System Modeling and Transformational Design Refinement in ForSyDe

Ingo Sander, *Member, IEEE,* and Axel Jantsch, *Member, IEEE*

Abstract—**The scope of the Formal System Design (ForSyDe) methodology is high-level modeling and refinement of systems-on-a-chip and embedded systems. 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 ForSyDe modeling technique and the formal treatment of transformational design refinement. We introduce *process constructors*, that cleanly separate the computation part of a process from the synchronization and communication part. We develop the *characteristic function* for each process type and use it to define *semantic preserving* and *design decision transformations*. These transformations are characterized by *name*, the format of the *original process network*, the *transformed process network*, and a *design implication*. The implication expresses the relation between original and transformed process network by means of the characteristic function. The objective of the refinement process is a model that can be implemented cost efficiently. To this end, process constructors and processes have a *hardware* and *software interpretation* which shall facilitate accurate performance and cost estimations. In a study of a digital equalizer example, we illustrate the modeling and refinement process and focus in particular on *refinement of the clock domain*, *communication refinement*, and *resource sharing*.**

Index Terms—**Formal methods, hardware/software codesign, modeling, system-on-a-chip (SoC).**

I. 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 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."

Manuscript received July 9, 2002; revised January 24, 2003. This work was supported by the Swedish Foundation for Strategic Research and is part of the Integrated Electronic Systems program. This paper was recommended by Associate Editor R. Camposano.

The authors are with the Department of Microelectronics and Information Technology, Royal Institute of Technology, 164 40 Stockholm, Sweden (e-mail: ingo@imit.kth.se; axel@imit.kth.se).

Digital Object Identifier 10.1109/TCAD.2003.819898

These arguments strongly support the formal system design (ForSyDe) methodology, as many of their main requirements on a system design methodology are not only part of our methodology, but establish the foundations of ForSyDe. We believe that a future system design methodology must give support for the integration of formal methods, since it is more and more unlikely that the ever increasing functionality inside an SoC can be verified with only simulation and emulation techniques.

We have developed the ForSyDe methodology with the design of SoC (system-on-a-chip) technologies in mind. However, we are aware of the fact that ForSyDe, as a research project, cannot handle all aspects of an SoC, since SoCs are of extreme complexity and can include a variety of components, such as analog parts, microcontroller cores, digital signal processor cores, memories, IP blocks, and custom hardware. The design of an SoC demands the development of a communication structure between these heterogeneous components on all layers, from the physical layer to the application layer. Software has to be developed not only for different processors, but also for different operating systems.

The contribution of ForSyDe is a modeling and refinement technique that can be used as part of the design process of an SoC. Our goal is to develop these abstract techniques to such an extent that they can be incorporated into future computer-aided design methodologies and tools. The main objective is to move design refinement from the implementation into the functional domain. 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 specification model into an implementation model. This implementation model is based on the same semantics as the initial specification model, but is more detailed and optimized for implementation after the stepwise application of semantic preserving and design decisions transformations. At this late stage in the design process, the implementation model is mapped onto an implementation.

This paper presents the modeling and refinement technique in ForSyDe. We exemplify these techniques by the case study of a digital equalizer. The first part of this case study covers the development of the specification model, while the second part illustrates the potential for design refinement by the application of transformations for the clock domain and communication refinement and resource sharing.

II. Related Work

Skillicorn and Talia discuss models of computation for parallel architectures in [2]. Their community faces similar or even identical problems as are typical for SoC design, since an SoC architecture often includes a number of parallel microprocessors. In fact, all typical parallel computer structures (SIMD, MIMD) can be implemented on an SoC architecture. Recognizing, that programming of a large number of communicating processors is an extremely complex task, they try to define properties for a suitable model of parallel computation. They emphasize that a model should hide most of the details (decomposition, mapping, communication, synchronization) from programmers, if they shall be able to manage intellectually the creation of software. The exact structure of the program should be inserted by the translation process rather than by the programmer. Thus, models should be as abstract as possible, which means that the parallelism has not even to be made explicit in the program text. They point out that *ad hoc* compilation techniques cannot be expected to work on problems of this complexity, but advocate to build software, that is correct by construction rather then verifying program properties after construction. Programs should be architecture independent to allow reuse. The model should support cost measures to guide the design process and should have guaranteed performance over a useful variety of architectures.

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

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

According to this scheme, the ForSyDe modeling approach can both be classified: 1) with focus on modeling as "nothing explicit," since the designer is able to develop the specification model without even being aware that the system will be implemented on a parallel architecture and 2) with focus on implementation as "parallelism" and "communication explicit," since the specification model can be interpreted as a concurrent process model with a synchronous communication. However, neither the process nor the communication structure is fixed, since during the refinement phase, processes can be merged and split and synchronous communication channels can be refined into asynchronous channels as elaborated in this article.

Edwards *et al.* [3] believe, that the design approach should be based on the use of one or more formal methods to describe the behavior of the system at a high level of abstraction, before a decision on its decomposition into hardware and software is taken. The final implementation of the system should be made by using automatic synthesis from this high-level of abstraction to ensure implementations, that are "correct by construction." Validation through simulation or verification should be done at the higher levels of abstraction.

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

1) a functional specification;
2) a set of properties, that the design has to satisfy;
3) a set of performance indexes that evaluate the design in different aspects (speed, size, reliability, etc.);
4) a set of constraints on performance indexes.

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

Using the tagged signal model introduced by Lee and Sangiovanni-Vincentelli [4], they classify and analyze several models of computation, in particular discrete event models, communicating finite state machines (FSMs), synchronous/reactive models and data flow process networks. It appears that different models fundamentally have different strength and weaknesses, and that attempts to find their common features result in models that are very low level and difficult to use.

According to the tagged signal model our system model can be classified as synchronous computational model. The ForSyDe system model is based on the perfect synchrony hypothesis, that also forms the base for the family of 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." [5] The basic synchronous assumption is that the outputs of the system are synchronized 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 that are conceptually simultaneous and instantaneous. This assumption frees the designer from the modeling of complex communication mechanisms. These properties give a solid base for formal methods.

A synchronous model can be implemented on a target machine, if this machine is "fast enough." This design technique has been used in hardware design for clocked synchronous circuits. A circuit behavior can be described determinately independent of the detailed timing of gates, by separating combinational blocks from each other with clocked registers. An implementation will have the same behavior as the abstract circuit under the assumption, that the combinational blocks are "fast enough."

Data flow models [6], such as Kahn process networks [7] or synchronous data flow (SDF) [8], are well suited for applications that 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 be able to express timing properties and constraints on a level that abstracts from physical time.

Balarin *et al.* [9] argue that the synchronous assumption, though very convenient from the analyzing point of view, imposes a too strong restriction on the implementation, as it has to be "fast enough." They advocate a globally asynchronous locally synchronous (GALS) approach and implement it in their methodology as a network of Codesign FSMs communicating via events. Each CFSM is a synchronous FSM, but the communication is done by the emission of events by the CFSMs, which can happen at any time and independently. The CFSM network is inherently nondeterministic. Balarin *et al.* argue, that this enables them to easily model the unpredictability of the reaction delay of a CFSM both at the specification and at the implementation level, while they admit, that nondeterminism makes the design and verification process more complex.

We argue, that the advantages of a deterministic synchronous system model outweigh the disadvantages. Nondeterminism in the system model implies, that all possible solutions have to be considered, which makes both the design and verification process more complex. We believe that the task to develop and to verify a system model for an SoC application is so complex, that a system model has to be deterministic. In our opinion, the fact that SoC applications will be implemented on at least partly asynchronous architectures does not justify a nondeterministic approach. Following Skillicorn and Talia [2], we think that the synthesis of the system model into a partly asynchronous implementation should be part of the synthesis process and not already be decided at the system level.

The synchronous languages [10] are based on the perfect synchrony hypothesis and have been developed for the design of reactive systems, i.e., systems that maintain a permanent interaction with the environment. All synchronous languages are defined formally and lead to deterministic system models. The family of synchronous languages can be divided into two groups, one group targeting data flow applications, e.g. Lustre [11] and Signal [12], the other targeting control oriented applications, e.g., Statecharts, [13], Esterel [14], and Argos [15]. However, there is no synchronous language covering both application domains [5]. The clean mathematical formalism has led to the development of several verification tools for these languages. [16] give an overview over the techniques and tools developed for the validation of reactive systems described in Lustre. However, they point out, that these techniques can be adapted to any synchronous language.

Hsieh *et al.* define another *synchronous assumption* in [17]. A cycle consists of an interaction phase (where the environment interacts with the design) followed by a computation phase (where the components in the design perform computation and communicate internally). They do not assume a *zero delay* for the computation phase, as in the case of the perfect synchrony hypothesis. Using their definition they define *synchronous equivalence* as: “Two implementations are synchronously equivalent if and only if any two synchronous assumption conforming runs of the two implementations that have the same input traces also have the same output traces.” As long as the primary outputs are the same at the end of every cycle, the internal details of the execution do not matter. In ForSyDe, *synchronous equivalence* can be shown with the characteristic function. The use of the characteristic function is not restricted to synchronous models, but can also be implied on models with synchronous subdomains.

We base our approach on the same foundation as the synchronous languages, the perfect synchrony hypothesis, but extend it to cover both, control and data flow applications. We achieve this by a formal system modeling technique which models can be expressed with the purely functional programming language Haskell [18]. While functional languages fit naturally for data flow applications, Haskell provides a rich variety of control constructs, making it suitable for control dominated applications. Haskell is defined by a formal semantics and is purely functional, i.e., a function has no side effects, resulting in a system model, that in itself is a function with no side effects and thus totally deterministic.

While these properties mainly support formal verification methods, other properties support design correctness. According to Lee [19], type systems do more than any other formal method to ensure software correctness. Haskell is not only a strongly typed language, but its type system has also the ability to infer the correct type for a function, which offers another dimension of polymorphism compared to some popular object oriented languages, such as C++ or Java.

While the high level of abstraction fits well for system level specification, there is a gap between the system model and a possible implementation on an SoC architecture. We try to bridge this gap by the concept of process constructors. Though the system model is formulated as a function, the use of process constructors implies that the functional model can be interpreted as a network of synchronously communicating concurrent processes. Such a process structure is almost fixed in other design languages (VHDL, SDL), but in ForSyDe processes can be merged and split during the application of transformation rules during the design transformation phase [20]. 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.

