Transformation Based Communication and Clock Domain Refinement for System Design

Ingo Sander
Royal Institute of Technology
Stockholm, Sweden
ingo@imit.kth.se

Axel Jantsch
Royal Institute of Technology
Stockholm, Sweden
axel@imit.kth.se

ABSTRACT

The ForSyDe methodology has been developed for system level design. In this paper we present formal transformation methods for the refinement of an abstract and formal system model into an implementation model. The methodology defines two classes of design transformations: (1) semantic-preserving transformations and (2) design decisions. In particular we present and illustrate communication and clock domain refinement by way of a digital equalizer system.

Categories and Subject Descriptors

B.7.2 [**Integrated Circuits**]: Design-Aids; J.6 [**Computer-Aided Engineering**]: Computer-Aided Design (CAD)

General Terms

Design, Theory

Keywords

System Design, System Modeling, Design Refinement

1. INTRODUCTION

The increasing capacity of integrated circuits makes it possible to integrate more and more functionality on a single chip. A SoC (System-on-a-Chip) architecture can include a variety of components, such as analog parts, micro controller cores, digital signal processor cores, memories, IP blocks and custom hardware. Software, running on a number of different processors, has to be designed to coordinate these components. While such architectures allow for totally new application areas, it is not obvious how to design such applications.

Keutzer et al. discuss system-level design in [7]. They 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 to use formal models and transformations in system design so that verification and synthesis can be applied to the advantage of the design methodology" and believe that "the most important point for functional specification is the underlying mathematical model of computation".

These arguments strongly support the ForSyDe (Formal System Design) 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.

The ForSyDe methodology addresses the design of SoC applications. Starting with a formal system 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, which serves as a starting point for synthesis into hardware and software.

In this paper we present the refinement process in our methodology. We discuss semantic-preserving transformations and design decisions and illustrate them by the refinement of an equalizer system.

2. RELATED WORK

Edwards et al. [5] use the tagged signal model, which is further elaborated by Lee and Sangiovanni-Vincentelli in [8], to classify and analyze several models of computation, in particular discrete event models, communicating finite state machines, synchronous 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" [2]. The synchronous assumption implies a total order of events and leads to a clean separation between computation and communication and gives a solid base for formal methods.

Balarin et al. [1] 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 GALS (globally asynchronous locally synchronous) approach and implement it in their methodology as a network of Codesign FSM's communicating via events. Each CFSM is a syn-

chronous FSM, but the communication is done with events that can occur 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 is a heavy burden put on the designer's shoulders. The task of developing and verifying a system model for a SoC application is so complex, that a system model has to be deterministic, thus avoiding the unnecessary complexity of nondeterminism. The fact, that SoC applications will be implemented on at least partly asynchronous architectures does not justify a nondeterministic approach. 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.

Approaches like the MASCOT methodology [4] which integrates data and control flow at the system specification level, using the two languages Matlab and SDL, are successful in that they provide an integrated simulation environment. They fail with respect to full integration because they cannot provide an integrated formal analysis and synthesis. For instance, all heterogeneous approaches advocate separate synthesis and design flows for the different domains.

To overcome this difficulty internal representations like FunState [14] have been developed to integrate a heterogeneous system model into one internal representation. However, when compiling high level, abstract and application oriented models into a combined internal representation based on common, low level primitives, most application oriented features are lost and with them optimization opportunities that rely on the meaning of these application oriented concepts.

Functional languages have been used in other research projects in electronic design. The parallel programming community has used functional languages to derive parallel programs from a functional specification [13]. They use skeletons to structure a problem. This formulation is then transformed using cost measures into an efficient implementation for a chosen computer architecture. Reekie [11] has used Haskell to model digital signal processing applications. Similarly to us he modeled streams as infinite lists and used higher-order functions to operate on them. Finally, semantic-preserving methods were applied to transform a model into a more efficient representation. This representation was not synthesized to hardware or software. Lava [3] is a hardware description language based on Haskell. It focuses on the structural representation of hardware and offers a variety of powerful connection patterns. Lava descriptions can be translated into VHDL and there exist interfaces to formal method tools. Hardware ML (HML) [9] is a hardware description language, that is based on the functional programming language Standard ML. 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.

