieh et al. [4] define *synchronous equivalence* based on a different *synchronous assumption* which does not assume a *zero delay* for the computation phase. They focus only on verification aspects but do not propose a design methodology.

The parallel programming community has used functional languages to derive parallel programs from a functional specification [14]. They use skeletons to structure a problem. This formulation is then transformed, using cost measures, into an efficient implementation for a chosen computer architecture. Reekie [11] has used Haskell to model digital signal processing applications. Similarly to us, he modeled 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. Lava [2] is a hardware de-

scription 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. Hardware ML (HML) [7] 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 [9] to hardware/software codesign. They transform SAFL programs by means of meaning preserving transformations and compile the resulting program in a resource-aware manner, i.e. a function that is called more than once will be a shared resource.

Transformational approaches have a long history [10]. However, most of the approaches are concerned with software programs where concepts of synchronous subdomains and resource sharing, as discussed in this paper, have no relevance. There are also a number of other transformational approaches targeting hardware design, e.g. [13], but none of them explicitly develops the concept of design decisions or addresses the refinement of a synchronous model into multiple synchronous sub-domains as we attempt in this article. In particular our approach allows to use the large amount of work that exists for high-level synthesis [3] by defining design decision transformations for refinement techniques like re-timing or resource sharing.

3 The ForSyDe Methodology

3.1 The Design Process

The ForSyDe design process starts with the development of a formal, abstract, functional *specification model* that 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 into an *implementation model*. As the implementation model is a refined version of the specification model, the same validation and verification methods can be applied to both models. In the partitioning phase, the implementation model is partitioned into hardware and software blocks, which are mapped on architectural components. Only now, in the code generation phase, we leave the functional domain and enter the *implementation domain* to produce VHDL or C/C++ for the hardware and software parts [8].

3.2 The Specification Model

The specification model is based on 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 behavior 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 [6]. 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 value at a certain tag, a data type D can be extended into a data type $D_\perp$ by adding the special value $\perp$. Absent values are used to establish a total order of events when dealing with signals with different or aperiodic event rates.

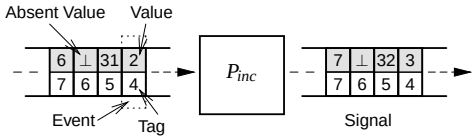


Figure 1. Modeling of Signals and Processes

Figure 1 illustrates the modeling of signals and the behavior of processes. During the event cycle n a process processes the events of each signal with the tag n and outputs the result at 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. In case of 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.

$$\begin{aligned} s &= \langle e_0, e_1, \ldots \rangle \\ s_k &= \langle (e_k)_0, (e_k)_1, \ldots \rangle \end{aligned}$$

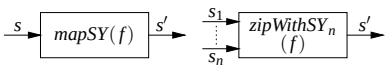
Parallel signals are described as a tuple of signals. A process P maps m input signals on n output signals.

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

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

As we use the perfect synchrony hypothesis [1], 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 modeling of processes. This leads to a well defined specification model where all processes are constructed by process constructors. There is a clean separation between *synchronization* (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 [8].



$$
\begin{array}{lcll}
mapSY(f) & = & P & \\
\text{where} & & P(s) & = & s' \\
 & & f(e_i) & = & e'_i \\
zipWithSY_n(f) & = & P & \\
\text{where} & & P(s_1,s_2,\ldots,s_n) & = & s' \\
 & & f((e_1)_i,(e_2)_i,\ldots,(e_n)_i) & = & e'_i
\end{array}
$$

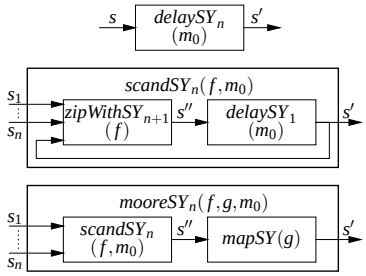
Figure 2. The Combinational Process Constructors *mapSY* **and** *zipWithSY*

The process constructor *mapSY* takes a combinational function f and constructs a process with one input and output signal, where f is applied on all values of the input signal. The process constructor $zipWithSY_n$ corresponds to *mapSY*, but creates processes with multiple input signals. There is the short notation $*$ for *mapSY* and $\triangleright_n$ for $zipWithSY_n$. Both processes are illustrated in Figure 2.