While we advocate to use a single unified-system model in the ForSyDe methodology, a lot of work is done using mixed models of computation. This approach has the advantage, that a suitable model of computation can be used for each part of the system. On the other hand, as the system model is based on several computational models, the semantics of the interaction of fundamentally different models has to be defined, which is not a simple task. This even amplifies the verification problem, because the system model is not based on a single semantics. There is little hope that formal verification techniques can help, and 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 optimize systems between different models of computation. In summary, cross domain verification and optimization will remain elusive for many years for any heterogeneous modeling approach.

The MASCOT methodology [21] integrates data and control flow at the system specification level, using the two languages Matlab and SDL. The dataflow parts are described in Matlab. and the control flow parts in SDL. The system is then cosimulated using a library of wrappers and communication functions. The computational model is elaborated in [22].

The Ptolemy project [23] “studies heterogeneous modeling, simulation, and design of concurrent systems. The objective in Ptolemy II is to “support the construction and interoperability of executable models that are build under a wide variety of models of computation.”

Internal representations like the SPI model [24] and FunState [25] 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."

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

Reekie [26] has used Haskell to model digital signal-processing applications. Similar to us, he modeled streams as infinite lists and used higher-order functions to operate on them. Finally, correctness-preserving methods were applied to transform a model into a more efficient representation. This representation was not synthesized to hardware or software.

Ruby is a declarative language that has mainly been used for hardware design. In [27], 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 [28] is a hardware description language based on Haskell. Similar to Ruby it focuses on the structural representation of hardware and offers a variety of powerful connection patterns. Lava descriptions can be translated into VHDL and there exist interfaces to formal method tools.

Mycroft and Sharp have used the languages statically allocated functional language (SAFL) and SAFL+ [29], mainly for hardware design but extended their approach in [30] 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.

The Hawk language [31] embedded in Haskell is used for building executable specifications of microprocessors. The Hawk project addresses the need for verification of complex modern microprocessors, which is supported by the formal nature of a Hawk specification. Hawk has been used to specify and simulate the integer part of a pipelined DLX processor.

Hardware ML (HML) [32] is a hardware-description language that is based on the functional programming language Standard ML [33]. Though HML uses some features of Standard ML, such as polymorphic functions and its type system, it is mainly an improvement of VHDL—there is a direct mapping from HML constructs into the corresponding VHDL constructs.

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

A good overview about program transformation in general is given in [34] and for transformation of functional and logical programs in [35]. One of the most well-known transformation systems is the computer-aided, intuition-guided programming (CIP) project [36]. Inside CIP, program development is viewed as an evolutionary process that usually starts with a formal problem specification and ends with an executable program for the intended target machine. The individual transformations are done by semantic preserving transformation rules, which guarantees that the final version of the program still satisfies the initial specification. Such an approach has the following advantages [36].

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

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

Most of the transformational approaches are concerned with software programs, where concepts of synchronous 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., [37] and [38]), but none of them explicitly develops the concept of design decisions or addresses the refinement of a synchronous model into multiple synchronous subdomains as we attempt in this article. In particular, our approach allows us to use the large amount of work that exists for high-level synthesis [39], [40] by defining design decision transformations for refinement techniques like retiming or resource sharing.

Voeten points out that each transformational design that is based on a general purpose language will suffer from fundamental incompleteness problems [41]. This means that the initial model to a large extent determines whether an effective and satisfying implementation can be obtained or not, since only a limited part of the design space can be explored. The same problem is known in the context of high-level synthesis as syntactic variance problem [42], which in general is unsolvable.

III. ForSyDe Methodology

The design process in ForSyDe starts in the functional domain with the development of an abstract and formal specification model according to the ForSyDe modeling guidelines. The designer verifies the functionality of the design by either simulation of the executable specification model, which is the verification technique we use today, and/or by formal verification techniques, which are planned to be incorporated into ForSyDe.

Design refinement starts after the functional verification of the design. Here, the designer refines the specification model into an implementation model by a stepwise application of design transformations, which are selected from a transformation library. The task of the refinement process is to optimize the specification model and to add the necessary implementation details in order to allow for an efficient mapping of the implementation model onto the chosen architecture. Every process must either be directly mappable or be part of mappable process network, provided with a mapping function onto an architecture component or IP. If a concurrent process structure shall be implemented on a single processor with operating system and memory, the refinement process must transform the specification model into an implementation model that includes a scheduling process. Only at this stage, the design process leaves the functional domain and enters the implementation domain by the mapping of the implementation model onto architecture components. The scheduling process is mapped onto services of the operating system and the concurrent processes are mapped to software processes.

So far, ForSyDe is restricted to a small number of transformations and mapping rules for hardware and software processes (Section IV-E), which however allow to show the potential of ForSyDe, but are far from sufficient to target realistic designs. In order to make ForSyDe applicable for larger designs, the following is necessary.

- New transformation rules have to be developed which capture the characteristics of a possible architecture component.
- A rich set of mappable processes and process networks has to be provided.
- Mapping rules between process, channels, and the architecture components have to be formulated in the same way as the hardware and software semantics.

The rest of the paper discusses the current state of ForSyDe and is structured as follows. Section IV presents the computational model and explains the ForSyDe modeling technique. Section V presents the foundations for refinement in ForSyDe. In particular, we show how the characteristic function is used for the development of new design transformations. Section VI uses the example of a digital equalizer to illustrate the development of a specification model and shows how three different refinement techniques can be used to optimize the specification model and to refine it into a more efficient implementation model.

IV. ForSyDe System Model

A. Computational Model

In order to formally describe the computational model of ForSyDe, we use a similar definition as used by Lee and Sangiovanni-Vincentelli [4].

A signal $\overrightarrow{s}$ is defined as a sequence of events

$$\overrightarrow{s} = \left\langle E(\overrightarrow{s},0), E(\overrightarrow{s},1), E(\overrightarrow{s},2), \cdots \right\rangle$$

where each event $E(\overrightarrow{s},j)$ has a tag $T(\overrightarrow{s},j)$ and a value $V(\overrightarrow{s},j)$, i.e., $E(\overrightarrow{s},j) = (T(\overrightarrow{s},j), V(\overrightarrow{s},j) \in \mathbf{T} \times \mathbf{V}$.

The ForSyDe methodology uses two kinds of system model. The *specification model* is used in the specification phase and complies with the perfect synchrony hypothesis [5], i.e., all processes are infinitely fast and all signals have the same set of tags, for which we use the set of natural numbers $\mathbb{N}_0$. We define the term *event cycle* $C(\overrightarrow{s})$, which defines the "distance" between the tags of two adjacent events in a signal $\overrightarrow{s}$ as

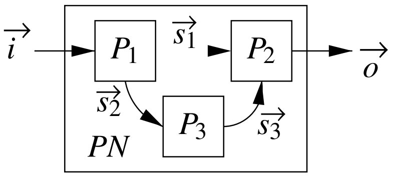


Fig. 1. System model is a hierarchical network of concurrent processes.

$$C(\overrightarrow{s}) = T(\overrightarrow{s}, j+1) - T(\overrightarrow{s}, j) = const \quad \forall j \in \mathbb{N}_0$$

and the *event rate* $R(\overrightarrow{s})$ of a signal $\overrightarrow{s}$ as $R(\overrightarrow{s}) = 1/C(\overrightarrow{s})$.

All signals in the specification model have an event rate of 1, while a signal in the *implementation model* may have an event rate $R(\overrightarrow{s}) \in \mathbb{Q}$ that is different from 1. However, in both models all signals have a constant event rate and have their first event at tag 0, so that there is a total order of all events in all signals in the model.

In order to model the absence of a value at a certain tag, a data type $\mathbf{V}$ can be extended into a data type $\mathbf{V}_\perp$ by adding the special value $\perp$. These absent values are used to establish a total order of events when modeling signals with a slower data rate or an aperiodic behavior, such as a reset signal. Note, that the term data rate expresses the rate of useful, i.e., nonabsent, values in a signal and is not identical to our definition of event rate.

A system model is hierarchical network of concurrent processes. The processes communicate with each other by means of signals. The process network PN of Fig. 1 is, in itself, a process and can be expressed as the following set of equations:

$$\mathrm{PN}(\overrightarrow{i}) = \overrightarrow{o}$$

where

$$\begin{aligned}(\overrightarrow{s_1}, \overrightarrow{s_2}) &= P_1(\overrightarrow{i})\\ \overrightarrow{o} &= P_2(\overrightarrow{s_1}, \overrightarrow{s_3})\\ \overrightarrow{s_3} &= P_3(\overrightarrow{s_2}).\end{aligned}$$

In the following, we define the terms process (Def. 1), *synchronous process* (Def. 2), and *domain interface* (Def. 3).

Definition 1: A process P is a functional mapping of m input signals $\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}$ into n output signals $\overrightarrow{o_1}, \ldots, \overrightarrow{o_n}$.

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

The set of processes is designated as $\mathbf{P}$.

Definition 2: A synchronous process P_S is a functional mapping of m input signals $\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}$ into n output signals $\overrightarrow{o_1}, \ldots, \overrightarrow{o_n}$, where all input and output signals have the same event rate r.

$$R(\overrightarrow{i_1}) = \cdots = R(\overrightarrow{i_m}) = R(\overrightarrow{o_1}) = \cdots = R(\overrightarrow{o_n}) = r$$

The set of synchronous processes is designated as $\mathbf{P_S}$.

Definition 3: A domain interface P_D is a functional mapping of m input signals $\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}$ into n output signals $\overrightarrow{o_1}, \ldots, \overrightarrow{o_n}$,

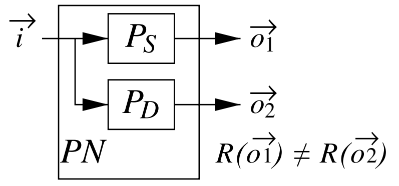


Fig. 2. Compositions of processes and domain interfaces may lead to processes that are neither synchronous processes nor domain interfaces.

where all input signals have the event rate r_i and all output signals have another event rate $r_o \neq r_i$

$$R(\vec{i_1}) = \cdots = R(\vec{i_m}) = r_i$$
$$R(\vec{o_1}) = \cdots = R(\vec{o_n}) = r_o$$

where

$$r_i \neq r_o.$$

The set of domain interfaces is designated as $\mathbf{P_D}$.

