Formal System Design Based on the Synchrony Hypothesis, Functional Models, and Skeletons

Ingo Sander, Axel Jantsch
Department of Electronics, Royal Institute of Technology, Stockholm, Sweden

Abstract

Formal approaches to HW and system design have not been generally adopted, because designers often view the modelling concepts in these approaches as unsuitable for their problems. Moreover, they are frequently on a too high abstraction level to allow for efficient synthesis with today's techniques. We address this problem with a modelling method, which is strictly formal and based on formal semantics, a pure functional language, and the synchrony hypothesis. But the use of skeletons in conjunction with a proper computational model allows to associate a direct hardware interpretation. In particular we use (1) the synchrony hypothesis and a timed signal model to provide a high abstraction for communication at the system level. This facilitates efficient modelling and design space exploration at the functional level, because the designer is not concerned with complex communication mechanisms, and functionality can easily be moved from one block to another. To bridge the gap between an elegant and abstract functional model and the details of an implementation we use (2) skeletons to encapsulate primitive structures, such as FSMs, buffers, computation units, etc. in a purely functional way.

1. Introduction

High abstract, formal models can be conveniently used to capture the essential functionality of a system and to utilize theorem provers and other formal analysis and verification techniques. This is of increasing importance when systems become more and more complex and system analysis and verification becomes one of the major roadblocks in product development. On the other hand, there is a significant gap between high abstract, formal models and all the details of an implementation, which has so far prohibited efficient synthesis techniques. We attempt to bridge this gap without compromising the formal properties and the high abstraction level of a functional model. We do this with (1) a carefully selected computational model and (2) a modelling discipline based on timed signals for communication and skeletons for typical design patterns.

The computational model essentially provides abstractions of *communication* and *time*. This allows the concentration on the important system functionality. Furthermore, because communication in our model is a very simple data flow mechanism, functions can be added, removed and re-grouped very easily. This is in contrast to many modelling approaches based on concurrent processes with an asynchronous, FIFO based communication. There, functions cannot be easily moved from one process to another because this requires the redesign of the communication structure. Furthermore, asynchronous message passing is a very complex mechanism which cannot be implemented in hardware directly.

The modelling discipline employs skeletons in form of higher order functions to capture and provide primitive structures such as FSMs, buffers, FIFOs, functional units, etc. Although they conform to a strictly formal model and can therefore be subject to formal analysis and transformation, they also have a direct hardware interpretation which forms the base for efficient synthesis.

The rest of the paper is organized as follows. Section 2 discusses related work, Section 3 introduces the computational model and skeletons, and Section 4 illustrates the modelling technique with a fairly large industrial example. In Section 5 we discuss what research remains to be done to make this approach usable in praxis, but we try to underscore that the concepts presented here are the core of the method and the remaining research on synthesis and verification are based on existing knowledge and techniques, and are therefore not insurmountable problems.

2. Related Work

Many computational models have been described. For a comprehensive overview see Edwards et al. [4].

Very often real-time systems are specified by means of concurrent processes, which communicate asynchronously. Such a communication model forms the base for languages as SDL [17, 19], VHDL [18], or SpecCharts [15]. While this model serves as a good implementation model, due to its closeness to architecture, we argue, that it is not a good choice for a functional system model. Many design decisions are already present in such a model, in particular the partitioning into processes and the communication mechanism between the processes. It is very difficult to correct a wrong design decision in later design phases. The complexity of the communication mechanism in some of these languages, such as asynchronous message passing with infinite buffers e.g. in SDL or Erlang [21], is a major difficulty for both, the functional design exploration and the subsequent implementation, even though its

simple usage in these languages does not make it always apparent. For functional design exploration it is a problem, because it makes it difficult to move a sub-functionality from one process to another. If for instance a sub-function *SF1* is to be moved from process *P1* to process *P2*, the communication link between *SF1* and *P1* has to be changed from the intra-process communication scheme, shared memory, to the inter-process communication scheme, say message passing. On the other hand, the communication link between *SF1* and *P2* has to be changed from message passing to shared memory based. This is often a complicated and error prone procedure which makes functional design space exploration tedious. For the implementation phase a complex communication mechanism such as asynchronous message passing with infinite FIFOs is problematic, because it can never be implemented fully in hardware. In fact, this is most of the time not even necessary because in many cases a much simpler mechanism, such as a strobe based or a handshake based protocol suffices [16, 19]. But even a complicated analysis cannot always find the simplest possible implementation.