The basic sequential process constructor $delaySY_n$ (short notation $\triangle_n$) constructs a process that delays a signals n cycles. We define $scandSY_n$ that 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 define *mooreSY* to model a Moore FSM (Figure 3).



$$
\begin{array}{llcl}
delaySY_n & (m_0) & = & P \\
\text{where} & P(s) & = & s' \\
 & e'_i & = & \begin{cases} m_0 & i < n \\ e_{i-n} & i \geq n \end{cases} \\
scandSY_n & (f,m_0) & = & P \\
\text{where} & P(s_1,\ldots,s_n) & = & s' \\
 & s' & = & \langle m_0, e_0'', e_1'', \ldots \rangle \\
 & \triangleright_{n+1}(s_1,\ldots,s_n,s') & = & s''
\end{array}
$$

$$mooreSY_n(f,g,m_0) = mapSY(g) \circ scandSY_n(f,m_0)$$

Figure 3. The Sequential Process Constructors $delaySY_n$**,** $scandSY_n$ **and** $mooreSY_n$

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 *synchronous sub-domains* which comprise a local synchronous process network with a different signal rate as illustrated in Figure 4.

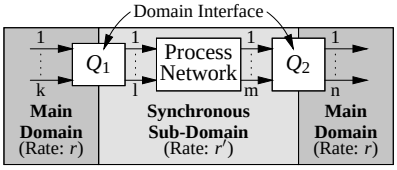


Figure 4. Synchronous Sub-Domains

In order to formally describe implementation models with synchronous sub-domains we extend our notation of signals by including the rate $r \in \mathbb{Q}$ to the form $s^r = \langle e_0^r, e_1^r, \ldots \rangle$ where the tag $\frac{r}{i}$ is given by position i and rate r. A domain interface consumes m input signals with rate r and produces n output signals with another rate r'. In the specification model the rate of all signals is 1.

Synchronous sub-domains violate the synchronous assumption since 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 can also reason about a refined model with synchronous sub-domains as further elaborated in Section 4.

4 Refinement

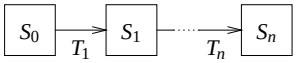


Figure 5. Transformational Refinement

One main 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 S_0 is stepwise refined by well defined design transformations T_i into a final implementation model S_n (Figure 5). Only at this late stage the implementation model is translated using the ForSyDe hardware and software semantics into a synthesizable implementation description.

Definition 1 (Transformation Rule) *A* transformation rule *is a functional mapping of a process network PN_1 onto another process network PN_2 with the same input signals and the same number of output signals. A transformation rule is denoted by $R(PN_1) = PN_2$ or $PN_1 \xrightarrow{R} PN_2$.*

Definition 2 (Transformation) *A* transformation *$T(S_1, PN_1, R)$ is a functional mapping of a system model S_1 onto another system model S_2 with the same input signals and the same number of output signals. Using the transformation rule R the internal process network PN_1 in S_1 is replaced by $R(PN_1)$ to yield S_2. A transformation is denoted by $T(S_1, PN_1, R) = S_2 = S_1[R(PN_1)/PN_1]$ or $S_1 \xrightarrow{T(PN_1)} S_2 = S_1[R(PN_1)/PN_1]$, 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 characterizes the functional behavior 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 $(s'_1)^u, \ldots, (s'_n)^u$ expresses the dependence of the output events at tag $\frac{u}{i}$ on the input signals.*

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

The characteristic function can be derived for any process network including domain interfaces. Processes based only on combinatorial process constructors have a characteristic function that only depends on current input events. Here we give the characteristic function for the basic combinatorial processes *mapSY* and $zipWithSY_n$.