The specification model is a process network of synchronous processes and, thus in itself, a synchronous process. The implementation model is a composition of synchronous processes and domain interfaces and in itself a process that may not fall into the category of synchronous processes or domain interfaces as illustrated in Fig. 2.

B. Specification Model

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

- It is based on a *synchronous computational model*, which cleanly separates computation from communication.
- It is *purely functional* and *deterministic*.
- It uses *ideal data types* such as lists with infinite size.
- It uses the concept of well-defined *process constructors* which implement the synchronous computational model.
- It is based on a *formal semantics* and can be expressed in a modeling language. We have chosen to embed ForSyDe in the functional language Haskell [18].

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

Fig. 3 illustrates signals and processes inside the specification model. At tag n a process processes the events of each signal with the tag n and outputs the result at the same tag n.

As we use the perfect synchrony hypothesis [5], 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

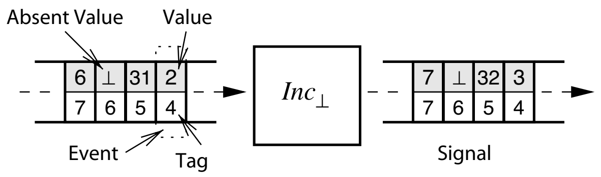
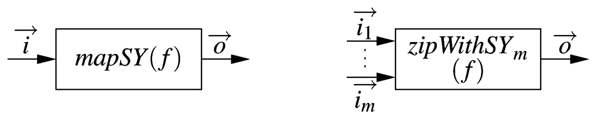


Fig. 3. Modeling of signals and processes.



$$
\begin{array}{lrcl}
\multicolumn{4}{l}{mapSY(f) = P_S \in \mathbf{P_S}} \\
\text{where} & \vec{o} & = & P_S(\vec{i}) \\
& T(\vec{o}, j) & = & T(\vec{i}, j) \\
& V(\vec{o}, j) & = & f(V(\vec{i}, j)) \\
\multicolumn{4}{l}{zipWithSY_m(f) = P_S \in \mathbf{P_S}} \\
\text{where} & \vec{o} & = & P_S(\vec{i_1}, \ldots, \vec{i_m}) \\
& T(\vec{o}, j) & = & T(\vec{i_k}, j) \quad \forall k.1 \leq k \leq m \\
& V(\vec{o}, j) & = & f(V(\vec{i_1}, j), \ldots, V(\vec{i_m}, j))
\end{array}
$$

Fig. 4. Combinational process constructors $mapSY$ and $zipWithSY$.

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 [43].

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 $zipWithSY_m$ corresponds to $mapSY$, but creates processes with multiple input signals. There is the short notation for $*$ $mapSY$ and $\triangleright_m$ for $zipWithSY_m$. Both processes are illustrated in Fig. 4.

We denote the processes created by $mapSY$ and a function f as the name of the function in capital letters, in this case F. The process $Inc_\perp$ of Fig. 3 is modeled by means of the process constructor $mapSY$, that maps a function $inc_\perp$ on all values of a signal. The function $inc_\perp$ differs from the increment function inc as it is able to process absent values. The function $inc_\perp$ can be generated by the higher-order function Ψ. It takes a function $f : D \rightarrow R$ and extends domain and range in order to cope with absent values $f_\perp : D_\perp \rightarrow R_\perp$. Ψ is defined by

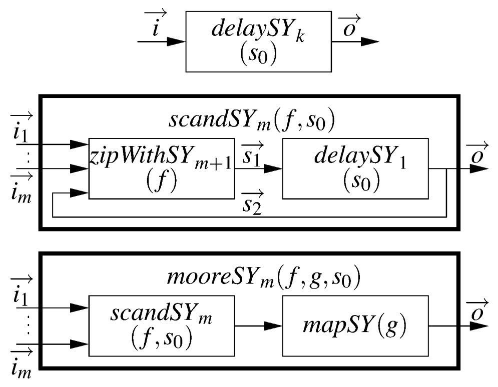
$$\Psi(f) = f_\perp$$

where

$$f_\perp(x) = \begin{cases} \perp, & \perp \\ f(x), & \text{otherwise} \end{cases}$$
$$\forall f : D \rightarrow R.$$

In the following, we define process constructors for sequential processes, i.e., processes that have an internal state.

The basic sequential process constructor $delaySY_k$ (short notation $\triangle_k$) constructs a process that delays a signals k event cycles. The process constructor $scandSY_m$ takes a function f and



$$
\begin{array}{l}
\mathit{delaySY}_k(s_0) = P_S \in \mathbf{P_S}\\
\quad \text{where} \quad \begin{array}{lcl}
\vec{o} & = & P_S(\vec{i})\\
T(\vec{o}, j) & = & T(\vec{i}, j)\\
V(\vec{o}, j) & = & \begin{cases} s_0 & \text{if } j < k\\ V(\vec{i}, j-k) & \text{otherwise}\end{cases}
\end{array}\\
\mathit{scandSY}_m(f, s_0) = P_S \in \mathbf{P_S}\\
\quad \text{where} \quad \begin{array}{lcl}
\vec{o} & = & P(\vec{i_1}, \ldots, \vec{i_m})\\
\vec{s_1} & = & (\mathit{zipWithSY}_{m+1}(f))(\vec{i_1}, \ldots, \vec{i_m}, \vec{s_2})\\
\vec{s_2} & = & (\mathit{delaySY}_1(s_0))(\vec{s_1})\\
\vec{o} & = & \vec{s_2}
\end{array}\\
\mathit{mooreSY}_m(f, g, s_0) = P_S \in \mathbf{P_S}\\
\quad \text{where} \quad \begin{array}{lcl}
\vec{o} & = & P_S(\vec{i_1}, \ldots, \vec{i_m})\\
P_S & = & \mathit{mapSY}(g) \circ \mathit{scandSY}_m(f, s_0)
\end{array}
\end{array}
$$

Fig. 5. Sequential process constructors $delaySY_k$, $scandSY_m$, and $mooreSY_m$.

a value s_0 for the initial state to construct the basic FSM process. Using the function composition operator $\circ$, where

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

we define the process constructor $mooreSY$ for the modeling of Moore FSMs (Fig. 5).

C. Implementation Model

The implementation model is a result of the refinement process (Section V). In contrast to the specification model, which is a network of concurrent synchronous processes, it may also conclude *domain interfaces* (Def. 3) in order to establish *synchronous subdomains* that comprise a local synchronous process network with a different event rate. These subdomains will be implemented in an own-clock domain. Domain interfaces are illustrated in Fig. 6.

Since synchronous subdomains violate the synchronous assumption, they are not allowed in the specification model, but are introduced by well-defined transformations during the refinement process. Inside a synchronous subdomain, the synchronous assumption is still valid and the same formal techniques can be used as for the initial system model. Due to the formal definition of domain interfaces, we can also reason about a refined model with synchronous subdomains as further elaborated in Section V.

In Fig. 7, we define domain interfaces for up- and down-sampling.

Next, we define interfaces for serial/parallel conversion (Fig. 8).

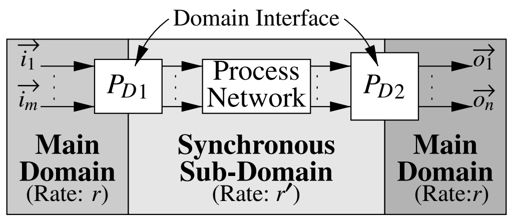
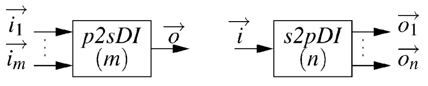


Fig. 6. Synchronous subdomains are introduced by domain interfaces.

upDI (k)
downDI (k)

$$
\begin{array}{l}
\mathit{upDI}(k) = P_D \in \mathbf{P_D}\\
\quad \text{where} \quad \begin{array}{lcl}
\vec{o} & = & P_D(\vec{i})\\
T(\vec{o}, j) & = & T(\vec{i}, kj)\\
V(\vec{o}, j) & = & V(\vec{i}, kj)\\
R(\vec{o}) & = & R(\vec{i})/k
\end{array}\\
\mathit{downDI}(k) = P_D \in \mathbf{P_D}\\
\quad \text{where} \quad \begin{array}{lcl}
\vec{o} & = & P_D(\vec{i})\\
T(\vec{o}, j) & = & jC(\vec{i})/k\\
V(\vec{o}, j) & = & \begin{cases} V(\vec{i}, \mathrm{mod}(j,k)) & \text{if } j = uk; u \in \mathbb{N}_0\\ \bot & \text{otherwise}\end{cases}\\
R(\vec{o}) & = & kR(\vec{i})
\end{array}
\end{array}
$$

Fig. 7. Domain interfaces for up- and down-sampling.



$$
\begin{array}{l}
\mathit{p2sDI}(m) = P_D \in \mathbf{P_D}\\
\quad \text{where} \quad \begin{array}{lcl}
\vec{o} & = & P_D(\vec{i_1}, \ldots, \vec{i_m})\\
T(\vec{o}, j) & = & jC(\vec{i_1})/m = \cdots = jC(\vec{i_m})/m\\
V(\vec{o}, j) & = & V(\vec{i_u}, w) \text{ with } \begin{array}{lcl} u & = & \mathrm{rem}(j,m)\\ w & = & \mathrm{mod}(j,m)\end{array}\\
R(\vec{o}) & = & mR(\vec{i_1}) = \cdots = mR(\vec{i_m})
\end{array}\\
\mathit{s2pDI}(n) = P_D \in \mathbf{P_D}\\
\quad \text{where} \quad \begin{array}{lcl}
(\vec{o_1}, \ldots, \vec{o_n}) & = & P_D(\vec{i})\\
T(\vec{o_k}, j) & = & T(\vec{i}, nj) \quad \forall k.1 \le k \le n\\
V(\vec{o_k}, j) & = & \begin{cases} \bot & \text{if } j = 0\\ V(\vec{i}, (j-1)n + k - 1) & \text{otherwise}\end{cases}\\
R(\vec{o_k}) & = & R(\vec{i})/n \quad \forall k.1 \le k \le n
\end{array}
\end{array}
$$

Fig. 8. Domain interfaces for parallel/serial conversion.