3. THE FORSYDE METHODOLOGY

3.1 The Design Process

The ForSyDe design process starts with the development of a formal abstract functional system model, written in the functional language Haskell. This model is then refined inside the functional

domain by a stepwise application of well defined design transformations into an implementation model, which is discussed in Section 4. As the implementation model is a refined version of the system model, the same validation and verification methods can be applied to both models. In the partitioning phase, the implementation model is partitioned into hardware and software blocks, which are mapped on architectural components. Only now, in the code generation phase, we leave the functional domain to produce VHDL or C/C++ for the hardware and software parts. Code generation in our methodology is discussed in [12].

3.2 The System Model

The system model reflects the design principles of the ForSyDe methodology. In order to allow for formal design on a high abstraction level, the system 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 construction templates, called *skeletons*, which implement the synchronous computational model.

- It is expressed in the executable functional language Haskell, with a well defined *formal semantics*.

The system 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 4).

In order to formally describe our computational model, we follow the denotational framework of Lee and Sangiovanni-Vincentelli [8]. They define signals as a set of events, where each event e has a tag t and a value v, i.e. $e = (t, v) \in T \times V$. As our system model is synchronous, T is the set of natural numbers, and all signals have the same set of tags. In order to model the absence of an event, a data type D can be extended into a data type $D_\perp$ by adding the special value $\perp$, which is used to model the absence of a value. Absent values are used to establish a total order of events when dealing with signals with different or aperiodic event rates.
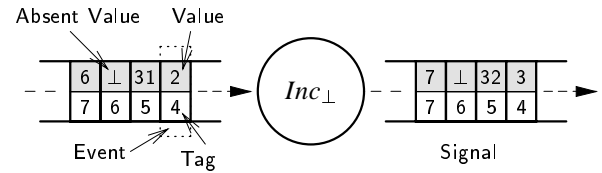

Figure 1: Modeling of Signals and Processes

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

We denote a signal s with $\{x_1, x_2, \dots\}$, where the tag is given by the position in the signal. To model a process we use the concept of skeletons as *process constructors*. A skeleton is a higher-order function like *mapSY*. *mapSY* takes a combinatorial function

as argument and produces a process. In this paper, we denote the processes created by *mapSY* and a function f as the name of the function in capital letters, in this case F. A process F is then created by $mapSY(f)$ and formally defined by

$$
\begin{aligned}
F &= mapSY(f) \\
F\{x_1, x_2, \dots\} &= \{f(x_1), f(x_2), \dots\}
\end{aligned}
$$

The process $Inc_\perp$ of Figure 1 is also modeled by means of the higher-order function *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 $f : D \to D$ and generates $f_\perp : D_\perp \to D_\perp$. Ψ is defined by

$$
\begin{aligned}
\Psi(f) &= f_\perp \\
\text{where} \quad f_\perp(x) &= \begin{cases} \perp & x = \perp \\ f(x) & \text{otherwise} \end{cases} \\
\forall f &: D \to D
\end{aligned}
$$

We use the functional language Haskell as our modeling language. Haskell fits well in our modeling technique since it is a functional language, that supports many of our concepts, such as higher-order functions and laziness, and has a formal semantics. Thus the implementation of higher-order functions like *mapSY* and Ψ is straight forward. We implement the synchronous computational model with skeletons, like *mapSY*. A skeleton is a higher-order function, that takes *combinatorial functions*, i.e. functions that have no internal state, and *values* as input and produces a process as output. Processes consume signals and generate signals as output. Hence, skeletons can be viewed as *process constructors*. The ForSyDe library defines skeletons for more than one computational model, but for the ForSyDe system model only *synchronous skeletons* are used.

The concept of skeletons give the following benefits:

- Due to the construction of processes by skeletons and combinatorial functions, we get a *clean separation between synchronization and computation*. Synchronization is expressed by skeletons and computation by the supplied combinatorial functions.

- There is a family of skeletons that include a *local state*. But there is no global state in the model, which would make it more difficult to reason about the system model.