$$\begin{array}{lcll} (e')_i^u & = & \mathcal{F}_{*(f)}(s^r, i) & = & (f(e_i))_i^r \\ (e')_i^u & = & \mathcal{F}_{\triangleright_n(f)}((s_1)^r, \ldots, (s_n)^r, i) & = & (f(e_1)_i, \ldots, (e_n)_i)_i^r \end{array}$$

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

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

The characteristic functions for FSM processes like $mooreSY_n$ are more complex since they depend on past values and include an internal feedback loop.

$$\begin{array}{lcl} (e')_i^r & = & \mathcal{F}_{mooreSY_n(f,g,m_0)}((s_0)^r, \ldots, (s_n)^r, i) \\ & = & \begin{cases} (g(m_0))_i^r & i = 0 \\ (g(f((e_1)_0, \ldots, (e_n)_0, m_0)))_i^r & i = 1 \\ \vdots & \vdots \end{cases} \end{array}$$

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

Definition 4 (Semantic Preserving Transformation) *A transformation $S_1 \xrightarrow{T(PN_1)} S_2$ is* semantic preserving, *if $\mathcal{F}_{S_1}(s_1, \ldots, s_m, i) = \mathcal{F}_{S_2}(s_1, \ldots, s_m, i)$.*

Definition 5 (Design Decision) *A transformation $S_1 \xrightarrow{T(PN_1)} S_2$ is a* design decision, *if $\mathcal{F}_{S_1}(s_1, \ldots, s_m, i) \neq \mathcal{F}_{S_2}(s_1, \ldots, s_m, i)$.*

Semantic preserving transformations do not change the meaning of the model and are mainly used to optimize the model for synthesis. In contrast, design decisions change the meaning of a model. 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 characterized by a *name*, the *required format and constraints of the original process network*, the *format of the transformed process network* and the *implication for the design*, i.e. the relation between original and transformed process network expressed by the characteristic function.

We exemplify transformation rules by a combinatorial process with *n* inputs. If the process has a regular structure such as an *N*-input adder or multiplier, where N=4,8,16,... the process can be transformed into a balanced network of $N-1$ 2-input processes. This transformation $BalancedTree(PN_1)$ is defined in the transformation library as

Transformation Rule : $BalancedTree(PN_1)$
Orignal Process Network :

$$\begin{array}{lcl} PN_1(s_1, \ldots, s_N) & = & \triangleright_N(f)(s_1, \ldots, s_N) \\ N & = & 2^k; k \in \mathbb{N} > 1 \\ f(x_1, \ldots, x_N) & = & x_1 \otimes \ldots \otimes x_N; \otimes \text{ is associative} \end{array}$$

Transformed Process Network :

$$\begin{array}{lcl} PN_2(s_1, \ldots, s_N) & = & \triangleright_2(g)(\ldots(\triangleright_2(g)(s_1, s_2), \triangleright_2(g)(s_3, s_4)), \\ & & \ldots(\triangleright_2(g)(s_{N-3}, s_{N-2}), \triangleright_2(g)(s_{N-1}, s_N))) \\ g(x,y) & = & x \otimes y \end{array}$$

Implication :

$$\mathcal{F}_{PN_1}(s_1, \ldots, s_N, i) = \mathcal{F}_{PN_2}(s_1, \ldots, s_N, i); \forall i \in \mathbb{N}_0$$

This transformation can be used for all processes that comply to 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 $BalancedTree(PN_1)$ is semantic preserving since the characteristic function of the original and transformed process network is identical.

There is another transformation *PipelinedTree* that pipelines a balanced tree structure of possibly different 2-input processes into a pipelined tree structure.

