A Case Study of Hardware and Software Synthesis in ForSyDe

Zhonghai Lu
Royal Institute of Technology
Stockholm, Sweden
zhonghai@imit.kth.se

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

ForSyDe (FORmal SYstem DEsign) is a methodology which addresses the design of SoC applications which may contain control as well as data flow dominated parts. Starting with a formal system specification, which captures the functionality of the system, it provides refinement methods inside the functional domain to transform the abstract specification into an efficient implementation model which serves as a starting point for synthesis into hardware and software. In this paper we illustrate with a case study of a digital equalizer how a ForSyDe model can be synthesized into a hardware, a software or a combined hardware/software implementation.

Categories and Subject Descriptors

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

General Terms

Design

Keywords

Hardware Synthesis, Software Synthesis, Design Methodology, System Design

1. INTRODUCTION

A SoC (System-on-a-Chip) will be able to integrate more and more heterogeneous computing resources. They can be dedicated (ASICs), programmable (processors and DSP), configurable (FPGAs), passive (memory) or most likely a mixture of these. The design methodology for such complex systems is not obvious.

The ForSyDe methodology [1] targets the design of SoC applications. It offers a modeling technique that results in an abstract and formal system model, and formal design transformation methods for a transparent refinement process of the system model into an efficient implementation model, which serves as a starting point for synthesis into hardware and software.

In earlier papers [2, 3] we have outlined a method for hardware synthesis. Based on this method we have developed a software synthesis method. In this paper we illustrate with a case study of a digital equalizer the synthesis of a ForSyDe model into a hardware, a software and a combined hardware/software description. Although the synthesis steps were applied manually, the whole synthesis process can be automated and the case study will be the groundwork for the future development of our synthesis tool.

2. RELATED WORK

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". 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. And 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 not only strongly support but also establish the foundations of the ForSyDe methodology [1].

Edwards et al. [6] give a comprehensive overview about models of computation. The ForSyDe system model uses the *perfect synchrony hypothesis* which forms the basis for the family of the synchronous languages [4]. It assumes that the outputs of the system are synchronized with the system inputs, while the reaction of the system takes no observable time. This hypothesis abstracts from physical time and serves as a base for a mathematical formalism.

Functional languages have been used in other research projects in electronic design. Reekie [11] used the functional language 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. But this representation was not synthesized to hardware or software. Lava [5] is a hardware description language based on Haskell. It focuses on the structural representation of hardware and offers a variety of powerful connection patterns. Our approach uses the same language, but addresses both data flow and control dominated applications, and adopts a synthesis method rather than dealing with structures on lower levels. Hardware ML (HML) [10] is a hardware description language based on the functional 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, while our system specification is on a significantly higher abstraction level with a very different computational model.

3. THE FORSYDE METHODOLOGY

3.1 The Design Process

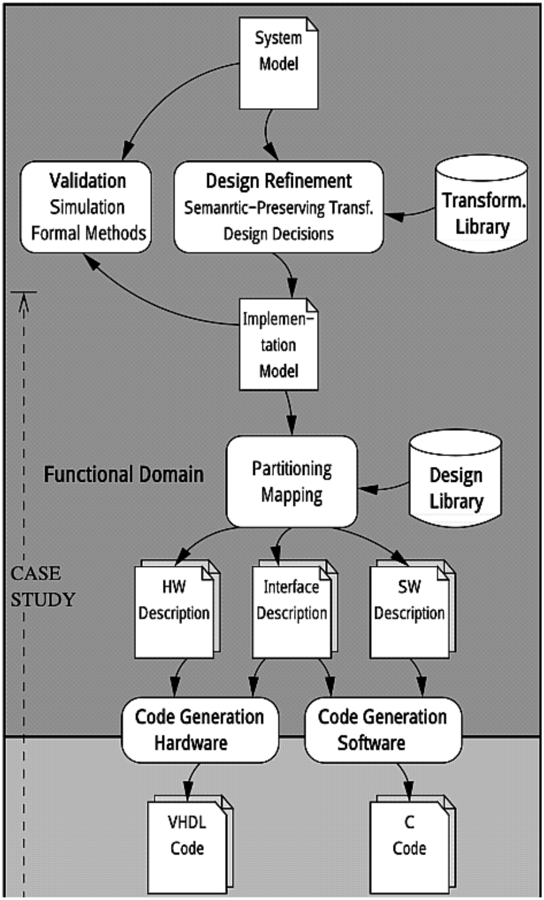

System Model
Validation
Simulation
Formal Methods
Design Refinement
Semanrtic-Preserving Transf.
Design Decisions
Transform. Library
Implementation Model
Functional Domain
Partitioning
Mapping
Design Library
CASE STUDY
HW Description
Interface Description
SW Description
Code Generation Hardware
Code Generation Software
VHDL Code
C Code

