# System Synthesis Utilizing a Layered Functional Model

Ingo Sander
Department of Electronics
Royal Institute of Technology
Stockholm, Sweden
ingo@ele.kth.se

Axel Jantsch
Department of Electronics
Royal Institute of Technology
Stockholm, Sweden
axel@ele.kth.se

## ABSTRACT

We propose a system synthesis method which bridges the gap between a highly abstract functional model and an efficient hardware or software implementation. The functional model is based on a formal semantics and the synchrony hypothesis. However, the use of skeletons in conjunction with a proper computational model structures the system description into three layers, the system layer, the skeleton layer, and the elementary layer. The synthesis process takes advantage of this structure and uses a different technique for each layer: (a) connection of components and processes at the system layer; (b) template based generation of compound entities possibly containing state information, memory, and complex control at the skeleton layer; this layer also determines the communication and timing behaviour; (c) direct translation into combinatorial functions at the elementary layer. Thus, without compromising the formal properties of the abstract system model we provide an efficient synthesis method.

## 1. INTRODUCTION

High abstract, formal models can be conveniently used to capture the essential functionality of a system and to utilize theorem provers, model checkers, and other formal analysis and verification techniques. 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 based on timed signals for communication and skeletons for typical design patterns, (2) a two-phase design exploration, i.e. data type exploration and architecture exploration, and (3) a synthesis method based on skeletons which uses explicitly formulated design decisions to infer all the details of an implementation.

The computational model [11] is based on the perfect synchrony hypothesis and data flow as communication concept. It essentially provides abstractions of *communication* and *time*. This allows the concentration on the important system functionality. Furthermore, due to the abstract communication, functions can be added, removed and re-grouped very easily, in contrast to many modelling approaches based on concurrent processes with complicated asynchronous or synchronous communication mechanisms.

Our synthesis method is based on skeletons which provide a structural hardware and software interpretation. During a design exploration phase design parameters are evaluated, e.g. the size of buffers and the interconnect architecture between processes. The resulting design decisions are explicit inputs to the synthesis process which fills in all the details implied by the design decisions and the functional model. The synthesis deals with several aspects, i.e. skeletons, elementary functions, timing model, lists and data types, and communication. The result is synthesizable VHDL and C code. However, in this paper we focus mainly on hardware synthesis of skeletons and elementary functions.

The next section discusses related work, Section 3 gives an overview of the design methodology, Section 4 introduces the computational model and skeletons, and Section 5 describes the synthesis techniques and gives some results.

## 2. RELATED WORK

Many computational models have been described in the literature. 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 such as SDL and VHDL. 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 the 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, 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.

The synchrony hypothesis [1] forms the base for the family of synchronous languages. 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. Synchronous languages are either dedicated for data flow applications (e.g. Lustre [5]) or control oriented applications (e.g. ESTEREL [3]). 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 [8] used the functional language Haskell [7] 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.

The parallel programming community has used functional languages to derive parallel programs from a functional specification [9, 10]. They use skeletons to structure a problem.

This formulation is then transformed into an efficient implementation for a chosen parallel computer architecture.

Our modelling approach has been described in detail in [11] and it falls, according to a classification by Skillicorn and Talia [12], into the "nothing explicit" level, with parallelism, mapping and communication implicit in the model and therefore left to be decided by the design and synthesis. However, the use of a specific computational model and skeletons restrict the model to a static structure with bounded communication which can be determined at synthesis time. Because of this restriction cost measures can be developed to control and predict performance and cost of an implementation, as elaborated in [12].

Only few attempts to synthesize hardware from an abstract, functional specification have been published. All of them differ significantly from our approach. Ruby [14] is a circuit description language based on relations. The target applications are regular, data flow intensive algorithms, and much of its emphasis is on layout issues. The same idea is the base for Lava [2], which is a tool to assist circuit designers in specifying, designing and implementing hardware. It is based on the functional language Haskell. Our approach uses the same language, but addresses both data flow and control dominated applications and uses links to commercial logic synthesis tools rather than dealing with structures on lower levels. HML [13] is a hardware description language based on Standard ML, which is a functional language similar to Haskell. However, HML attempts to replace VHDL or Verilog as hardware description languages, while we propose a system specification concept on a significantly higher abstraction level with a very different computational model. In [13] a direct translation of HML to VHDL is described, which would not be possible in our approach since we propose a design space exploration and synthesis method which requires explicit user input in the form of design decisions.

To summarize, we base our work on the synchrony hypothesis, place it in a functional environment, use skeletons to limit the models to statically determined computation and communication structures, and propose a design and synthesis method which involves design space exploration and explicit design decisions.