TransformationRule :$PipelinedTree(PN_1)$
OriginalProcessNetwork :

$$\begin{aligned} PN_1(s_1,\ldots,s_N) &= \triangleright_2(g_{N-1})(\ldots(\triangleright_2(g_1)(s_1,s_2),\ldots),\\ &\quad \ldots(\triangleright_2(\ldots,\triangleright_2(g_{N/2})(s_{N-1},s_N))) \\ N &= 2^k; k \in \mathbb{N} > 1 \end{aligned}$$

TransformedProcessNetwork :

$$\begin{aligned} PN_2(s_1,\ldots,s_N) &= \triangle_1(m_0)\circ\triangleright_2(g_{N-1})\\ &\quad (\ldots(\triangle_1(m_0)\circ\triangleright_2(g_1)(s_1,s_2),\ldots),\\ &\quad \ldots(\ldots,\triangle_1(m_0)\circ\triangleright_2(g_{N/2})(s_{N-1},s_N))) \end{aligned}$$

Implication :

$$\mathcal{F}_{\triangle_k(m_0)\circ PN_1}(s_1,\ldots,s_N,i) = \mathcal{F}_{PN_2}(s_1,\ldots,s_N,i); \forall i \geq k$$

As expressed in the Implication, *PipelinedTree* is a design decision since 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 he chooses transformations from the library and applies them successively as visualized in Figure 6, where a 4-input addition process is transformed into a pipelined structure.

A direct translation of a computation intensive algorithm such as an n-th-order FIR filter results in an implementation with a large amount of multipliers and adders. Using the concept of synchronous sub-domains we have developed a transformation *SerialClockDomain* that transforms a combinatorial processes of a regular structure into a structure with two clock domains that uses an FSM process to schedule the operations into several clock cycles. This transformation which is illustrated in Figure 7 and formally given below is very efficient, if there are identical operations which can be shared.

TransformationRule :$SerialClockDomain(PN_1)$
OriginalProcessNetwork :

$$\begin{aligned} PN_1(s_1,\ldots,s_n) &= \triangleright_n(f)(s_1,\ldots,s_n) \\ f(x_1,\ldots,x_n) &= g_{n-1}(h_n(x_n),(\ldots,(g_1(h_2(x_2),h_1(x_1)))\ldots)) \end{aligned}$$

TransformedProcessNetwork :

$$\begin{aligned} PN_2(s_1,\ldots,s_n) &= (D_H(n)\circ P_{FSM}\circ P/S_n)(s_1,\ldots,s_n) \\ P_{FSM} &= mooreSY(f',g',m_0) \\ f'(x,(i,m)) &= \begin{cases} (1,h_i(x)) & i=0 \\ (i+1,g_i(h_i(x),m)) & 0<i<n-1 \\ (0,g_i(h_i(x)+m)) & i=n-1 \end{cases} \\ g'(i,m) &= \begin{cases} m & i=0 \\ \perp & 0<i<n \end{cases} \end{aligned}$$

Implication :

$$\mathcal{F}_{\triangle_1(m_0)\circ PN_1}(s_1,\ldots,s_n,i) = \mathcal{F}_{PN_2}(s_1,\ldots,s_n,i)$$

The transformed process network works as follows. During an input event cycle the domain interface P_n/S (Parallel to Serial) reads all input values at rate r and outputs them at rate nr one by one in the corresponding n output cycles. The process P_{FSM} is based on *mooreSY* and executes the combinational function f of the original process in n cycles. In state 0 the first input value (operand x_1) is stored as intermediate value m. In the $n-1$ following states a function g_i is applied to the new input value (x_i) and the intermediate value. At tag 0,n,2n,... the process produces the output value, otherwise the output has the value $\perp$. The domain interface $D_H(n)$ down-samples the input signal to rate r and outputs only each n-th input value starting with tag 0, thus suppressing the absent values from the output of P_{FSM}.

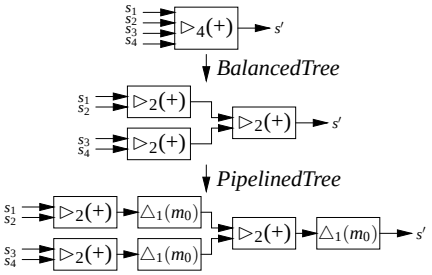


Figure 6. Transformation into Balanced Pipelined Structure

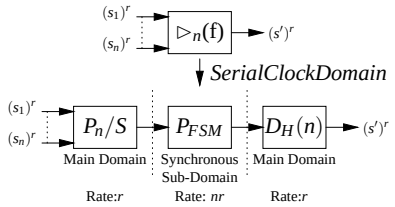


Figure 7. Transformation into FSM using two Clock Domains

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

This transformation can e.g. be applied to the 4-input adder of Figure 6, 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 a FIR-filter