- Skeletons have a *structural hardware and software interpretation*. This means, that a system model, which is composed of skeletons also has an interpretation in hardware, software or a mix of both.

- As skeletons are higher-order functions, the work on *correctness preserving transformations* [10] can be used to transform a system model inside the functional domain into a more effective implementation model [15].

The process $Inc_\perp$ illustrates the clean separation between synchronization and computation. The skeleton *mapSY* processes an event in each event cycle, while the function $inc_\perp$ computes the output value for each input value.

The hardware interpretation of this process is a combinatorial block implementing the function $inc_\perp$ with one input and one output. The software interpretation is the implementation of $inc_\perp$ as a software function taking one value as input and producing one output value.

The skeleton *mooreSY* is an example for a skeleton with a local state. It models a finite state machine of Moore type. The skeleton takes a function *ns* to calculate the next state as first argument, a function *out* to calculate the output as second argument and a value s_0 for the initial state as last argument. Thus the process

$$Moore = mooreSY(ns, out, init)$$

implements a finite state machine. In hardware the process *Moore* can be implemented as an FSM, where the next state decoder implements the function *ns*, the output decoder the function *out* and the memory elements store data based on the data type for s_0. The general hardware interpretation is visualized in Figure 2.
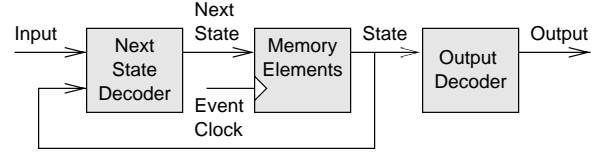

Figure 2: Hardware Interpretation of *mooreSY*

Functions can be glued together by function composition. A new function h is composed of two functions f and g by application of the composition operator $\circ$ according to

$$
\begin{aligned}
h &= f \circ g \\
h(x) &= f(g(x))
\end{aligned}
$$

As processes are functions function composition can also be used for process composition. We can also build a network of processes by expressing it as a set of equations. Using this technique we can express a hierarchical system model.

4. REFINEMENT OF THE SYSTEM MODEL

The system model is stepwise refined through the use of well defined design transformations from an initial specification model S_0 to a final implementation model S_n (Figure 3).
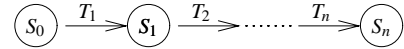

Figure 3: Refinement through Design Transformation

There are two classes of transformation techniques:

Semantic Preserving Transformations Semantic preserving transformations do not change the meaning of the model, i.e. the transformed model behaves in the same way as the original model. Semantic preserving transformations are mainly used to optimize the model for synthesis.

Design Decisions 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.

Before we illustrate the refinement methodology with the system model of the equalizer we want to point out, how powerful simple semantic-preserving transformations can be, as they are performed on a system model, that is a pure composition of functions. Figure 4 shows the power of the formal and functional approach. (1) Processes can easily be move over block borders and (2) two processes
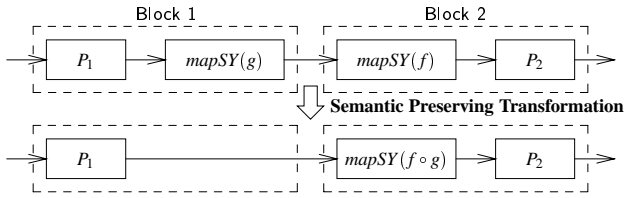
Figure 4: A Semantic Preserving Transformation

can be combined and possibly optimized by use of function composition. In this case we use the semantic-preserving transformation

$$mapSY(f) \circ mapSY(g) = mapSY(f \circ g) \tag{1}$$

Another useful semantic-preserving transformation is

$$\Psi(f) \circ \Psi(g) = \Psi(f \circ g) \tag{2}$$

To the best of our knowledge optimizations across process and block borders have not been reported for any other system modeling and design methodology. We believe it is very difficult to achieve in most other approaches because process boundaries tend to be very "hard" and moving a piece of functionality from one process to another would involve a major redesign of the processes and their interfaces. This is particularly true for heterogeneous approaches when the two processes are in different modeling domains. In contrast, in the ForSyDe methodology it is a comparably easy step because processes are semantically just like any other function.