D. Modeling Language

In principle, all languages that are able to express our computational model may be used as modeling language for ForSyDe. Since we started from a research perspective, we have chosen the functional language Haskell [18] as modeling language, because it is free from side effects and supports many of our concepts, such as higher-order functions and laziness, and has a formal semantics. Thus, the implementation of signals, process constructors, domain interfaces, and Ψ is straightforward and allows us to express ForSyDe models in a clean way with minimal effort. For examples of ForSyDe expressed in Haskell, see [44].

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

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

- An incorporation of a more accepted modeling language would enable the designer to use ForSyDe, without learning a new language paradigm. However, the use of that language should be restricted in accordance to the ForSyDe principles. A similar idea has been used successfully in using VHDL for logic synthesis, where the synthesis semantics differ from the simulation semantics. Such an approach would allow us to gradually introduce new concepts into industrial design practice. It seems that C++ together with System C [45] is a good candidate, since it allows the development of a class library for process constructors. However, in contrast to Haskell, it is weak typed and not free from side effects. Thus, C++ will not aid the designer to the same extent.
- The development of a graphical user interface for ForSyDe would allow us to "hide" Haskell from the designer. The designer would be able to pick process contructors, which may have more intuitive names like $comb_1$ or $comb_2$, instead of $mapSY$ and $zipWithSY$, and only to formulate the corresponding combinational functions and initial values in order to specify a process. The specification model can then be developed by drawing signals between processes. Such a GUI could also assist the designer during design refinement, where the tool ideally would highlight possible transformations together with estimation data and the designer selects one of the proposed transformations.

Naturally, both approaches can be combined.

E. Hardware and Software Semantics

Processes constructed by process constructors and domain interfaces have a hardware and software interpretation. Since synthesis is not the topic of this article, we only exemplify the hardware and software semantics by a single example, the process constructor $mooreSY$. Hardware synthesis in ForSyDe has been discussed in [46] and [47], while [43] covers a case study on the synthesis of the ForSyDe model of a digital equalizer into VHDL and C.

Processes constructed as $mooreSY_m(f, g, s_0)$ are translated into a finite state machine as illustrated in Fig. 9. The FSM consists of two combinational blocks and registers. The next state decoder is implemented by a combinational block with m inputs, where $\mathcal{H}(f)$ is read as the hardware implementation of the combinational function f. The output decoder is implemented by another combinational block that implements $\mathcal{H}(g)$. Finally, there is a register that has the size of the data type of s_0 that contains the reset state s_0.

The process $mooreSY_m(f, g, s_0)$ is implemented as a software process that has a local state and m input parameters. Inside this process, a software function $\mathcal{S}(f)$, the software interpretation of f, takes the input parameters and the current state and calculates the new state, while the function $\mathcal{S}(g)$ takes the current state and calculates the output value.

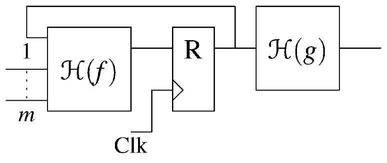


Fig. 9. Hardware semantics of $mooreSY_m$.

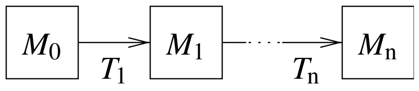


Fig. 10. Transformational refinement.

If multiple operations are defined in the system model, the hardware semantics imply that multiple resources also have to be generated in the implementation. The decision if resources are to be shared can be taken during the refinement process. This is in contrast to SAFL [30], where two equally named operations in the specification are mapped into one single resource in the implementation.

V. 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 M_0 is stepwise refined by well-defined design transformations T_i into a final implementation model M_n (Fig. 10). Only, at this late stage, the implementation model is translated using the ForSyDe hardware and software semantics into a synthesizable implementation description.

Definition 4 (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 $R(\mathrm{PN}_1) = \mathrm{PN}_2$ or $\mathrm{PN}_1 \xrightarrow{R} \mathrm{PN}_2$.

Definition 5 (Transformation): A transformation $T(M_1, \mathrm{PN}_1, R)$ is a functional mapping of a system model M_1 onto another system model M_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 M_1 is replaced by $R(\mathrm{PN}_1)$ to yield M_2. A transformation is denoted by $T(M_1, \mathrm{PN}_1, R) = M_2 = M_1[R(\mathrm{PN}_1)/\mathrm{PN}_1]$ or $M_1 \overset{T(\mathrm{PN1})}{\longrightarrow} M_2 = M_1[R(\mathrm{PN}_1)/\mathrm{PN}_1]$, 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 6 (Characteristic Function): The characteristic function $\mathcal{F}_{\mathrm{PN}}(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}, j)$ of a process network PN with the input signals $\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}$ and the output signals $\overrightarrow{o_1}, \ldots, \overrightarrow{o_n}$ expresses the dependence of the output events at tag j on the input signals

$$\mathcal{F}_{\mathrm{PN}}(\overrightarrow{i_1}, \ldots, \overrightarrow{i_m}, j) = \Big(E(\overrightarrow{o_1}, j), \ldots, E(\overrightarrow{o_m}, j)\Big).$$

The characteristic function of a process network PN is often expressed indirectly by the *characteristic tag function* $\mathcal{T}_{\mathrm{PN}}$ and the *characteristic value function* $\mathcal{V}_{\mathrm{PN}}$.

$$\mathcal{T}_{\mathrm{PN}}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j) = \Big(T(\overrightarrow{o_1},j),\ldots,T(\overrightarrow{o_n},j)\Big)$$
$$\mathcal{V}_{\mathrm{PN}}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j) = \Big(V(\overrightarrow{o_1},j),\ldots,V(\overrightarrow{o_m},j)\Big)$$

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 $zipWithSY_m$.

$$\mathcal{T}_{mapSY(f)}(\overrightarrow{i},j) = T(\overrightarrow{i},j)$$
$$\mathcal{V}_{mapSY(f)}(\overrightarrow{i},j) = f\Big(V(\overrightarrow{i},j)\Big)$$
$$\mathcal{T}_{zipWithSY_m(f)}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j) = T(\overrightarrow{i_1},j) = \cdots = T(\overrightarrow{i_m},j)$$
$$\mathcal{V}_{zipWithSY_m(f)}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j) = f\Big(V(\overrightarrow{i_1},j),\ldots,V(\overrightarrow{i_m},j)\Big)$$

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

$$\mathcal{T}_{delaySY_k(s_0)}(\overrightarrow{i},j) = T(\overrightarrow{i},j)$$
$$\mathcal{V}_{delaySY_k(s_0)}(\overrightarrow{i},j) = \begin{cases} s_0, & \text{if } j < k \\ V(\overrightarrow{i},j-k), & \text{otherwise.} \end{cases}$$

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

$$\mathcal{T}_{mooreSY_m(f,g,s_0)}$$
$$= T(\overrightarrow{i_1},j) = \cdots = T(\overrightarrow{i_m},j)$$
$$\mathcal{V}_{mooreSY_m(f,g,s_0)}$$
$$= \begin{cases} g(s_0) & j=0 \\ g\Big(f\Big(V(\overrightarrow{i_1},0),\ldots,V(\overrightarrow{i_m},0),s_0\Big)\Big) & j=1 \\ \vdots & \vdots \end{cases}$$

We classify transformations into *semantic preserving* and *design decision* according to the following definitions.

Definition 7 (Semantic Preserving Transformation): A transformation $M_1 \overset{T(\mathrm{PN}_1)}{\longrightarrow} M_2$ is semantic preserving, if $\mathcal{F}_{M_1}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j) = \mathcal{F}_{M_2}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j)$.

Definition 8 (Design Decision): A transformation $M_1 \overset{T(\mathrm{PN}_1)}{\longrightarrow} M_2$ is a design decision, if $\mathcal{F}_{M_1}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j) \neq \mathcal{F}_{M_2}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j)$.

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 m inputs. If the process has a regular structure such as an m-input adder or multiplier, where $\mathrm{m} = 4, 8, 16, \ldots$ the process can be transformed into a balanced network of $\mathrm{m}-1$ 2-input processes. This transformation $BalancedTree$ is defined in the transformation library as

Transformation Rule : $BalancedTree$

Original Process Network :

$$\mathrm{PN}_1(\overrightarrow{i_1},\ldots,\overrightarrow{i_m}) = \rhd_m(f)(\overrightarrow{i_1},\ldots,\overrightarrow{i_m})$$
$$m \qquad = 2^k; k \in \mathbb{N} > 1$$
$$f(x_1,\ldots,x_m) \qquad = x_1 \otimes \cdots \otimes x_m; \otimes \text{ is associative}$$

Transformed Process Network :

$$\begin{aligned}\mathrm{PN}_2(\overrightarrow{i_1},\ldots,\overrightarrow{i_m}) = \rhd_2(g)\Big(\ldots\Big(&\rhd_2(g)(\overrightarrow{i_1},\overrightarrow{i_2}),\\ &\rhd_2(g)(\overrightarrow{i_3},\overrightarrow{i_4})\Big),\\ \ldots\Big(&\rhd_2(g)(\overrightarrow{i_{m-3}},\overrightarrow{i_{m-2}}),\\ &\rhd_2(g)(\overrightarrow{i_{m-1}},\overrightarrow{i_m})\Big)\Big)\end{aligned}$$
$$g(x,y) \qquad = x \otimes y$$

Implication :

$$\mathcal{F}_{\mathrm{PN}_1}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j) = \mathcal{F}_{\mathrm{PN}_2}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j) \quad \forall j \in \mathbb{N}_0.$$

This transformation can be used for all processes that comply to the format and constraints given in the original process network, here, multiple 2^k-input processes, where the operator $\otimes$ is associative. From the Implication we see that $BalancedTree$ 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.