We will now use the developed transformation *SerialClockDomain* for the refinement of a FIR-filter which is part of the specification model of a digital equalizer [12]. A FIR-Filter is described by the following equation:

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

We have modeled the FIR-filter as shown in Figure

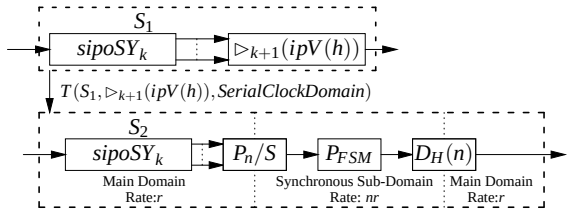


Figure 8. Transformation of a FIR-Filter

8. The process $sipoSY_k$ models a serial-in-parallel-out shift register with $k+1$ output signals. The process $zipWithSY_{k+1}(ipV(h))$ computes the inner product of the coefficient vector, $h_0,\ldots,h_k$, and the outputs of the process $sipoSY_k$, $x_n,\ldots,x_{n-k}$. Since the process $ipV(h)$ is defined as

$$(ipV(h))(x_0,\ldots,x_n) = h_0x_0 + \ldots + h_nx_n$$

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

$$\begin{aligned} g_i(x,y) &= x+y \\ h_i(x) &= h_ix \end{aligned}$$

We can use this rule to apply the transformation $T(S_1, zipWithSY_{k+1}(ipV(h)), SerialClockDomain)$ on the FIR-filter model S_1 in order to receive a model S_2, where $sipoSY_k$ remains unchanged and the FIR-filter is realized with two clock domains and only one multiplier and one adder (Figure 8).

We have used the ForSyDe hardware semantics to translate both the original model and the transformed model for an 8-th order FIR-filter with sample and coefficient size of 10-bit into VHDL and synthesized it for the CLA90K 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).

6 Conclusion

The contribution of this paper 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 that 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 by an implication part, expressed with the characteristic function.

As illustrated in the paper, traditional and powerful synthesis techniques can now be formulated as transformation rules and be applied inside the functional domain. This is in 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.

References

[1] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, September 1991.

[2] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in Haskell. In *International Conference on Functional Programming*, 1998.

[3] D. D. Gajski, N. D. Dutt, A. C.-H. Wu, and S. Y.-L. Lin. *High-Level Synthesis*. Kluwer Academic Publishers, 1992.

[4] H. Hsieh, F. Balarin, L. Lavagno, and A. Sangiovanni-Vincentelli. Synchronous approach to functional equivalence of embedded system implementations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 20(8):1016–1033, August 2001.

[5] K. Keutzer, S. Malik, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: Orthogonolization of concerns and platform-based design. *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, 19(12):1523–1543, December 2000.

[6] E. A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, December 1998.

[7] Y. Li and M. Leeser. HML, a novel hardware description language and its translation to VHDL. *IEEE Transactions on VLSI*, 8(1):1–8, February 2000.

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

[9] A. Mycroft and R. Sharp. Hardware/software co-design using functional languages. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2031 of *LNCS*. Springer-Verlag, 2000.

[10] A. Pettorossi and M. Proietti. Rules and strategies for transforming functional and logic programs. *ACM Computing Surveys*, 28(2):361–414, June 1996.

[11] H. J. Reekie. *Realtime Signal Processing*. PhD thesis, University of Technology at Sydney, Australia, 1995.

[12] I. Sander and A. Jantsch. Transformation based communication and clock domain refinement for system design. In *39th Design Automation Conference (DAC 2002)*, New Orleans, USA, June 2002.

[13] T. Seceleanu. *Systematic Design of Synchronous Digital Circuits*. PhD thesis, University of Turku, Finland, 2001.

[14] D. Skillicorn. *Foundations of Parallel Programming*. Cambridge University Press, 1994.