Figure 1: The Design Process in ForSyDe

The ForSyDe design process (Figure 1) starts with the development of a formal abstract functional system model, written in the functional language Haskell [14]. This model is then refined inside the functional domain by a stepwise application of well defined design transformations into an efficient implementation model. 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 generate VHDL or C code for the hardware and software parts.

The case study focuses on the synthesis into a hardware and software description. The refinement of the system model is briefly presented in Section 3.3 and not part of this case study.

3.2 The System Model

A system is modeled by concurrent processes. Signals connect processes with each other. A signal is defined as a set of events according to the denotational framework of Lee and Sangiovanni-Vincentelli [9]. It is possibly an infinite, ordered sequence of events. Events have a tag and a value. Synchronous models require totally ordered events with the same set of tags. Events with the same tag

are processed synchronously. In order to model the absence of a value, a data type D can be extended into a data type $D_\perp$ by adding the special value $\perp$. Absent values are used to establish a total order of events when dealing with signals with different or aperiodic event rates.

To model processes, we use the concept of skeletons. A skeleton is a *process constructor*, which takes *combinatorial functions*, i.e. functions that have no internal state, and *values* as input to construct a process. The concept of skeletons has the following additional properties:

- A skeleton cleanly separates computation from synchronization. Synchronization is expressed by the skeleton and computation by the employed combinatorial function(s).

- Skeletons have a structural HW and SW interpretation. Thus a system model based on skeletons also has an interpretation in HW, SW or a mixture of both.

There are two classes of skeletons. The skeleton *zipWithSY* is an example of a *combinatorial skeleton*. It repeatedly applies a function f pairwise on all values of two input signals, and generates the corresponding output signal as illustrated in Figure 2. A process *Sum* can be expressed as $Sum = zipWithSY(+)$.

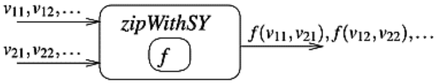


Figure 2: Process Construction with *zipWithSY*

A process $zipWithSY(f)$ is implemented as a combinatorial circuit that implements the function f. Thus the process of Figure 2 is implemented as an adder in hardware. In software such a process is implemented as a software function f with two arguments, in case of Figure 2 an addition operation.

The skeleton *mooreSY* is an example for a *sequential skeleton*. 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 a process $Moore = mooreSY(ns, out, s_0)$ implements the behavior of a finite state machine. In hardware such process is implemented as follows. The functions *ns* and *out* are implemented as combinatorial circuits, the next state and output decoder, and the state is maintained by the memory elements with the initial state s_0, as shown in Figure 3.

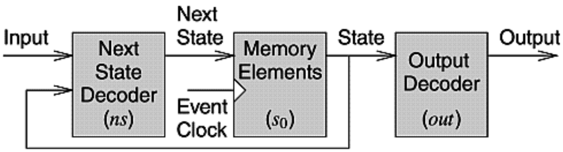
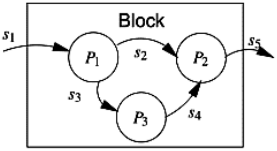


Figure 3: Process Construction with *mooreSY*

Processes can be glued together to build networks of processes. Such a network is called a block. Figure 4 shows how a block is formed by a network of processes. The function of a block is expressed by a set of equations. In the same way, blocks can be composed into higher level blocks, subsystems and a system. The hierarchy can be represented by a structural object tree in which we distinguish three layers as illustrated in Figure 5. The top layer is the *system layer* which defines the system's interfaces to its environment. The middle layer is the *subsystem layer* which consists of one or more levels of networks of blocks. The bottom layer is the *process layer*.



$$\begin{aligned} Block(s_1) = \quad & s_5 \\ \text{where} \quad & (s_2, s_3) = P_1(s_1) \\ & s_5 = P_2(s_2, s_4) \\ & s_4 = P_3(s_3) \end{aligned}$$

Figure 4: A Network of Processes

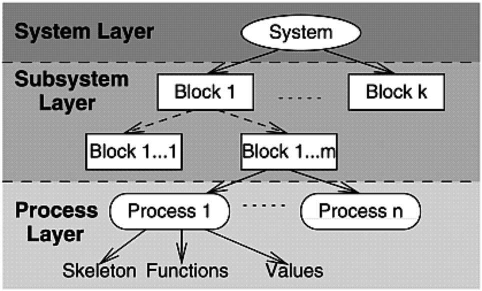


Figure 5: The Layered Structure of a System Model

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.