In addition the asynchronous communication mechanism makes it very difficult to reason about such a model and to apply formal methods, because of the state space explosion and potential non-determinism.

A formal approach, the synchrony hypothesis [1], forms the base for the family of synchronous languages, which are designed to target reactive systems. It assumes, that the outputs of a system are synchronized with the system inputs, while the reaction of the system takes no observable time. The synchrony hypothesis abstracts from physical time and serves as a base for a mathematical formalism. All synchronous languages are defined formally and system models are deterministic.

The family of synchronous languages can be divided into two groups. LUSTRE [5] and SIGNAL [9] are designed for data flow applications, while ESTEREL [3] and STATECHARTS [6] target control-oriented applications. However, there is no language, which is good in both areas as elaborated in [1]. We use this theory for our computational model, but go beyond it by using a more powerful language paradigm, which allows us to address both, data flow and control flow applications.

Reekie [12] used the functional language Haskell [10] to model digital signal processing applications. He modelled streams as infinite lists and used higher-order functions to operate on them. Finally, correctness preserving transformations were applied to transform a system model into an effective implementation. Transformations of functional programs is an active research field of the functional programming community [2, 11].

The parallel programming community has used functional languages to derive parallel programs from a functional specification [13, 14]. Skeletons are used to structure a problem, and then transformed into an efficient implementation for a chosen parallel architecture.

For complex telecommunication systems [20] the computational model should fulfil the following requirements:

- The computational model abstracts from implementation details.
- The computational model is based on a sound mathematical formalism and support formal methods.
- The modelling language used in the computational model is able to represent control and dataflow parts.
- The system model can be transformed into an implementation.

To fulfil these requirements we choose to adopt the synchrony hypothesis and place it in a functional environment. We use skeletons, which have a hardware interpretation and allow program transformation. The system model is implemented in the functional language Haskell [10], which is based on formal semantics.

3. Computational Model

3.1. Definition

For a formal definition of the computational model we use the denotational framework of Lee and Sangiovanni-Vincentelli [8]. It defines a signal as a set of events, where an event has a tag and a value. Tags are used to model the order of events. In our model events are totally-ordered by their tags. We model synchronous systems, that means every signal has the same set of tags. Events with the same tag are processed synchronously. To model the absence of an event, we use a special value $\perp$ (“bottom”).

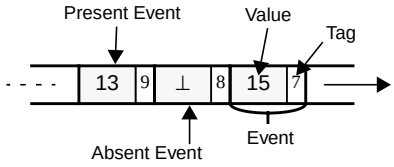

Figure 1. A signal is a set of events

Absent events are necessary to establish a total ordering of events for real time systems with variable event rates.

A system is modelled by means of concurrent processes. Events with the same tag are processed synchronously. The output signals of a process are synchronized with its input signals and are processed instantaneously. There is no delay inside a process.

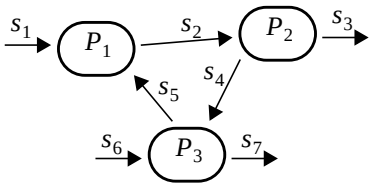

Figure 2. A system is modelled with concurrent processes

3.2. Modelling Language

The language we use for system modelling has to fulfil the following requirements:

- It is *based on formal semantics* and *purely functional*, i.e functions have no internal state. They are free of side effects. This facilitates the application of formal methods for transformation, synthesis, and verification.

- It supports *higher-order functions*, which we use to formulate skeletons, see Section 3.3.
- It has a *lazy* evaluation mechanism, i.e. an argument to a function is only evaluated when needed. This allows to model infinite input streams in a pure functional, side effect free way.
- It provides a variety of control constructs to facilitate also the modelling of complex control flow. Functional languages are naturally to use for data flow applications, but we also want to address control dominated systems, as we illustrate in Section 4.
- It is executable to allow the simulation of the system model.