Transformation Rule : $PipelinedTree$

Original Process Network :

$$\begin{aligned}\mathrm{PN}_1(\overrightarrow{i_1},\ldots,\overrightarrow{i_m}) = \rhd_2(g_{m-1})&\Big(\ldots\Big(\rhd_2(g_1)(\overrightarrow{i_1},\overrightarrow{i_2}),\ldots\Big)\Big),\\ &\ldots\Big(\rhd_2\Big(\ldots,\rhd_2\big(g_{\frac{m}{2}}\big)\big(\overrightarrow{i_{m-1}},\overrightarrow{i_m}\big)\Big)\Big)\end{aligned}$$
$$m \qquad = 2^k; k \in \mathbb{N} > 1$$

Transformed Process Network :

$$\begin{aligned}\mathrm{PN}_2(\overrightarrow{i_1},\ldots,\overrightarrow{i_m}) = \triangle_1(s_0) \circ \rhd_2(g_{m-1})&\\ \Big(\ldots\Big(\triangle_1(s_0) \circ \rhd_2(g_1)(\overrightarrow{i_1},\overrightarrow{i_2}),\ldots\Big),&\\ \ldots\big(\ldots,\triangle_1(s_0) \circ \rhd_2\big(g_{\frac{m}{2}}\big)&\\ (\overrightarrow{i_{m-1}},\overrightarrow{i_m})\big)\Big)&\end{aligned}$$

Implication :

$$\mathcal{F}_{\triangle_k(s_0)\circ \mathrm{PN}_1}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j) = \mathcal{F}_{\mathrm{PN}_2}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j); \forall j \geq k.$$

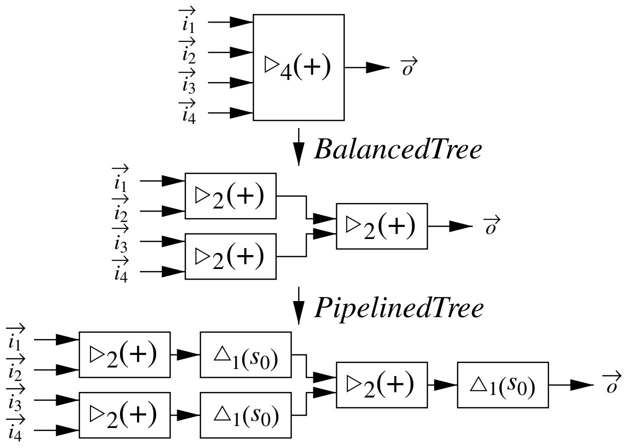


Fig. 11. Transformation into balanced pipelined structure.

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 Fig. 11, where a 4-input addition process is transformed into a pipelined structure.

A direct translation of a computation 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 subdomains the transformation $SerialClockDomain$ 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 Fig. 12 and formally given below is very efficient, if there are identical operations which can be shared.

Transformation Rule : $SerialClockDomain$

Original Process Network :

$$\begin{aligned} \mathrm{PN}_1(\overrightarrow{i_1},\ldots,\overrightarrow{i_m}) &= \triangleright_m(f)(\overrightarrow{i_1},\ldots,\overrightarrow{i_m}) \\ f(x_1,\ldots,x_m) &= g_{m-1}\left(h_m(x_m),(\ldots,(g_1\left(h_2(x_2),\right.\right. \\ &\qquad\qquad h_1(x_1)))\ldots)) \end{aligned}$$

Transformed Process Network :

$$\begin{aligned} \mathrm{PN}_2(\overrightarrow{i_1},\ldots,\overrightarrow{i_m}) &= (downDI(m)\circ P_{FSM}\circ p2sDI(m)) \\ &\quad (\overrightarrow{i_1},\ldots,\overrightarrow{i_m}) \\ P_{FSM} &= mooreSY(f',g',s_0) \\ f'(x,(j,s)) &= \begin{cases} (1,h_j(x)) & j=0 \\ (j+1,g_j\left(h_j(x),s\right)) & 0<j<m-1 \\ (0,g_j\left(h_j(x)+s\right)) & j=m-1 \end{cases} \\ g'(j,s) &= \begin{cases} s & j=0 \\ \perp & 0<j<m \end{cases} \end{aligned}$$

Implication :

$$\mathcal{F}_{\triangle_1(s_0)\circ\mathrm{PN}_1}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j) = \mathcal{F}_{\mathrm{PN}_2}(\overrightarrow{i_1},\ldots,\overrightarrow{i_m},j)$$

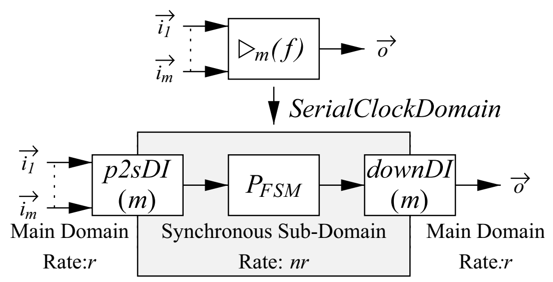


Fig. 12. Transformation into FSM using two clock domains.

The transformed process network works as follows. During an input event cycle the domain interface $p2sDI(m)$ (parallel to serial) reads all input values at event rate r and outputs them at event 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 0 the first input value (operand x_1) is stored as intermediate value s. In the $m-1$ following states, a function g_j is applied to the new input value (x_j) and 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 $downDI(m)$ down-samples the input signal to the event rate r and outputs only each mth input value starting with tag 0, thus suppressing the absent values from the output of P_{FSM}.

As domain interfaces can be characterized by a characteristic function, it means, though not shown here, that 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 Fig. 11, where $h_j(x)$ is the identity function and $g_j(x,y)$ is an add operation, resulting in a circuit with two clock domains using a single adder.

VI. Case Study: A Digital Equalizer

The following case study illustrates system modeling and design refinement inside the ForSyDe methodology by means of a digital equalizer. Our specification model is based on the original MASCOT specification [21], where the control flow parts are expressed in SDL and the data flow parts in Matlab.

The task of the equalizer is to adjust an audio signal $\overrightarrow{AudioIn}$ according to the current bass and treble levels. These levels are adjusted with four control buttons, which are modeled by means of the control signals $\overrightarrow{BassDn}$, $\overrightarrow{BassUp}$, $\overrightarrow{TrebleDn}$ and $\overrightarrow{TrebleUp}$. In addition the equalizer shall assure that the bass level of the output signal $\overrightarrow{AudioOut}$ does not exceed a predefined threshold in order to prevent damage to the speakers.

We have developed a hierarchical specification model as illustrated in Fig. 13.

Based on Fig. 13, we discuss the specificaton model of the equalizer in Section VI-A and refinement in Section VI-B. The boxed numbers point out the locations for refinement.

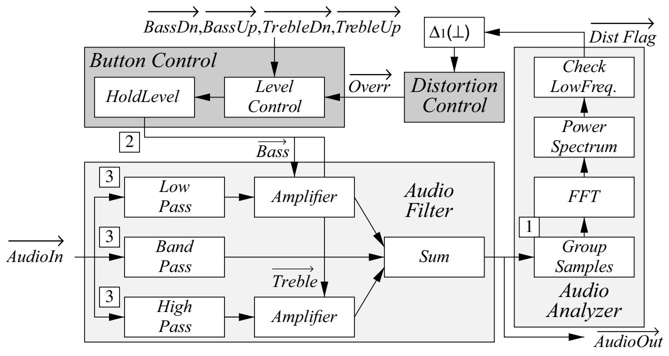


Fig. 13. Specification model of the equalizer.

A. Specification Model of the Equalizer

At the highest level, the equalizer is composed of four subsystems. The *Button Control* subsystem monitors the button inputs and the override signal from the subsystem *Distortion Control* and adjusts the current bass and treble levels. This information is passed to the subsystem *Audio Filter*, which receives the audio input, and filters and amplifies the audio signal according to the current bass and treble levels. This signal, the output signal of the equalizer, is analyzed by the *Audio Analyzer* subsystem, which determines whether the bass exceeds a predefined threshold. The result of this analysis is passed to the subsystem *Distortion Control*, which decides if a minor or major violation is encountered and issues the necessary commands to the *Button Control* subsystem.

This level of the equalizer is expressed as a set of equations. The insertion of a delay process is necessary since the feedback needs an initial value to stabilize. Zero-delay feedback loops are critical in synchronous specifications. The *modeling guidelines* of ForSyDe forbid the designer to use zero-delay feedback loops, which is the same approach as taken in the synchronous languages Lustre. Other approaches to address the zero-delay feedback problem are discussed in [48]. Esterel uses a unique fixed-point and Signal a relational approach. Since ForSyDe at present is embedded in Haskell, the semantics of Haskell prohibit a more advanced treatment of zero-delay feedback loops.

$$Equalizer(\overrightarrow{BassDn}, \overrightarrow{BassUp}, \overrightarrow{TrebleDn}, \overrightarrow{TrebleUp}, \overrightarrow{AudioIn}) = \overrightarrow{AudioOut}$$

where

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

The subsystems *Button Control* and *Distortion Control*, are control dominated (dark shaded in Fig. 13), while the *Audio Filter* and the *Audio Analyzer* are data flow dominated subsystems (light shaded). We exemplify the modeling of data flow oriented subsystems by the *Audio Filter*, and of control flow oriented subsystems by the *Distortion Control*.

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

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

where

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

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

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

We model the FIR-filter (shown in Fig. 14), a composition of a shift register with parallel outputs $(\mathrm{SIPO}(k{+}1, 0))$ which captures the current state of the filter and a combinational process $InnerProd(h)$ that calculates the inner product of the outputs of the shiftregister and the coefficient vector h

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

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

$$\mathrm{SIPO}(n, s_0) = unzipxSY_n \circ scandSY_1(shiftr, [s_0, \ldots, s_0])$$

$scandSY_1(shiftr, [v, \ldots, v])$ creates a process which models a shiftregister with a vector of size n, where all elements have the initial value s_0. However, since we use $scandSY_1$, the output of the shiftregister is a signal of a vector with n elements, which has to be converted into n parallel signals. This conversion is done be the process $unzipxSY_n$. The process $InnerProd(h)$ has the coefficient vector h as paramenter. It is modeled with the process constructor $zipWithSY_{k+1}$. The supplied function $ipV(h)$ calculates the inner product of the coefficient vector and the given state vector

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

The band pass filter in the equalizer is expressed as

$$\begin{aligned}
BandPass = \mathrm{FIR}\,([&0.063,\ 0.081,\ 0.095,\ 0.104,\ 0.107,\\
&0.104, l0.095,\ 0.081,\ 0.063]).
\end{aligned}$$

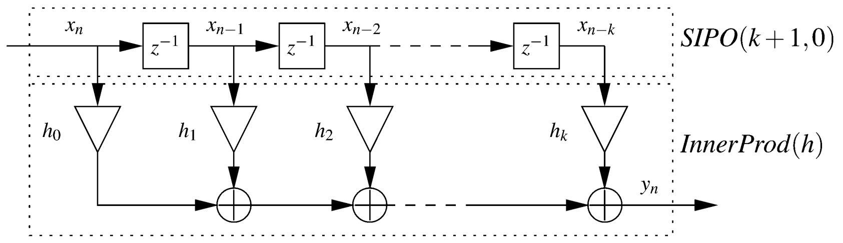


Fig. 14. FIR-filter.