We have chosen the functional language Haskell [14] as modeling language. Haskell is based on a formal semantics and includes many powerful concepts such as higher-order functions, which fit well with the semantics of the ForSyDe system model. In addition the system model is executable.

3.3 Refinement of the System Model

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 technique [1].

During the refinement phase the system model is stepwise refined through the use of well defined design transformations from an initial specification model S_0 to a final optimized implementation model S_n (Figure 6).

$$\begin{tikzcd} S_0 \arrow[r, "T_1"] & S_1 \arrow[r, "T_2"] & \cdots\cdots \arrow[r, "T_n"] & S_n \end{tikzcd}$$

Figure 6: Refinement through Design Transformations

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. Since in a formal sense, processes are just like any

other function powerful transformations can be applied to change the process structure. For instance, processes can be merged and split, functions inside processes can be moved from one process to another. Thus, at this level it is much easier to explore alternative designs than at the VHDL, SystemC, or C level. 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.

Usually before synthesis the system model is refined into an implementation model using our refinement technique. However, for this case study we used the initial system model as a starting point for the synthesis process since the synthesis principles are the same.

4. CASE STUDY: THE DIGITAL EQUALIZER

The digital equalizer regulates the bass and treble parts of an input audio signal in response to the button levels. In addition, it prevents the bass from exceeding a predefined threshold in order not to damage the later stages in the audio system.

In our case study we have synthesized the digital equalizer model into a hardware, a software and a combined hardware/software description. The refinement was not part of the case study. Since HW synthesis has already been presented in [2, 3] we illustrate the synthesis steps by mainly focusing on the synthesis to software and a combined hardware/software description.

4.1 The Digital Equalizer Model

The digital equalizer is structurally decomposed into four functional blocks or subsystems shown in Figure 7. The function of the

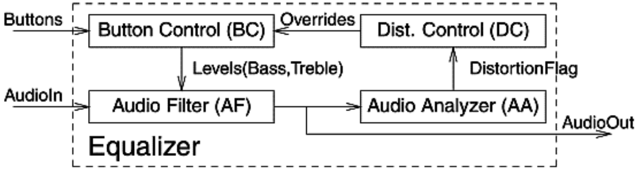


Figure 7: The Digital Equalizer

digital equalizer can be described by the following set of equations:

$$\begin{array}{lcl} \mathit{AudioOut} & = & \mathit{Equalizer}(\mathit{Buttons},\mathit{AudioIn}) \\ \quad \text{where} & & \\ \quad \mathit{AudioOut} & = & \mathit{AF}(\mathit{Levels},\mathit{AudioIn}) \\ \quad \mathit{Levels} & = & \mathit{BC}(\mathit{Buttons},\mathit{init.Overrides}) \\ \quad \mathit{DistortionFlag} & = & \mathit{AA}(\mathit{AudioOut}) \\ \quad \mathit{Overrides} & = & \mathit{DC}(\mathit{DistortionFlag}) \\ \quad \mathit{init} & = & \perp \end{array}$$

The first equation represents the system layer. It takes two input signals *Buttons* and *AudioIn* as arguments, generating the output signal *AudioOut*. The evaluation of this equation calls for the evaluation of the next four equations. These equations describe the subsystem layer. The final equation sets the initial value of the signal *Overrides*, which is needed as the system includes a feedback loop. The *Audio Filter (AF)* subsystem handles the bass and treble part of the digital audio input in response to the amplification level from the *Button Control (BC)*. The internal structure of the *Audio Filter* is depicted in Figure 8. It also shows the skeletons employed for modeling those processes. The *Audio Analyzer (AA)* analyzes