We have chosen Haskell because it fulfils our requirements and offers some other versatile facilities such as a powerful *type system.*

A Haskell program is a function, which is a composition of other functions. We introduce some of the properties of Haskell with the higher-order function `map`. It applies a function on all elements in a list.

```
map f []     = []
map f (x:xs) = f x : (map f xs)
```

The function `map` has two arguments (written by juxtaposition). The first argument is a function `f` and the other argument is a list. The function uses pattern matching. The first pattern matches, when the list is empty. The second pattern matches, when the list is constructed (`:`) by a first element (`x`) and a rest list (`xs`).

```
map (*2) [1,2] = (1*2) : map (*2) [2]
               = 2 : (*2) 2 : map (*2) []
               = 2 : 4 : []
               = [2,4]
```

The type system of Haskell infers the following type for `map`:

```
map :: (a -> b) -> [a] -> [b]
```

This means `map` is a function, that takes a function as its first argument. This function is characterized by the specification, that it takes one argument of a type `a` and produces a result of type `b`. The second argument of `map` is a list with elements of type `a`. The result of `map` is a list of type `b`. Thus `map` can be used for all functions and lists which are compatible to the type of `map`, which makes it very general and useful. In addition the lazy evaluation mechanism of Haskell allows to use infinite lists, as lists are processed elementwise from the front. We use this mechanism to model signals.

3.3. System Modelling with Skeletons and Function Composition

Following the definition of our computational model in Section 3.1 we describe in this section how a system is modelled in the functional language Haskell. First, we discuss the modelling of signals. Second, we show how skeletons are used to model processes, and finally we introduce function composition, which is used to compose the system model.

Signals. We model signals in the functional language Haskell by means of infinite lists, where the tag corresponds to the position in the list.

```
type Signal value = [value]
```

In our model we use *timed* signals, which can contain absent events. We define a datatype `Token`, which is used to represent absent events or present events of the type `value`.

```
data Token value =  Absent
                 | Present value
```

A timed signal is a signal of the type `Token value`. This is expressed by means of a type synonym `Timed`:

```
type Timed value = Signal (Token value)
```

Elementary Processes. Elementary processes are modelled with *skeletons*. A skeleton is a *higher-order function*, which takes *elementary functions* and signals as input parameters and produces signals as output. We define an elementary function as a function, that is combinatorial and does not include any timing behaviour.

The use of skeletons is the following:

- Skeletons are used for the synchronization of signals. They separate timing behaviour from computation, the latter is done by means of the elementary functions.
- Skeletons can contain state information.
- A skeleton has a hardware interpretation. Thus, a system model, which is a composition of skeletons, has also an interpretation in hardware.
- As skeletons are higher-order functions, the work on correctness-preserving transformations, which has been done by the functional programming community [2, 11] can be used to transform a system model into a more effective implementation model.

Notice, that the transformation of skeletons does not affect the elementary functions used by the skeletons. Thus functions written in other languages as VHDL or C can be used as elementary functions as well, as long as they fulfil the requirements on elementary functions.

In the following we present some important skeletons and give hardware interpretations for them. We use the following naming convention: if a skeleton operates on both untimed and timed signals, we name it `skeletonS`, if it operates only on timed signals we name it `skeletonT`.

The skeleton `mapS` is based on the higher-order function `map` (Section 3.2), which recursively applies a function `f` on all elements of a list. `mapS` can be interpreted as a combinatorial component with one input. An inverter can be modelled by means of `mapS not`. In this case `not` is an inverter function.

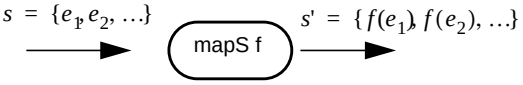

Figure 3. The skeleton mapS

The skeleton `zipWithS` applies a function `f` eventwise on two signals. `zipWithS` is illustrated in Fig. 4. `zipWithS` can be interpreted as a combinatorial component with two inputs. It can be used to model an adder. In

this case the function `f` is an addition function.