We illustrate our refinement methodology by means of the system model of an equalizer which has also been described in [4]. The main task of the equalizer (Figure 5) is to adjust the audio signal according to the *Button Control*. In addition, the bass level must not exceed a predefined threshold to avoid damage to the speakers. This specification is naturally decomposed into four functions shown in Figure 5.
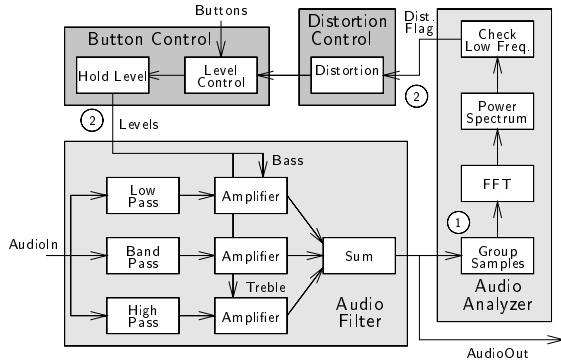

Figure 5: Subsystems of the Equalizer

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 signal, and filters and amplifies it 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. In the following we use the equalizer model to discuss two refinement techniques as indicated in Figure 5. In addition to the transformation techniques described in the following sections ForSyDe includes data type and memory refinement, which are beyond the scope of this paper.
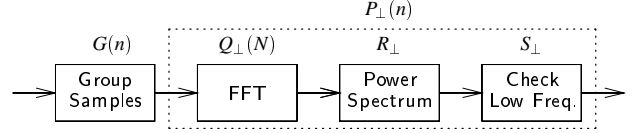
4.1 Refinement of the Clock Domain

Figure 6: The AudioAnalyzer

Figure 6 shows the *Audio Analyzer* subsystem, which includes a Fast-Fourier Transform (FFT) algorithm. This function *fft* takes a vector of 2^k samples and produces the corresponding FFT result. Vectors are denoted as $< x_1, x_2, \ldots, x_n >$. The FFT algorithm is used to determine the frequency spectrum of a signal. It is implemented in the process *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 and issues a warning in this case, which is sent to the *Distortion Control*. The FFT algorithm takes a vector of 2^k samples and calculates a frequency spectrum. The process *Group Samples* ($G(n)$) reads 2^k samples and groups them into a vector of size 2^k. However since we use a synchronous computational model in the system model the grouping process has to produce an output event for each input event. $G(n)$ is defined formally as

$$G(n)(\{x_1, \ldots, x_n, \ldots\}) = \{\underbrace{\perp, \ldots, \perp}_{n-1}, < x1, \ldots, x_n >, \ldots\}$$

As all processes $Q_\perp(n)$, $R_\perp$, $S_\perp$ are constructed with the skeleton *mapSY* we can use the equations 1 and 2 to replace these processes by $P_\perp(n)$.

$$\begin{aligned} p(n) &= s \circ r \circ q(n) \\ P_\perp(n) &= mapSY(\Psi(p(n))) \end{aligned}$$

$$s_1 = \{x_1, x_2, \ldots, x_n, \ldots\}$$
$$s_2 = \{\underbrace{\perp, \ldots, \perp}_{n-1}, \underbrace{< x_1, \ldots, x_n >}_{1}, \ldots\}$$
$$s_3 = \{\underbrace{\perp, \ldots, \perp}_{n-1}, \underbrace{p(< x_1, \ldots, x_n >)}_{1}, \ldots\}$$
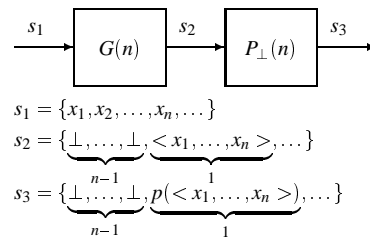
Figure 7: A Mathematical Model of the Audio Analyzer

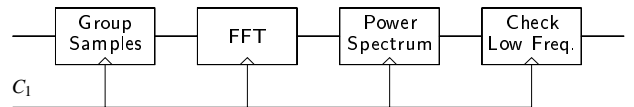We can now model the *Audio Analyzer* as illustrated in Figure 7.