## 3. DESIGN METHODOLOGY

In our design methodology (Fig. 1) system design starts with the

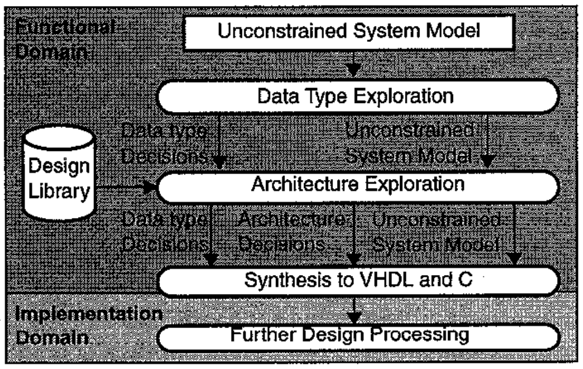


**Figure 1. Design Methodology**

development of an *unconstrained functional* system model which is based on a synchronous computational model (Section 4.1), a functional modelling language (Section 4.2) and the use of skeletons (Section 4.3). The system model is *functional* in the sense, that it uses formally defined functions to focus on the system functionality rather than structure and architecture. The behaviour of the system model is only based on data-dependences. It abstracts from implementation details, in particular from low-level communication and timing mechanisms. The system model is *unconstrained* in the sense, that it uses unconstrained data types, such as infinite lists and numbers with arbitrary precision. The nature of the unconstrained system model leaves a wider design space compared to traditional system models based on imperative languages such as VHDL, SDL, or C++.

During *data type exploration* unconstrained system functions are constrained by replacing infinite data types with fixed size data types. The resulting *data type decisions* serve as input to the next step in the design space exploration process, *architecture exploration.*

Architecture exploration uses the unconstrained system model together with the data type decisions and a *design library.* The design library contains possible implementations for skeletons and library elements. The resulting *architecture decisions* define the details of interfaces and implementation necessary to generate the details in a VHDL model. An example of the activities in this phase is the exploration of sequential-parallel trade-offs of communication links.

Synthesis is done in two steps. First the unconstrained system model, guided by data type and architecture decisions, is synthesized into VHDL-Code for hardware and C-Code for software. In the second step the design is further processed by traditional hardware synthesis and software compilation.

## 4. COMPUTATIONAL MODEL

### 4.1 Definition

For a formal definition of the computational model we use the denotational framework of Lee and Sangiovanni-Vincentelli [6]. They define 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 ⊥ ("bottom"). Absent events are necessary to establish a total ordering among events for real time systems with variable event rates.

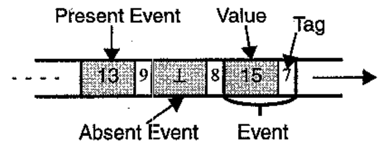


**Figure 2. A signal is a set of events**

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 generated instantaneously. There is no delay inside a process.

### 4.2 Modelling Language

We have chosen the functional language Haskell as our modelling language, as it is based on *formal semantics,* is *purely functional,* supports *higher-order functions,* has a *lazy evaluation* mechanism, provides a variety of control constructs to facilitate also the modelling of complex control flow and is executable to allow the simulation of the system model.

A Haskell program is a function, which consists of a composition of other functions. Functions produce only one result. However, a result can be a tuple (similar to a record) consisting of values of different data types.

## 4.3 System Modelling with Skeletons and Function Composition

Following the definition of our computational model in Section 4.1 we describe in this section how a system is modelled. 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. In our model we use *timed* signals, which can contain absent events. We define a data type `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 = [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 and a software interpretation. Thus, a system model, which is a composition of skeletons, has also an interpretation in hardware, software or a mixture of both.
- As skeletons are higher-order functions, the work on correctness-preserving transformations can be used to transform a system model into a more effective implementation model.

In the following we present two important skeletons and give a hardware interpretation for each of them.

The skeleton **`mapS`** is based on the higher-order function **`map`**, which applies a function `f` on all elements of a list. **`mapS`** can be interpreted as a combinatorial component with one input.

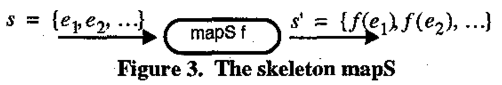


**Figure 3. The skeleton mapS**

The skeleton **`scanlS`** applies a function `f` on the events of a signal and an internal state `mem`. The result of the function `f` is used 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 data type of `mem`.

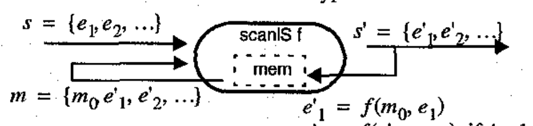

$s = \{e_1, e_2, \ldots\}$
scanlS f
mem
$s' = \{e'_1, e'_2, \ldots\}$
$m = \{m_0\, e'_1, e'_2, \ldots\}$
$e'_1 = f(m_0, e_1)$
$e'_i = f(e'_{i-1}, e_i)$, if $i > 1$

**Figure 4. 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 composition of skeletons. Hence each library element has a hardware interpretation. However, often an effective implementation is known for a certain library element and can be added to the design library.