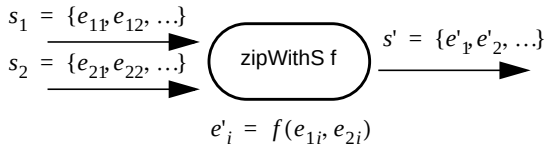

Figure 4. The skeleton zipWithS

The skeleton `scanlS` applies a function `f` on the events of a signal and an internal state `mem`. The result of the function `f` works as the new state and as output. `scanlS` can be interpreted as a state machine with no output decoder. The needed memory elements can be derived from the datatype of `mem`.

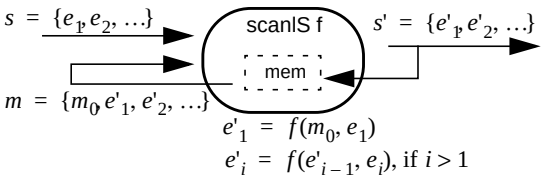

Figure 5. The skeleton scanlS

Composition of Processes. We use *function composition* to compose new processes. Haskell provides a composition operator ".", which takes two functions `f` and `g` as arguments and produces a new function. The composition operator is defined by

```
(f . g) x = f(g(x))
```

Systems are modelled by composition of processes. In addition, libraries of application-oriented functions, can be built by the composition of skeletons. Due to its composition of skeletons, each library element has a hardware interpretation. However, often an effective implementation is known for certain library elements and can be added to the library.

The concept of library elements is illustrated with a small example. We use the skeletons `mapS` and `scanlS` to constitute a new library element `mooreS`.

```
mooreS nextState output initState
       = mapS output . scanlS nextState initState
```

`mooreS` can be interpreted as a Moore-FSM. It takes two elementary functions, `nextState` and `output`, and a value `initState` for the initial state as arguments. We illustrate mooreS in Fig. 6.

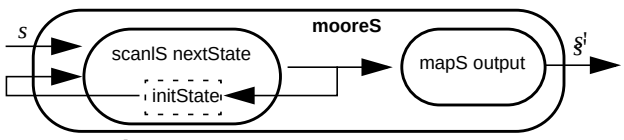

Figure 6. The composite process mooreS

Now, we use `mooreS` to constitute a new library element `idealFifoT`, that is used to model an *ideal FIFO*. An ideal FIFO has the following characteristics: (1) it has a buffer of infinite length; (2) it accepts 0, 1 or more data items per event cycle; (3) it outputs at most one event per event cycle. Following (1) we model the buffer, which is the state of the FIFO, by means of a list. The input to the FIFO is modelled by a timed signal which carries a list of values (2), while the output of the FIFO is a timed signal of a which carries values (3). We model the ideal FIFO by means of the function `mooreS` and two elementary functions `fifoState` and `fifoOutput`.

```
idealFifoT :: Timed [a] -> Timed a
idealFifoT = mooreS fifoState fifoOutput []
```

The function `fifoState` is used to calculate the new state of the buffer. The function `fifoOutput` analyses the buffer and outputs an absent event, if the buffer is empty, or the first element of the buffer as a present event.

Real FIFOs can be modelled by means of the skeleton `mooreS` as well. In this case the datatypes used for the buffer and the parallel input data have to be changed to a bounded datatype, e.g. an array or a list with a maximum number of elements.

4. Case Study: ATM Switch with Operation and Maintenance Functionality

We illustrate our design approach by means of an industrial example, an ATM switch with operation and maintenance functionality [7, 16]. This application monitors and switches large amounts of data and has a defined timing behaviour.

In particular, we emphasize the following:

- We show how a functional system model is developed with our approach. The behaviour of the system model is only based on data-dependences. It abstracts from a low-level communication mechanism. This leaves a wide design space and is a good starting point for further design exploration. (Section 4.1)
- Although the system model is purely functional, the use of skeletons makes it possible to interpret the model as a hardware structure. The interpreted hardware structure works as a base for synthesis. (Section 4.2)

4.1. System Model Development

System-Level: ATM Switch. We start to develop the system model from the top. We model the ATM switch by means of a function *switchCore*, which is responsible for the switching, and two identical operation and maintenance functions *OAM* (Fig. 7).