Figure 8: Direct Implementation of the Audio Analyzer

The process $P_\perp(n)$ has to process all absent values. This is not a drawback for the specification phase, but a direct implementation

as shown in Figure 8 can make no use of the fact, that the FFT has only to be calculated at each 2^k-th clock cycle.

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

In order to get a more efficient specification the ForSyDe methodology allows to introduce synchronous sub-domains into the system model during the refinement process. These synchronous sub-domains use another set of tags. The introduction of a new sub-domain is done by well defined transformations, which are semantic-preserving though they introduce an additional timing domain into the model.

We present four processes, that can be used to introduce a synchronous sub-domain. There are two processes for *down-sampling* and two processes for *up-sampling*. Each of these processes comes in two versions, either with *head* or *tail synchronization*. The processes are defined formally as

$$U_H(n)(\{x_1, x_2, \dots\}) = \{x_1, \underbrace{\bot, \dots, \bot}_{n-1}, x_2, \underbrace{\bot, \dots, \bot}_{n-1}, \dots\}$$

$$U_T(n)(\{x_1, x_2, \dots\}) = \{\underbrace{\bot, \dots, \bot}_{n-1}, x_1, \underbrace{\bot, \dots, \bot}_{n-1}, x_2, \dots\}$$

$$D_H(n)(\{x_1, x_2, \dots, x_n, x_{n+1}, \dots\}) = \{x_1, x_{n+1}, \dots\}$$

$$D_T(n)(\{x_1, x_2, \dots, x_n, \dots x_{2n}, \dots\}) = \{x_n, x_{2n}, \dots\}$$

ForSyDe allows only transformations, that use an *appropriate* composition of up-sample and down-sample processes. These combinations are well defined, e.g. the identities $D_H(n) \circ U_H(n)$ and $D_T(n) \circ U_T(n)$. An example for a *non-causal* and thus not allowed composition is shown in Figure 9.
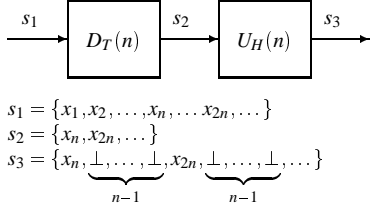

$$s_1 = \{x_1, x_2, \dots, x_n, \dots x_{2n}, \dots\}$$
$$s_2 = \{x_n, x_{2n}, \dots\}$$
$$s_3 = \{x_n, \underbrace{\bot, \dots, \bot}_{n-1}, x_{2n}, \underbrace{\bot, \dots, \bot}_{n-1}, \dots\}$$

Figure 9: Non-Causal Composition of Sampling Processes

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

$$G(n) = U_T(n) \circ D_T(n) \circ G(n).$$

Using another identity

$$P_\bot \circ U_T(n) = U_T(n) \circ P_\bot$$

it follows that

$$
\begin{aligned}
P_\bot(n) \circ G(n) &= P_\bot(n) \circ U_T(n) \circ D_T(n) \circ G(n) \\
&= U_T(n) \circ P_\bot(n) \circ D_T(n) \circ G(n) \quad (3) \\
&= U_T(n) \circ P(n) \circ D_T(n) \circ G(n)
\end{aligned}
$$

In the last step we have replaced $P_\bot(n)$ with $P(n)$ since $D_T(n) \circ G(n)$ does not produce any absent values. Analyzing equation 3 we conclude that the process $P(n)$ processes events only at each n-th tag and thus can be implemented with a slower clock.

Based on these considerations we define a semantic-preserving transformation that introduces a synchronous sub-domain. We implement a synchronous sub-domain 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 result of this transformation of the *Audio Analyzer* is illustrated in Figure 10.
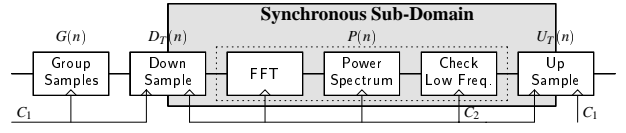