We develop the *Distortion Control* directly from the SDL-specification, that has been used for the MASCOT-model [21]. The specification is shown in Fig. 15.

The *Distortion Control* is a single FSM, which can be modeled by means of the process constructor $mealySY$. The global state is not only expressed by the explicit states—Passed, Failed, and Locked—but also by means of the variable cnt. The state machine has two possible input values, Pass and Fail, and three output values, Lock, Release, and CutBass. It takes two functions, ns to calculate the next state, and out to calculate the output. The state is represented by a pair of the explicit state and the variable cnt. The initial state is the same as in the SDL-model, given by the tuple (Passed, 0). The ns function uses pattern matching. Whenever an input value matches a pattern of the ns function the corresponding right hand side is evaluated, giving the next state. An event with an absent value leaves the state unchanged. The output function is modeled in a similar way. The output is absent, when no output message is indicated in the SDL-model.

The ForSyDe model for the *Distortion Control* is given below.

$$DistortionContol = mealySY_1(ns, out, \text{Passed})$$

where

$$ns(st, cnt, inp) = \begin{cases} (st, cnt) & \text{if } inp = \perp \\ (\text{Passed}, cnt) & \text{if } st = \text{Passed} \wedge inp = \text{Pass} \\ (\text{Failed}, \text{Lim}) & \text{if } st = \text{Passed} \wedge inp = \text{Fail} \\ (\text{Locked}, cnt) & \text{if } st = \text{Failed} \wedge inp = \text{Pass} \\ (\text{Failed}, cnt) & \text{if } st = \text{Failed} \wedge inp = \text{Fail} \\ (\text{Failed}, \text{Lim}) & \text{if } st = \text{Locked} \wedge inp = \text{Fail} \\ (\text{Passed}, \text{cnt} - 1) & \text{if } st = \text{Locked} \wedge inp \\ & \quad = \text{Pass} \wedge cnt = 1 \\ (\text{Locked}, \text{cnt} - 1) & \text{if } st = \text{Locked} \wedge inp \\ & \quad = \text{Pass} \wedge cnt \neq 1 \end{cases}$$

$$out(st, cnt, inp) = \begin{cases} \text{Lock} & \text{if } st = \text{Passed} \wedge inp = \text{Fail} \\ \text{CutBass} & \text{if } st = \text{Failed} \wedge inp = \text{Fail} \\ \text{Release} & \text{if } st = \text{Locked} \wedge inp = \text{Pass} \wedge cnt = 1 \\ \perp & \text{otherwise} \end{cases}$$

B. Refinement of the Equalizer

In this section, we use the equalizer model to discuss three refinement techniques as indicated in Fig. 13.

1) refinement of the Clock Domain;
2) communication Refinement;
3) resource Sharing.

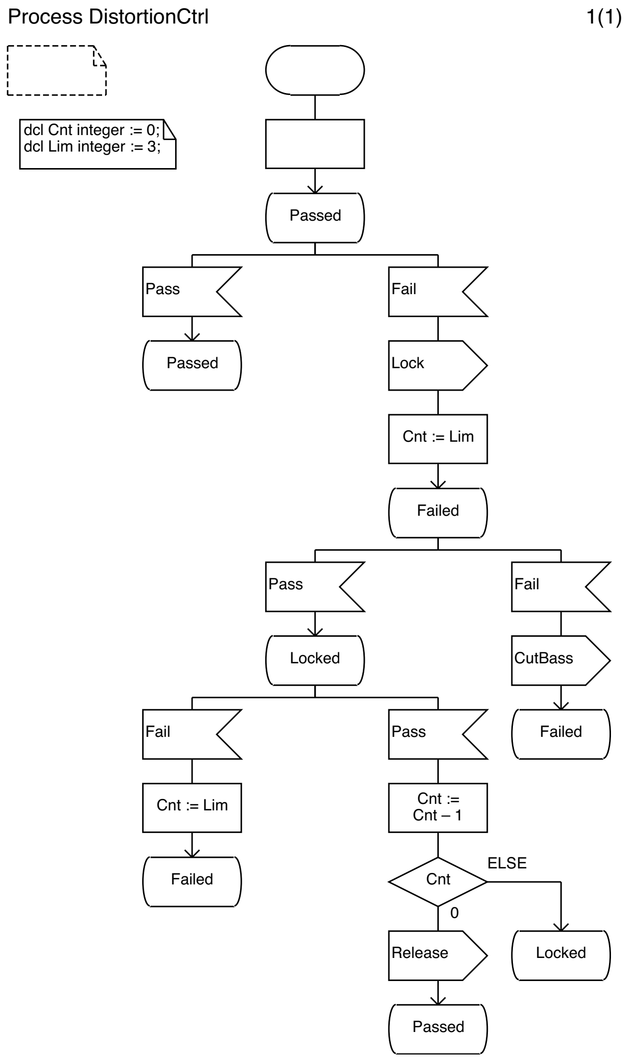


Fig. 15. SDL-description of distortion control.

In addition, ForSyDe includes data type and memory refinement, which are beyond the scope of this paper.

1) Refinement of the Clock Domain: Fig. 16 shows the *Audio Analyzer* subsystem, which includes a Fast-Fourier Transform (FFT) algorithm. This function FFT takes a vector of $n = 2^k$ samples and produces the corresponding FFT result in form of a vector of size n that is denoted as $[x_0, x_1, \ldots, x_{n-1}]$. The FFT algorithm is used to determine the frequency spectrum of a signal. It is implemented in the process $\text{FFT}(Q_\perp(n))$. The process *Power Spectrum* $(R_\perp)$ calculates the power spectrum. The process *Check Low Frequencies* $(S_\perp)$ analyzes if the power of the low frequencies exceeds a threshold

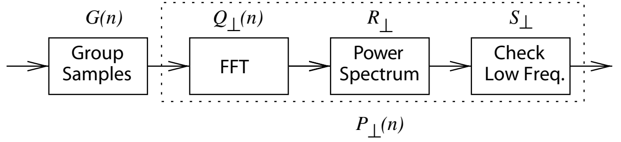


Fig. 16. Audio analyzer.

and issues a warning in this case, which is sent to the *Distortion Control*. The process *Group Samples* $(G(n))$ reads n samples and groups them into a vector of size n. The computation of this vector takes n event cycles and serves as input for the FFT. However, since we use a synchronous computational model in the specification model the grouping process has to produce an output event for each input event. This results in the output of $n-1$ absent values $(\perp)$ for each output of a grouped vector. $G(n)$ is based on the process constructor $mooreSY$ and formally defined as

$$G(n) = mooreSY_m(f, g, s_0)$$

where

$$s_0 = []$$
$$f(x,s) = \begin{cases} [x], & \text{if } \#s = n \\ \text{concat}(x,s), & \text{otherwise} \end{cases}$$
$$g(s) = \begin{cases} s, & \text{if } \#s = n \\ \perp, & \text{otherwise.} \end{cases}$$

Since all processes $Q_\perp(n)$, $R_\perp$, $S_\perp$ are constructed with the combinational process constructor $mapSY$ we can replace them by

$$P_\perp(n) = mapSY\,(\Psi\,(p(n)))$$
$$\text{where } p(n) = s \circ r \circ q(n).$$

We designate the specification model of the *Audio Analyzer* as $\mathrm{PN} = P_\perp \circ G(n)$

$$\mathcal{T}_{\mathrm{PN}}(\overrightarrow{i}, j) = T(\overrightarrow{i}, j)$$
$$\mathcal{V}_{\mathrm{PN}}(\overrightarrow{i}, j) = \begin{cases} p\,([])\,, & \text{if } j=0 \\ p\left(\left[V(\overrightarrow{i}, j-n),\right.\right. & \\ \quad \left.\left.\ldots, V(\overrightarrow{i}, j-1)\right]\right), & \text{if } \exists k\in\mathbb{N}. j=kn \\ \perp, & \text{otherwise.} \end{cases}$$

The process $P_\perp(n)$ has to process all absent values. While this is not a drawback for the specification phase, a direct implementation as shown in Fig. 17 can make no use of the fact, that the FFT has only to be calculated at each nth clock cycle.

Such an implementation will be very slow, since the computation of the FFT function is clearly the most time consuming one and will determine the overall system performance.

In order to get a more efficient specification, the ForSyDe methodology allows to introduce synchronous subdomains into the system model during the refinement process.

Using the special characteristic of the grouping process $G(n)$ we can derive the identity

$$G(n) = upDI(n) \circ downDI(n) \circ G(n)$$

and, finally,

$$P_\perp(n) \circ G(n) = upDI(n) \circ P(n) \circ downDI(n) \circ G(n)$$

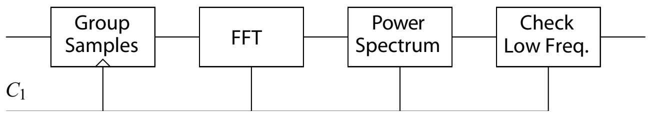


Fig. 17. Direct implementation of the audio analyzer.

which can be used as a refined model

$$\mathrm{PN}_{\mathrm{ref}} = upDI(n) \circ P(n) \circ downDI(n) \circ G(n).$$

The original process $P_\perp$ is replaced by $P(n)$ since $downDI(n) \circ G(n)$ does not produce any absent values. The identity $\mathrm{PN} = \mathrm{PN}_{\mathrm{ref}}$ can be proven by the characteristic function of $\mathrm{PN}_{\mathrm{ref}}$, but is not given here.