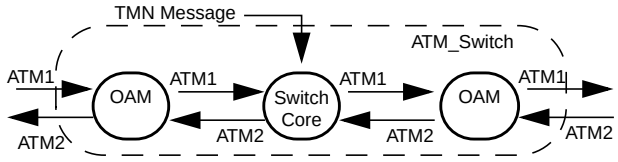

Figure 7. System model of the ATM-Switch

The ATM-Switch model is a function which takes two ATM signals and a signal with messages from the TMN layer as input and produces two ATM signals as output. The translation of the system model in Fig. 7 to Haskell is straightforward.

```
atm_Switch atm1_in atm2_in tmn_Msg
  = (atm1_out,atm2_out) where
    (atm1_toMux,atm2_out)
      = oam atm1_in atm2_fromMux
    (atm1_fromMux,atm2_fromMux)
      = switchCore atm1_toMux atm2_toMux tmn_Msg
    (atm2_toMux, atm1_out)
      = oam atm2_in atm1_fromMux
```

The top function `atm_Switch` takes `atm1_in`, `atm2_in` and `tmn_Msg` as arguments and produces two outputs `atm1_out` and `atm2_out`. The output is written as the tuple (`atm1_out`, `atm2_out`). The specification of `atm1_out`, `atm2_out` and the internal signals inside the function `atm_Switch` are given inside the *where*-clause, which is used in Haskell to define local declarations. The order of functions is irrelevant. This top-level specification of the ATM-Switch can be seen as a definition by means of a set of equations.

Subsystem-Level: Switch Core. Descending one level we illustrate the model of the Switch Core (Fig. 8). The function *switch table controller* updates the switch table (both directions) and distributes the necessary parts of the switch table (one direction) to the *cell handler* function, which switches the ATM cells on the virtual connection or virtual path level.

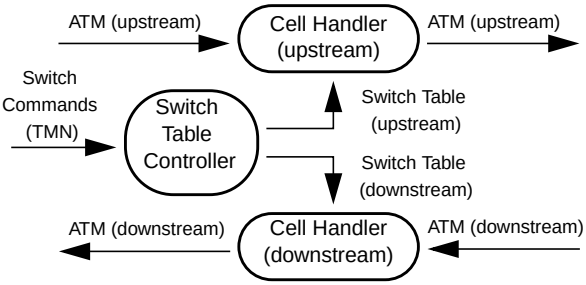

Figure 8. ATM Switch: Switch

```
switchCore atmUp_In atmDown_In command
  = (atmUp_Out, atmUp_Out) where
    atmUp_Out = cellHandler atmUp_In tableUp
    atmDown_Out =
      cellHandler atmDown_In tableDown
    (tableUp,tableDown) =
      switchTableController command
```

Subsystem-Level: OAM. The OAM functionality is modelled by means of the function `oam`, which is illustrated in Fig. 9. The function oam_Extractor monitors ATM cells and extracts OAM information. This information is sent to the function `oam_Handler`, which observes the status of all virtual paths. Depending on that, it sends OAM cells to the function `oam_Inserter`. The function `oam_Inserter` inserts OAM cells into a stream of ATM cells.

```
oam atmUp_In atmDown_In
  = (atmUp_Out, atmDown_Out) where
    atmUp_Out = oam_Inserter userUp oamUp
    atmDown_Out = oam_Inserter atmDown_In oamDown
    (userUp, oamInf) = oam_Extractor atmUp_In
    (oamUp, oamDown) = oam_Handler oamInf
```

Skeleton-Level: The functions in the subsystems Switch

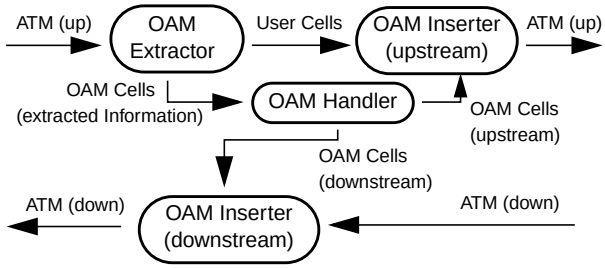

Figure 9. ATM-Switch: OAM-

Core and OAM are modelled with skeletons. Here, we show how the function oam_inserter is modelled with the skeleton zipWithS and the library element idealFifoT.

```
oam_Inserter userCells oamCells
  = (idealFifoT . zipWithS mergeSignals)
      userCells oamCells
```

The function has the following hardware interpretation:

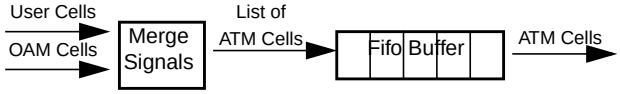

Figure 10. The function OAM_Inserter

4.2. Hardware Interpretation of the System Model

Using the hardware interpretation of Section 3.3 for each skeleton, we get a hardware interpretation for the entire system (Fig. 11).