This concept 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 (Fig. 5).

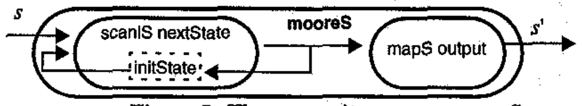


**Figure 5. The composite process mooreS**

## 5. SYSTEM SYNTHESIS

While the system model is only based on data dependences and thus abstracts from implementation details, such as a low-level communication mechanism, the use of skeletons makes it possible to give the system model a hardware and software interpretation. In the following discussion we use an ATM switch system as example.

Our design methodology leads to system models, which are composed of a hierarchy of functions. This results in a tree structure where the root of the tree is the *system function*. In this function tree we distinguish three layers with different composition techniques (Fig. 6). The top layer is the *system layer*, where composition is done by means of a set of equations. This layer can consist of several levels, depending on the size of the system. The middle layer is the *skeleton layer*, where functions are composed by composition of skeletons. Skeletons cannot be hierarchical because skeletons contain both structural and timing information. The bottom layer is the *elementary layer* consisting of elementary, pure combinatorial functions with arbitrary hierarchical depth.

Our synthesis methodology reflects the layered structure of the system model. We divide the synthesis task into three sub-tasks, each of them corresponding to one layer and discuss the synthesis of the system layer (Section 5.1), the skeleton layer (Section 5.2) and finally the elementary layer (Section 5.3).

We point out, that our method includes hardware and software synthesis. However, in this paper the focus lies mainly on hardware synthesis to VHDL.

### 5.1 System Layer

Functions on the system level are composed of a set of equations. They are synthesized into a netlist of software processes and hardware components. During hardware synthesis an entity interface is derived from the type declaration of the function. The functions on the right hand side of the equations result in components in the netlist, while the interconnection scheme is given by the function parameters, which work as equation variables. Fig. 7 illustrates how the function `oam` is synthesized

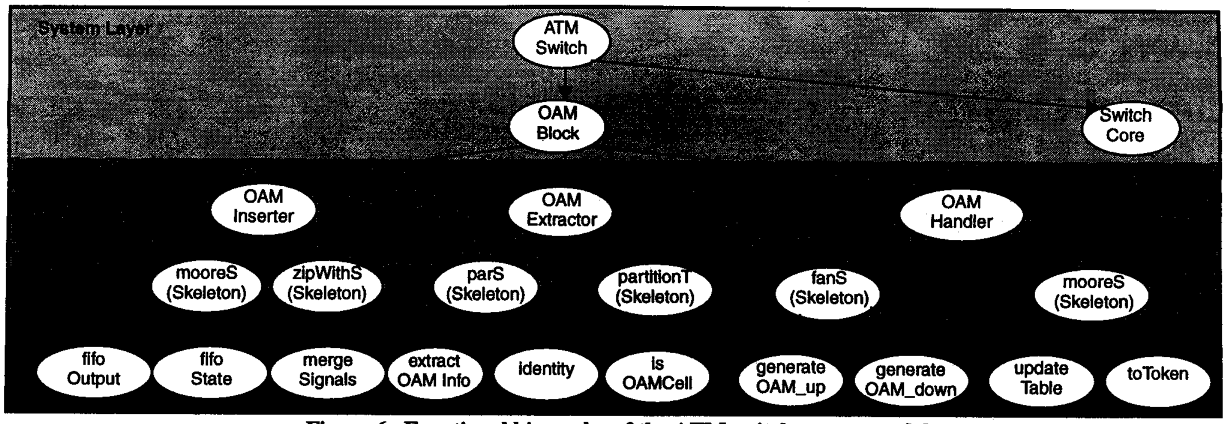


**Figure 6. Functional hierarchy of the ATM switch system model**

into a netlist.

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

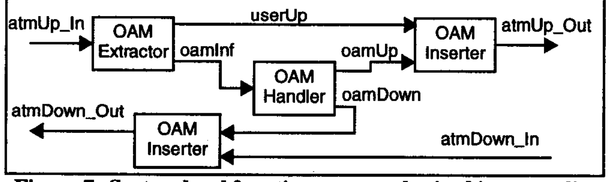


**Figure 7. System level functions are synthesized into a netlist**

## 5.2 Skeleton Layer

Each skeleton has a hardware and software interpretation, which is incorporated as a template in the synthesis library. During hardware synthesis a VHDL-template is modified depending on the datatype of the skeleton and the used elementary function.