Figure 10: The Audio Analyzer after Refinement

4.2 Communication Refinement

The system 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.
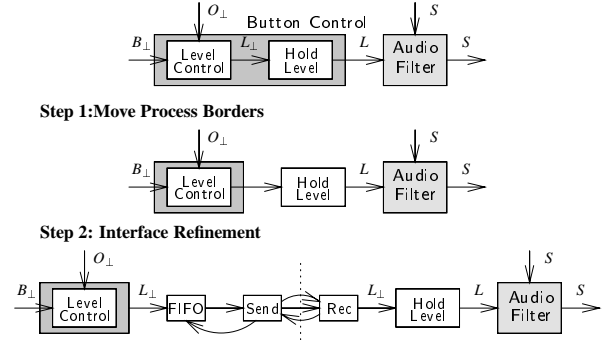

Figure 11: Refinement into a Handshake Protocol

In this paper we focus on the refinement of the synchronous interface between the *Button Control* and the *Audio Filter* subsystems, which is shown in Figure 11. The figure also shows the data types of the signals. Please note, that all the data types $B_\bot, O_\bot, L_\bot$ are extended data types, containing absent values. The block *Hold Level*, implemented by the synchronous skeleton *holdSY*, outputs the last valid value, when receiving an absent value as shown in Figure 12. Thus, it can also be viewed as a transformation unit between Ψ-extended and non-Ψ-extended data types.

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*. The second step is to refine the interface into a handshake protocol.

Figure 12: The Process *holdSY*

Here we introduce a FIFO, a *Send* and a *Receive* process. When *Send* is idle, it tries to read data from the FIFO. Then it sends the message `DataReady` to the *Receiver* and after receiving the message `Ready`, it sends the data. The *Receiver* sends a message `Ack`, when the data is received.

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 does also mean, 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 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 as illustrated in [12]. Assumptions about the environment and the application, such as the kind of expected input signal, have to be validated to justify the applied design decisions.

We can now synthesize the interface by applying the methods described in [12]. 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 nondeterministic channel delays. Of course, the channel can be modeled more realistically if desired. In the ForSyDe methodology we suggest to avoid a non-deterministic model but to use a stochastic channel model instead, which ForSyDe supports with stochastic skeletons [6].

5. CONCLUSION

This paper presents the refinement of an abstract system model into a more detailed implementation model in the ForSyDe methodology. Our main contribution is the development of a method, that allows the stepwise refinement of a system model into an implementation model by the application of well defined design transformations without leaving the functional domain. There are two classes of transformations, semantic-preserving transformations and design decisions. We have illustrated the use of both types of transformations with a digital equalizer system. By means of well defined transformations we refined by the system model with only one synchronous domain into a model with an additional synchronous clock domain. We also presented the formal refinement of a synchronous interface into a handshake protocol, that is used for communication via asynchronous channels.

We have formulated the basic foundations and techniques of the ForSyDe methodology and have applied them manually for several designs. We will continue our work with the development of tool support in order to automate the design flow.

6. REFERENCES

[1] 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 Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(6):834–849, June 1999.

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

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

[4] P. Bjuréus and A. Jantsch. MASCOT: A specification and cosimulation method integrating data and control flow. In *Proceedings of the Design and Test Europe Conference (DATE)*, 2000.

[5] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli. Design of embedded systems: Formal models, validation, and synthesis. *Proceedings of the IEEE*, 85(3):366–390, March 1997.

[6] A. Jantsch, I. Sander, and W. Wu. The usage of stochastic processes in embedded system specifications. In *Proceedings of the Ninth International Symposium on Hardware/Software Codesign*, April 2001.

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

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

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

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

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

[12] I. Sander and A. Jantsch. System synthesis based on a formal computational model and skeletons. In *Proceedings IEEE Workshop on VLSI'99*, pages 32–39, Orlando, Florida, April 1999. IEEE Computer Society.

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

[14] K. Strehl, L. Thiele, M. Gries, D. Ziegenbein, R. Ernst, and J. Teich. FunState - an internal design representation for codesign. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(4):524–544, August 2001.

[15] W. Wu, I. Sander, and A. Jantsch. Transformational system design based on a formal computational model and skeletons. In *Forum on Design Languages 2000*, Tübingen, Germany, September 2000.