This interpretation is still based on data-dependences only, which leaves a wide design space. In particular functions can easily be moved from one subsystem to another, because the communication mechanism between the blocks in Fig. 11 is identical with the mechanism inside the blocks, i.e. parameter passing during function calls. In other modelling approaches, where the interaction of processes does not depend on data-dependences, but on a complex control and communication mechanism, subfunctions cannot be moved from one block to another without great effort.

5. Conclusion

We presented a novel design methodology for system design, which combines the synchrony hypothesis with the functional language paradigm in order to design both control and data flow dominated systems. The design starts with a high level system model, that is purely functional and only based on data-dependences. Thus, the system model abstracts from implementation issues such as communication mechanisms and its formal nature supports formal methods and verification. However, despite the high abstraction level, the use of skeletons allows it to interpret the system model as a hardware structure. This hardware structure can be used as a starting point for synthesis.

We will focus our future work on synthesis to derive a synthesizable VHDL-model from our system model. We divide this problem into the following subproblems.

Synthesis of elementary functions. Elementary functions are combinatorial and without side-effects which

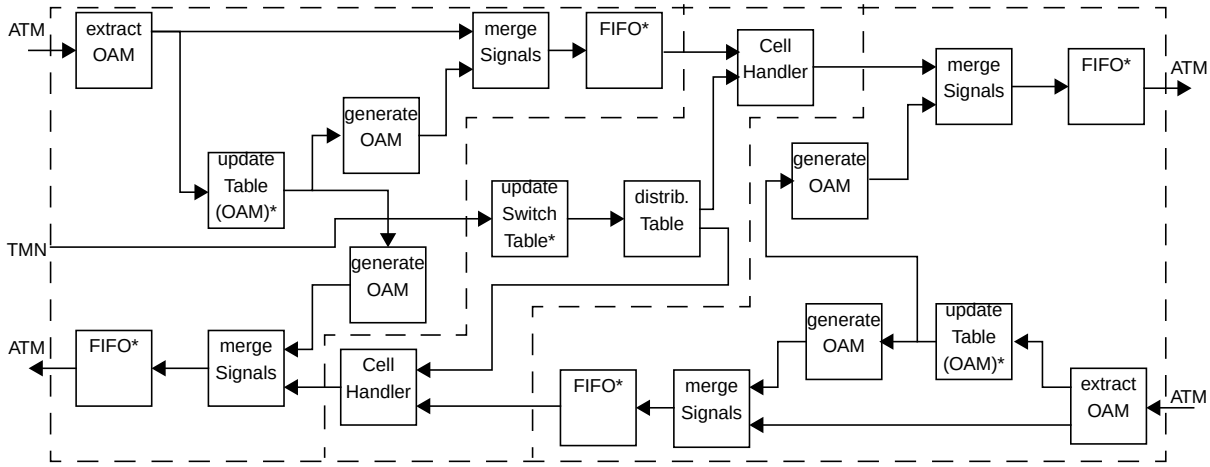

Figure 11. Hardware Interpretation of the System Model (Functions that have an internal state are marked with *)

makes their synthesis straightforward and yields a combinatorial VHDL-process.

Synthesis of states. The skeleton `scanlS` contains states. These states can be synthesized to memory elements in correspondence with its data type.

Synthesis of skeletons. A skeleton is synthesized into a hardware structure, e.g. `scanlS nextstate state` defines a finite state machine. The elementary function `nextstate` will be synthesized into a combinatorial process, that calculates the next state and `state` is synthesized into a memory element, which represents the state of the finite state machine.