Analyzing $\mathrm{PN}_{\mathrm{ref}}$ we conclude that the process $P(n)$ processes events only at each nth tag and thus can be implemented with a slower clock. Based on these considerations we define the transformation $GroupToMultiRate$, that introduces a synchronous subdomain and can be applied on all processes of the form $P_\perp(n) \circ G(n)$.

Transformation Rule : $GroupToMultiRate$

Original Process Network :

$$\overrightarrow{o} \quad = \mathrm{PN}(\overrightarrow{i})$$
$$\mathrm{PN}(\overrightarrow{i}) = (mapSY\,(\Psi(f)) \circ groupSY(k))\,(\overrightarrow{i})$$

Transformed Process Network :

$$\overrightarrow{o} \quad = \mathrm{PN}'(\overrightarrow{i})$$
$$\mathrm{PN}'(\overrightarrow{i}) = (upDI(k) \circ mapSY(f) \circ downDI(k)$$
$$\circ groupSY(k))\,(\overrightarrow{i})$$

Implication :

$$\mathcal{F}_{\mathrm{PN}}(\overrightarrow{i}, j) = \mathcal{F}_{\mathrm{PN}'}(\overrightarrow{i}, j).$$

We implement the synchronous subdomain with a clock frequency f_{C_2} that is $n = 2^k$ times slower than the clock frequency f_{C_1} of the main synchronous domain. The implementation of this transformation of the *Audio Analyzer* is illustrated in Fig. 18.

2) Communication Refinement: The specification model uses the same synchronous communication mechanism between all its subsystems. This is a nice feature for modeling and analyzing, since partitioning issues and special interfaces between subsystems have not to be taken into account in this phase. However, large systems are usually not implemented as one single unit, but are partitioned into hardware and software blocks communicating with each other via a dedicated communication protocol. The ForSyDe methodology offers transformations of a synchronous communication into other protocols. Looking at the equalizer example, we observe, that the aperiodic data rate of the *Button Control* and the *Distortion Control* subsystem is much lower than the data rate of the *Audio Filter* and *Audio Analyzer*. We decide to implement the *Button Control* and *Distribution Control* in software and the *Audio Filter* and *Audio Analyzer* in hardware. For the communication between these parts we implement a handshaking protocol with *Send* and *Receive* processes.

We focus on the refinement of the synchronous interface between the *Button Control* and the *Audio Filter* subsystems,

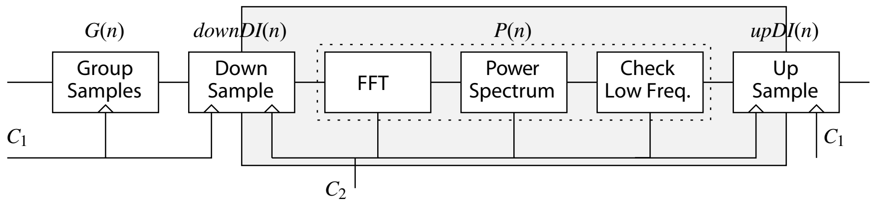


Fig. 18. Audio analyzer after refinement.

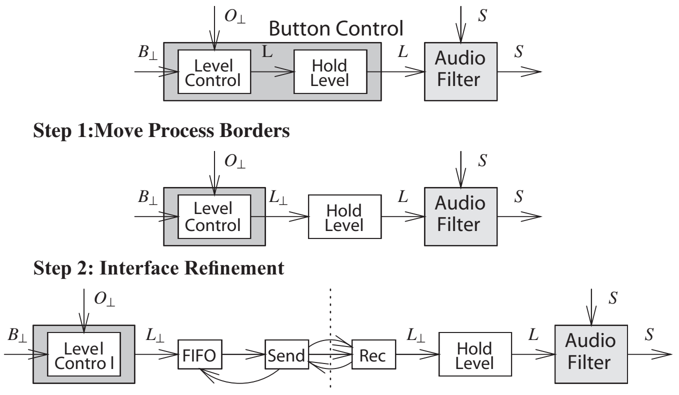


Fig. 19. Refinement into a handshake protocol.

which is shown in Fig. 19. The figure shows the data types of the signals. Please note, that all the data types $B_\perp$, $O_\perp$, $L_\perp$ are extended data types, containing absent values. Such signals have a lower data rate as the corresponding signals with the data types B, O, L since they do not carry a value in each event cycle. The block *Hold Level* outputs the last present value, when receiving an absent value.

The refinement is done in two steps. First, we move the block *Hold Level* out of the subsystem *Button Control* in order to implement the interface between the block *Level Control* and the block *Hold Level* since this communication channel has a significantly lower data rate as expressed by the data type of the signal $(L_\perp)$. The second step is to refine the interface into a handshake protocol. We do this by the transformation of the channel between *Level Control* and *HoldLevel* by means of the transformation $ChannelToHandshake$ which is given below.

Transformation Rule : $ChannelToHandshake$

Original Process Network :

$$\vec{o} = \vec{i} \quad \text{where } sigtype(\vec{o}) = sigtype(\vec{i}) = V_\perp$$

Transformed Process Network :

$$\vec{o} = \mathrm{PN}'(\vec{i})$$

where

$$\begin{aligned}(\vec{o}, \vec{r_m}) &= receive(\vec{s_o}, \vec{s_d})\\ (\vec{s_m}, \vec{s_d}, \vec{s_f}) &= send(\vec{f_o}, \vec{r_m})\\ \vec{f_o} &= fifo(\vec{i}, \vec{s_f})\end{aligned}$$

Implication :

see text in paper

The transformation introduces a FIFO, a *Send* and a *Receive* process. When *Send* is idle, it tries to read data from the FIFO on $\vec{s_f}$. Then it sends the message `DataReady` on $\vec{s_m}$ to the *Receiver* and after receiving the message `Ready` on $\vec{r_m}$, it sends the data on $\vec{s_d}$. The *Receiver* sends a message `Ack` on $\vec{r_m}$, when the data is received.

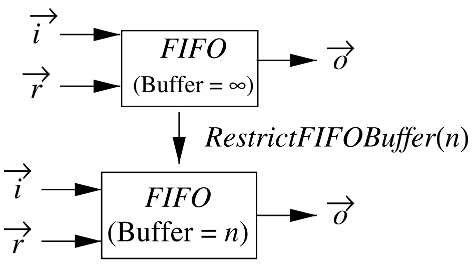


Fig. 20. Refinement of FIFO buffer.

The handshake protocol implies a delay of several cycles for each event, as *Send* and *Receive* are synchronous processes. This means that the timing behavior of the refined interface is different from the original interface. This also means that the *Audio Filter* will not process exactly the same combination of values in each event cycle as in the system model.

These consequences have to be taken into account, when interfaces are refined. In this case, it can be shown that the refined interface still behaves in practice as the system model, if we make two assumptions.

1) The average data rate of the block *Level Control* is much lower than the data rate of the *Audio Filter*. If the FIFO is correctly dimensioned, there will be no buffer overflow in the FIFO and all values reach the *Audio Control* after a small number of event cycles.
2) The output function of the *Audio Filter* does not change significantly if the input signals of the *Level Control* are delayed. That is clearly the case, as a small delay of the level signal only delays the change of the amplitude for the same small time, but does not effect the signals shape.

These assumptions point to obligations on other design activities. A further formalization of the design decisions will allow to make all assumptions and obligations explicit. The FIFO buffers have to be dimensioned sufficiently large based on a separate analysis. This will imply a further design decision transformation as illustrated in Fig. 20. Assumptions about the environment and the application, such as the kind of expected input signal, in this case the data rate, have to be validated to justify the applied design decisions.

We can now synthesize the interface using the hardware semantics of ForSyDe. The sole purpose of our transformation is to *prepare* for an asynchronous implementation. Note, however, that the model we have derived is not truly asynchronous in the sense that it is still completely *deterministic* without nondeter-

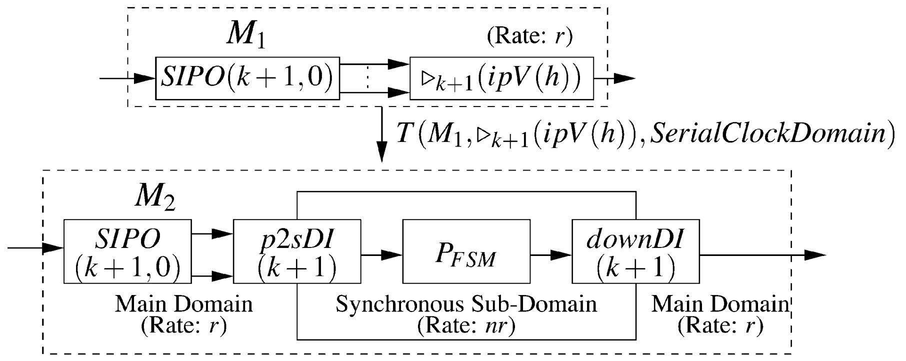


Fig. 21. Transformation of a FIR-filter.

ministic channel delays. Of course, the channel can be modeled more realistically, if desired. In the ForSyDe methodology, we suggest to avoid a nondeterministic model but to use a stochastic channel model instead, which ForSyDe supports with stochastic process constructors [49].

3) Resource Sharing: Fig. 21 shows the application of the design decision $SerialClockDomain$ on the FIR-filter of Fig. 14. Since the process $ipV(h)$ is defined as

$$(ipV(h))\,(x_0, \ldots, x_n) = h_0 x_0 + \cdots + h_n x_n.$$

It complies to the input process network format of the transformation rule $SerialClockDomain$, where

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

We can use this rule to apply the transformation $T(M_1, zipWithSY_{k+1}(ipV(h)), SerialClockDomain)$ on the FIR-filter model M_1 in order to receive a model M_2, where $\mathrm{SIPO}(k+1)$ remains unchanged and the FIR-filter is realized with two clock domains and only one multiplier and one adder (Fig. 21).

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 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 (10 482 gates).

VII. Conclusion

This article presents the modeling and transformational refinement technique of the ForSyDe methodology. Due to a carefully chosen computational model and a modeling technique that is based on well-defined process constructors ForSyDe allows the refinement of the specification model into a more efficient implementation model through the stepwise application of formally defined design transformations.

By 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 design decision. Each transformation rule is well defined by the original process network and the transformed process network. Each rule also shows the consequences for the design by an implication part, expressed with the characteristic function.