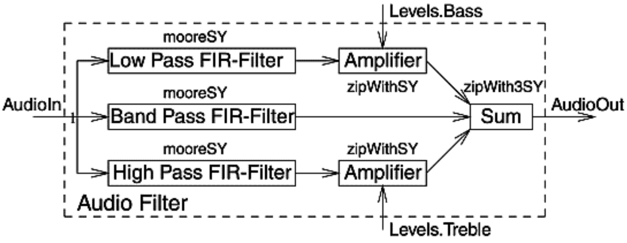


Figure 8: The AudioFilter's Internal Structure

the audio output signal and determines if the bass exceeds a predefined threshold using an FFT to determine the frequency spectrum. The *Distortion Control (DC)* is modeled as an FSM by a sequential skeleton and determines if a violation occurs. In this case it generates the corresponding commands for the *Button Control*. The *Button Control* is also modeled as an FSM and monitors the button inputs and the override signal from the *Distortion Control*, in turn passing the current amplification level to the *Audio Filter*. From the above description we see that the two subsystems *Button Control* and *Distortion Control* are control dominated while the other two subsystems *Audio Filter* and *Audio Analyzer* are data flow dominated.

4.2 Synthesis Technique

In our earlier work [2, 3] we have outlined a synthesis method for hardware. Based on this method we have extended this method to cover also software synthesis. In this case study we follow the steps in Figure 9 to synthesize the digital equalizer into a hardware and software description written in behavioral VHDL and C. It reflects the layered structure of the system model. The synthesis task

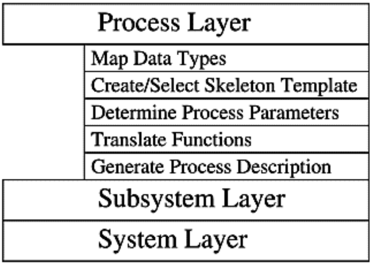

Process Layer
Map Data Types
Create/Select Skeleton Template
Determine Process Parameters
Translate Functions
Generate Process Description
Subsystem Layer
System Layer

Figure 9: The Synthesis Technique in ForSyDe

is divided into three sub-tasks, each of which corresponds to one layer in the system model. At first we synthesize the process layer. We identify all processes in the system model. Each process is constructed by a skeleton, at least one function, and in some cases values like initial states or generic parameters. The process synthesis is built on its skeleton template and the employed combinatorial function(s). Skeleton templates are created once, and later form a skeleton template library in order to be reused. Of course, the data types of the modeling language are mapped to the corresponding data types of the target language, including primitive and compound data types. We also identify values, that are used together with skeletons to form processes. These values are translated to generic parameters or initial states in case of sequential skeletons. Next, we derive translations for all blocks. Since blocks are either

composed of blocks or processes, the results from the process layer are used. A block can be implemented as a netlist of components in hardware or as a hierarchy of functions in software. Finally, the system layer is similarly constructed based on the results from the second step, resulting in a single top-level component in HW and the main program in SW.

4.3 Software Synthesis

In the following sections we illustrate software synthesis of the digital equalizer according to Figure 9. We have chosen to concentrate our presentation of the process layer on the synthesis of a FIR-Filter process in the *Audio Filter* subsystem.

4.3.1 Process Layer

Here we show the synthesis of the parametric process *FIR(h)* that is used to model the filter blocks of the block *Audio Filter* (Figure 8). The FIR filter is shown in Figure 10. The state of the FIR-

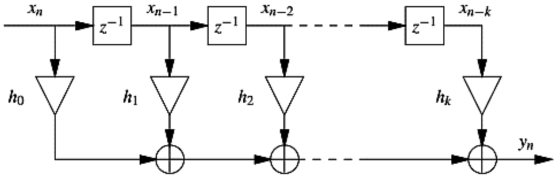


Figure 10: FIR-Filter

Filter can be viewed as a queue with the finite size $k+1$. At each event cycle the function *shiftlV* shifts the input samples one step to the right. The result of the FIR-filter operation is the inner product *ipV(h)* of the state queue and the coefficient vector *h*.

$$FIR(h) = mooreSY(shiftlV, ipV(h), s_0)$$

The FIR-filter can be modeled as a Moore process by means of the skeleton *mooreSY*, where *shiftlV* is the next state function *ns*, *ipV(h)* is the output function *out* and s_0 is the initial state containing a vector of zeros. The coefficients $h_0, \ldots, h_k$ are given as a vector h of the size $k+1$ as argument to the FIR-filter. We briefly show the synthesis of the process *FIR(h)*:

1. **Map Data Types**. The input signal x_n and the output signal y_n are floating point numbers which are directly mapped to the data type 'double' in C. The state type is a vector of floating point numbers which is mapped to an array of 'double'.