Synthesis of datatypes. For specification purposes systems are described on a high abstraction level. Thus abstract datatypes are used widely in the system model. These datatypes have to be transformed into synthesizable datatypes. This problem is general for all approaches where abstract data types are used.

Communication synthesis. One of the key properties of the system model is, that it is based on data-dependences and abstracts from a communication mechanism. This property leaves a wide design space with respect to communication between processes. The key issue is to analyse the data dependences to generate effective communication schemes.

References

[1] A. Benveniste and G. Berry, "The Synchronous Approach to Reactive and Real-Time Systems", *Proceedings of the IEEE*, Vol. 79, No. 9, pp. 1270-1282, September 1991.

[2] R.S. Bird, *Lectures on Constructive Functional Programming*, Oxford University Programming Research Group, Technical Monograph PRG-69, 1988.

[3] F. Boussinot and R. de Simone, "The ESTEREL Language", *Proc. of the IEEE*, Vol. 79, No. 9, pp. 1293-1304, Sep. 1991.

[4] S. Edwards, L. Lavagno, E. A. Lee and A. Sangiovanni-Vincentelli, "Design of Embedded Systems: Formal Models, Validation and Synthesis", *Proc. of the IEEE*, March 1997.

[5] N. Halbwachs, P. Caspi, P. Raymond and D. Pilaud, "The Synchronous Data Flow Programming Language LUSTRE", *Proc. of the IEEE*, pp. 1305-1320, September 1991.

[6] D. Harel, "STATECHARTS: A Visual Approach to Complex Systems", *Science of Computer Programming*, 8-3, 1987.

[7] ITU, *Recommendation I.610 - B-ISDN Operation and Maintenance Principles and Functions*, Geneva, 1993

[8] E. A. Lee and A. Sangiovanni-Vincentelli, "A Denotational Framework for comparing Models of Computation", *Technical Memorandum UCB/ERL M97/11*, University of California, Berkeley, California, 1997.

[9] P. Le Guernic, T. Gautier, M. Le Borgne and C. de Marie, "Programming Real-Time Applications with SIGNAL", *Proc. of the IEEE*, pp. 1321-1335, September 1991.

[10] J. Peterson and K. Hammond, editors, *Haskell Report 1.4*, http://haskell.org/.

[11] A. Pettorossi and M. Proietti, "Rules and Strategies for Transforming Functional and Logic Programs", *ACM Computing Surveys*, vol. 28, no. 2, pp. 361 - 414, June 1996.

[12] H. J. Reekie, *Realtime Signal Processing*, Ph.D. thesis, University of Technology at Sidney, Australia, 1995.

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

[14] Mario Südholt, *The Transformational Derivation of Parallel Programs using Data-Distribution Algebras and Skeletons*, Ph.D. thesis, Technical University of Berlin, 1997.

[15] S. Narayan, F. Vahid, and D. D. Gajski, "System Specification with SpecCharts Language", *IEEE Design & Test of Computers*, December 1992.

[16] W. Horn, *Modelling of an ATM Multiplexer in a Network Terminal for a Mixed Hardware/Software Implementation*, Royal Institute of Technology, Stockholm, report no. TRITA-ESD-1998-06, May 1998.

[17] J-M. Daveau, G.F. Marchioro, C.A. Valderrama, and A. Jerraya, "VHDL generation from SDL specifications", *Proc. of Computer Hardware Description Languages*, April 1997.

[18] W. Ecker, "Using VHDL for HW/SW Co-Specification", pp. 500-505, *European Design Automation Conference*, 1993.

[19] B. Svantesson, S. Kumar, A. Hemani, "A Methodology and Algorithms for Efficient Inter-process Communication Synthesis from System Description in SDL", *Proceedings of the IEEE International Conference on VLSI Design*, 1998.

[20] I. Bolsens, H. de Man, B. Lin, K. van Rompaey, S. Vercauteren, and D. Verkest, "Hardware/Software Co-design of Digital Telecommunication Systems", *Proc. of the IEEE*, vol. 85, no. 3, pp. 391 - 418, March 1997.

[21] J. Armstrong, R. Virding, and M. Williams, *Concurrent Programming in Erlang*, Prentice Hall, 1993.