The article shows the potential of ForSyDe. Traditional and powerful synthesis techniques can now be formulated as transformation rules and applied inside the functional domain. By selecting transformation rules from the transformation library, the designer is able to perform a transparent and documented refinement process inside the functional domain.

In order to apply ForSyDe on larger applications, the practical framework together with additional transformation rules and transformation strategies has to be developed. Future research will show to what extent ForSyDes transformational approach is able to generate implementation models that contain the necessary details to result in an effective implementation. We believe that at least for application-specific areas it is possible to develop a sufficient number of transformation rules, which will lead to an improved and transparent design process.

References

[1] 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 Trans. Computer-Aided Design.*, vol. 19, pp. 1523–1543, Dec. 2000.

[2] D. B. Skillicorn and D. Talia, "Models and languages for parallel computation," *ACM Comput. Surv.*, vol. 30, no. 2, pp. 123–169, 1998.

[3] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli, "Design of embedded systems: Formal models, validation, and synthesis," *Proc. IEEE*, vol. 85, pp. 366–390, March 1997.

[4] E. A. Lee and A. Sangiovanni-Vincentelli, "A framework for comparing models of computation," *IEEE Trans. Computer-Aided Design*, vol. 17, pp. 1217–1229, Dec. 1998.

[5] A. Benveniste and G. Berry, "The synchronous approach to reactive and real-time systems," *Proc. IEEE*, vol. 79, pp. 1270–1282, Sept. 1991.

[6] E. A. Lee and T. M. Parks, "Dataflow process networks," *IEEE Proc.*, vol. 83, pp. 773–799, May 1995.

[7] G. Kahn, "The semantics of a simple language for parallel programming," in *Proc. IFIP Congress*, North-Holland, Holland, 1974, pp. 993–998.

[8] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proc. IEEE*, vol. 75, pp. 1235–1245, Sept. 1987.

[9] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, A. Sangiovanni-Vincentelli, E. M. Sentovich, and K. Suzuki, "Synthesis of software programs for embedded control applications," *IEEE Trans. Computer-Aided Design*, vol. 18, pp. 834–849, June 1999.

[10] N. Halbwachs, *Synchronous Programming of Reactive Systems*. Norwell, MA: Kluwer, 1993.

[11] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language LUSTRE," *Proc. IEEE*, vol. 79, pp. 1305–1320, Sept. 1991.

[12] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Marie, "Programming real-time applications with SIGNAL," *Proc. IEEE*, vol. 79, pp. 1321–1335, Sept. 1991.

[13] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Program.*, vol. 8, no. 3, pp. 231–274, 1987.

[14] G. Berry and G. Gonthier, "The Esterel synchronous programming language: Design, semantics, and implementation," *Sci. Comput. Program.*, vol. 19, no. 2, pp. 87–152, 1992.

[15] F. Maraninchi, "The Argos language: Graphical representation of automata and description of reactive systems," presented at the *IEEE Workshop on Visual Languages*, Kobe, Japan, Oct. 1991.

[16] N. Halbwachs and P. Raymond, "Validation of synchronous reactive systems: From formal verification to automatic testing," in *ASIAN'99, Asian Comput. Sci. Conf.*, Phuket, Thailand, Dec. 1999, pp. 1–12.

[17] H. Hsieh, F. Balarin, L. Lavagno, and A. Sangiovanni-Vincentelli, "Synchronous approach to functional equivalence of embedded system implementations," *IEEE Trans. Computer-Aided Design*, vol. 20, pp. 1016–1033, Aug. 2001.

[18] S. Thompson, *Haskell—The Craft of Functional Programming*, 2 ed. Reading, MA: Addison-Wesley, 1999.

[19] E. A. Lee, "What's ahead for embedded software," *IEEE Comput.*, vol. 33, pp. 18–26, Sept. 2000.

[20] W. Wu, I. Sander, and A. Jantsch, "Transformational system design based on a formal computational model and skeletons," in *Forum Design Lang.*, Tübingen, Germany, Sept. 2000, pp. 321–328.

[21] P. Bjuréus and A. Jantsch, "MASCOT: A specification and cosimulation method integrating data and control flow," in *Proc. Design and Test Eur. Conf.*, Paris, France, Mar. 2000, pp. 161–168.

[22] A. Jantsch and P. Bjuréus, "Composite signal flow: A computational model combining events, sampled streams, and vectors," in *Proc. Design Test Eur. Conf.*, Paris, France, Mar. 2000, pp. 154–160.

[23] J. II, M. Goel, C. Hylands, B. Kienhuis, E. A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay, and Y. Xiong, "Overview of the ptolemy project," Dept. Elect. Eng. Comput. Sci., Univ. California, Berkeley, Tech. Rep. UCB/ERL no. M99/37, 1999.

[24] R. Ernst, D. Ziegenbein, K. Richter, L. Thiele, and J. Teich, "Hardware/software codesign of embedded systems—the SPI workbench," in *Proc. IEEE Workshop VLSI*, Orlando, FL, 1999, pp. 9–17.

[25] L. Thiele, K. Strehl, D. Ziegenbein, R. Ernst, and J. Teich, "Funstate—an internal design representation for codesign," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, 1999, pp. 558–565.

[26] H. J. Reekie, "Realtime signal processing," Ph.D. dissertation, School of Elect. Eng., Univ. Technology Sydney, Sydney, Australia, 1995.

[27] W. Luk and T. Wu, "Toward a declarative framework for hardware-software codesign," in *Proc. 3rd Int. Workshop Hardware/Software Codesign*, 1994, pp. 181–188.

[28] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: Hardware design in Haskell," in *Proc. Int. Conf. Functional Program.*, 1998, pp. 174–184.

[29] R. Sharp and A. Mycroft, "A higher level language for hardware synthesis," in *Proc. 11th Adv. Res. Working Conf. Correct Hardware Design Verification Methods (CHARME)*, vol. 2144, LNCS, 2001, pp. 228–243.

[30] A. Mycroft and R. Sharp, "Hardware/software co-design using functional languages," in *Proc. Tools Algorithms the Construction Anal. Syst. (TACAS)*, vol. 2031, LNCS, 2000, pp. 236–251.

[31] J. Matthews, B. Cook, and J. Launchbury, "Microprocessor specification in HAWK," in *Proc. Int. Conf. Comput. Lang.*, 1998, pp. 90–101.

[32] Y. Li and M. Leeser, "HML, a novel hardware description language and its translation to VHDL," *IEEE Trans. VLSI Syst.*, vol. 8, pp. 1–8, Feb. 2000.

[33] R. Milner, M. Tofte, R. Harper, and D. MacQueen, *The Definition of Standard ML—Revised*. Cambridge, MA: MIT Press, 1997.

[34] H. A. Partsch, *Specification and Transformation of Programs*. New York: Springer-Verlag, 1990.

[35] A. Pettorossi and M. Proietti, "Rules and strategies for transforming functional and logic programs," *ACM Comput. Surv.*, vol. 28, no. 2, pp. 361–414, 1996.

[36] F. L. Bauer, B. Möller, H. Partsch, and P. Pepper, "Formal program construction by transformations—computer-aided, intuition guided programming," *IEEE Trans. Software Eng.*, vol. 15, pp. 165–180, Feb. 1989.

[37] J. Plosila, "Self-timed circuit design—The action systems approach," Ph.D. dissertation, Dept. Appl. Phys., Univ. Turku, Turku, Finland, 1999.

[38] T. Seceleanu, "Systematic Design of Synchronous digital circuits," Ph.D. dissertation, Dept. Comput. Sci., Univ. Turku, Turku, Finland, 2001.

[39] D. D. Gajski, N. D. Dutt, A. C.-H. Wu, and S. Y.-L. Lin, *High-Level Synthesis*. Norwell, MA: Kluwer, 1992.

[40] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. New York: McGraw-Hill, 1994.

[41] J. Voeten, "On the fundamental limitataions of transformational design," *ACM Trans. Design Automation Electron. Syst.*, vol. 6, no. 4, pp. 552–553, 2001.

[42] D. D. Gajski and L. Ramachandran, "Introduction to high-level synthesis," *IEEE Design Test Comput.*, vol. 11, pp. 44–54, Winter 1994.

[43] Z. Lu, I. Sander, and A. Jantsch, "A case study of hardware and software synthesis in ForSyDe," in *Proc. 15th Int. Symp. Syst. Synthesis*, Kyoto, Japan, Oct. 2002, pp. 86–91.

[44] I. Sander and A. Jantsch, "Formal system design based on the synchrony hypothesis," in *Proc. 12th Int. Conf. VLSI Design*, Goa, India, Jan. 1999, pp. 318–323.

[45] T. Grötker, S. Liao, G. Martin, and S. Swan, *System Design With SystemC*. Norwell, MA: Kluwer, 2002.

[46] I. Sander and A. Jantsch, "System synthesis based on a formal computational model and skeletons," in *Proc. IEEE Workshop VLSI*, Orlando, FL, Apr. 1999, pp. 32–39.

[47] ——, "System synthesis utilizing a layered functional model," in *Proc. 7th Int. Workshop Hardware/Software Codesign*, Rome, Italy, May 1999, pp. 136–140.

[48] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. De Simone, "The synchronous languages 12 years later," *Proc. IEEE*, vol. 91, pp. 64–83, Jan. 2003.

[49] A. Jantsch, I. Sander, and W. Wu, "The usage of stochastic processes in embedded system specifications," in *Proc. 9th Int. Symp. Hardware/Software Codesign*, Copenhagen, Denmark, Apr. 2001, pp. 5–10.

Ingo Sander (M'03) received the M.Sc. degree in electrical engineering from the Technical University of Braunschweig, Braunschweig, Germany, in 1990 and the Ph.D. degree from the Royal Institute of Technology, Stockholm, Sweden, in 2003.

Since 1993, he has been a Lecturer at the Royal Institute of Technology. His current research interests include system-level design methodologies and system-on-a-chip architectures.

Axel Jantsch (M'97) received the M.Sc. and Ph.D. degrees in computer science from the Technical University Vienna, Vienna, Austria, in 1988 and 1993, respectively.

In 1997, he became an Associate Professor and, since 2002, he is a Full Professor at the Royal Institute of Technology, Stockholm, Sweden. His research interests include system-level modeling, design and verification, and on-chip communication architectures.