2. **Create Skeleton Templates**. Processes communicate via variables shared by the communicating processes. Skeletons are translated into function templates in software. The function template 'moore' for the skeleton *mooreSY* is created as

follows:

```
StateType  current_state , next_state ;
int  initial_flag ;
OutputType moore(InputType input)
{ if ( initial_flag ==0)
     { current_state = Initial_Value ;
       initial_flag =1;}
  else
       current_state = next_state ;
   next_state =ns( current_state , input );
  return out( current_state ); }
```

The external variable 'initial_flag' denotes if the system is in the initial state. The 'ns' corresponds to the next state function *ns* and the 'out' to the output function *out*.

3. **Function Translation.** We translate the two combinatorial functions *ipV(h)* and *shiftlV* in the FIR filter model into the function 'ipV' with an additional parameter 'h' and 'shiftlV' according to their definitions.

4. **Determination of Process Parameters.** The initial state s_0 corresponds to an array of zeros in C. The parameter h is translated into an input argument for the process function. Depending on different values for h, the filter process realizes a LPF, a BPF and a HPF.

5. **Generating the Process Description**. Integrating the results from step1 to 4 leads to the following code for 'filter'. Here 'shiftlV' replaces 'ns' and 'ipV' with an additional argument 'h' replaces 'out'.

```
Vector  current_state , next_state ;
```

```
int    initial_flag  ;
double  filter (double input ,  Vector  h)
{ int  j ;
   if (  initial_flag  ==0)
     {for  ( j =0;  j<length(h );  j++)
          current_state  [ j ]=0.0;
        initial_flag  =1;}
   else
     {for( j =0;  j<length(h );  j++)
          current_state  [ j ]= next_state [ j ];}
   next_state =shiftlV (  current_state  , input );
   return ipV( current_state  ,h ); }
```

4.3.2 *Subsystem and System Layer*

A subsystem is composed of concurrent processes. However in our synthesis scenario there is only one microprocessor. A scheduling policy is needed. We apply the *Periodic Admissible Sequential Schedule* (PASS) algorithm developed for the scheduling of *Synchronous Data Flow* (SDF) networks [8] to the subsystem and system layer of our system model. PASS is a nonempty ordered list of nodes (processes or subsystems) which can be executed repetitively.

Since a process is a function, a subsystem is a function as well. It is hierarchical. For example, we get three process functions 'filter', 'amplifier', 'sum' from the synthesis of the process layer. Applying the SDF theory and the PASS algorithm, we derive a PASS for the subsystem *Audio Filter*.

$$\begin{aligned} PASS(AF) = \{ \quad & LPF, BPF, HPF, \\ & BAmp, TrAmp, Sum\} \end{aligned}$$

The subsystem function 'AudioFilter' is created by calling its process functions by the PASS sequence. In the same way we transform the other three subsystems of the digital equalizer into subsystem functions.

At the system layer we schedule the four parallel subsystem components, leading to a PASS for the digital equalizer:

$$PASS(Equalizer) = \{BC, AF, AA, DC\}$$

The 'ButtonControl' function should run first due to the initial value *init* of the override signal.

4.4 Hardware Synthesis

The HW synthesis of the digital equalizer has been done following the same steps as the SW synthesis. For the process layer, a process in ForSyDe is translated into a component in VHDL. The synthesis is also centered on the skeleton template in HW. For instance,

the FIR filter is modeled by the sequential skeleton *mooreSY*, where its VHDL template consists of three processes (refer to Figure 3). The next state decoder implements *shiftlV* and the output decoder *ipV(h)*. The sequential logic implements the memory bank. Next, each subsystem can be easily translated into a subsystem component by instantiating and connecting the components from the process layer according to the subsystem's internal structure. Finally the system layer is translated into a system component by instantiating and connecting the subsystem components from the subsystem layer according to the system structure.

4.5 HW-SW Implementation

In the case study we have also investigated a mixed HW/SW implementation with two separated clock domains. The data flow parts *Audio Filter* and *Audio Analyzer* are the critical parts of the system and are implemented in HW. The control parts *Button Control* and *Distortion Control* can run at a much lower speed since the buttons are only pressed occasionally. Thus we implement the control parts in SW. Since we have already separately synthesized the digital equalizer into HW and SW, we reuse the data flow parts in HW and the control parts in SW. In addition we introduce an asynchronous interface, a handshake protocol, to handle the communication between the two parts. This asynchronous communication mechanism enables us to turn a synchronous system into a GALS (Globally Asynchronous Locally Synchronous) [12] [13] system to meet design constraints. Figure 11 shows the structure of the implementation using a sender *Send* and receiver *Rec* process for each connection. The protocol works as follows. After data is

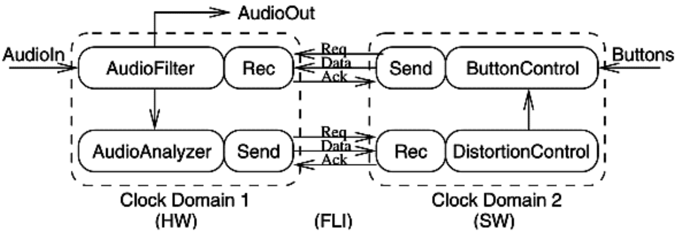


Figure 11: Asynchronous Interfaces

ready, a data transaction is completed in the following four steps: (1) *Send* asserts `Req`, and sends data. (2) *Rec* accepts data, and asserts `Ack`. (3) *Send* deasserts `Req`. (4) *Rec* deasserts `Ack`. For the SW solution the processes including the *Send* (S) and *Rec* (R) processes have to be scheduled. As we use the handshaking protocol receiving and sending data takes two cycles, which is reflected in the following PASS:

$$PASS(SW) = \{R, R, DC, BC, S, S\}$$

We have validated this mixed hardware/software implementation by co-simulation using the VHDL Foreign Language Interface (FLI) of the ModelSim simulator by Mentor Graphics. The VHDL FLI allows to replace a VHDL architecture with C code or to replace the body of a VHDL function with C code. Using the FLI we have co-simulated the hardware parts (VHDL) and software parts (C) together with the handshaking protocol (VHDL and C).

5. CONCLUSION

Haskell Model	VHDL (Ratio)	C (Ratio)
185	1278 (6.91)	1420 (7.68)

Table 1: Comparison of the No. of Code Lines

With a case study of a digital equalizer we have demonstrated the synthesis of an abstract ForSyDe system model into a hardware, a software and a combined hardware/software description. Table 1 compares the number of code lines of the ForSyDe model of the digital equalizer in Haskell, with its translated results, the hardware representation in behavioral VHDL as well as the software representation in C. Although the synthesis has been done manually it can be automated because of the formal definition of the synthesis steps.

The synthesis technique is an integral part of our methodology. We have formulated the basic foundations and techniques of the ForSyDe methodology including modeling, refinement as well as synthesis, 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. ACKNOWLEDGMENTS

The work reported in this paper was supported in part by the Swedish government within the Socware program, and in part by the Swedish Foundation for Strategic Research (SSF) in the Integrated Electronic Systems (INTELECT) program.

7. REFERENCES

[1] Ingo Sander and Axel Janstch. Transformation Based Communication and Clock Domain Refinement for System Design. In *Proceedings of the 39th Design Automation Conference*, 20(1):281–286, New Orleans, USA, June 2002.

[2] Ingo Sander and Axel Jantsch. System synthesis utilizing a layered functional model. In *Proceedings Seventh International Workshop on Hardware/Software Codesign*, pages 136-140, Rome, Italy, May 1999. ACM Press.

[3] Ingo Sander and Axel 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.

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

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

[6] 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.

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

[8] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, C-36(1), January 1987.

[9] 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.

[10] 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.

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

[12] Daniel M. Chapiro. *Globally-Asynchronous Locally-Synchronous Systems*. PhD Thesis, Stanford University, Oct. 1984.

[13] Thomas Meincke, Ahmed Hemani, S. Kumar, P. Ellervee, J. Öberg, T. Olsson, P. Nilsson, D. Lindqvist, and H. Tenhunen. *Globally asynchronous locally synchronous architecture for large high performance ASICs*. In Proceedings of IEEE International Symposium on Circuits and Systems, pages II 512-515, Orlando, USA, May 1999.

[14] S. Thompson. *Haskell - The Craft of Functional Programming*. Addison-Wesley, 2 edition, 1999.