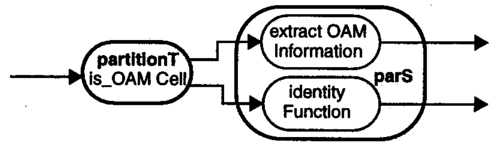
A skeleton, which does not contain state information, is synthesized into a VHDL-function. We illustrate this by the synthesis of the skeleton `partitionT` which is part of the function `oam_Extractor`. `partitionT` takes the function `is_oamCell` as argument and returns a record of `ATM_Tokens` as result. Fig. 8 shows the synthesized result. Note that only the boldface parts had to be changed from the VHDL-template for `partitionT`.

Skeletons which contain state information are synthesized into a VHDL-implementation of a FSM. This is illustrated by the synthesis of the library element `mooreS`, which is part of the function `oam_Handler`. It is synthesized into three VHDL processes implementing an FSM. Here, the process for the next state decoder is synthesized from the elementary function `updateTable` and the process for the output decoder from `toToken`. In addition a register process for the memory elements using an *event clock* is inferred as a direct consequence of `mooreS` as it contains state information. This event clock is the clock signal used in the VHDL model to define a synchronous hardware implementation. The state parameter models the initial state and is interpreted as the reset state (Fig. 9).

## 5.3 Elementary Layer

Elementary functions are combinatorial and thus synthesized into

```
oam_Extractor :: ATM_Channel ->
                   (VPI_Record_Channel, ATM_Channel)
oam_Extractor atmCells
  = (parS extract_oam_Information id
    . partitionT is_oamCell) atmCells
```



```
FUNCTION partitionT_is_oamCell(input : IN ATM_Token)
  RETURN Tuple_ATM_Token_ATM_Token IS
     VARIABLE result : Tuple_ATM_Token_ATM_Token;
BEGIN
  IF is_oamCell(input) = true THEN
     result.output1 := input;
     result.output2.event := Absent;
     result.output2.value := input.value;
  ELSE
     result.output1.event := Absent;
     result.output1.value := input.value;
     result.output2 := input;
  END IF;
  RETURN result;
END partitionT_is_oamCell;
```

**Figure 8. Skeletons without state information are transformed into VHDL-functions**

```
oam_Handler :: VPI_Record_Channel ->
                 (ATM_List_Channel, ATM_List_Channel)
oam_Handler oamInf
  = (fanS generateOAM_up generateOAM_down)
  . (mooreS updateTable toToken
              (emptyTable, startTime)) oamInf
```

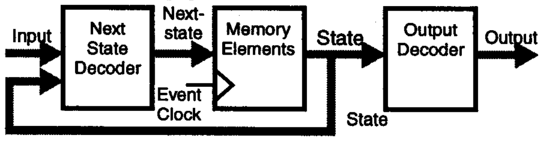


**Figure 9. Synthesis of mooreS into a FSM**

VHDL- or C-functions. We illustrate this by hardware synthesis of the elementary function `is_oamCell` in Fig. 10.

The function declaration is derived from the type declaration of `is_OAMCell`. Pattern Matching is synthesized into `IF`- or `CASE`-statements.

## 5.4 Synthesis Results

We have manually transformed the Haskell model of the subsystem oam (Fig. 7) with simplified data types for the ATM

```
is_oamCell :: ATM_Token -> Bool
is_oamCell Absent                              = False
is_oamCell (Present EmptyCell)                 = False
is_oamCell (Present (UserCell vci (VPI vpi)))= False
is_oamCell (Present (F5_OAM vci state))      = True
is_oamCell (Present (F4_OAM (VPI vpi) state))= True

FUNCTION is_oamCell(atmCell : IN ATM_Token)
   RETURN boolean IS
      VARIABLE result : boolean;
BEGIN
   IF atmCell.event = Absent THEN
      result := false;
   ELSE
      CASE atmCell.value.cellType IS
         WHEN UserCell  => result := false;
         WHEN EmptyCell => result := false;
         WHEN F4_OAM    => result := true;
         WHEN F5_OAM    => result := true;
      END CASE;
   END IF;
   RETURN result;
END is_OAMCell;
```

**Figure 10. An elementary function is synthesized into a VHDL-function**

cells into a synthesizable VHDL model according to the described method with the Synopsys Design Compiler using the LSI_10K library. The size of the synthesized design is 2611 gates.

In addition, we have compared the synthesized results for the FIFO part of the design with a manually written design with different timing constrains. As illustrated in Table 1 the number of gates generated with our synthesis method is only slightly higher than for a design directly written in VHDL.

**Table 1: Synthesis results for the FIFO part**

| Frequency | Manual Design (number of gates) | Synthesized Design (number of gates) | Difference (in percent) |
|---|---|---|---|
| 20 Mhz | 645 | 671 | 4.03% |
| 40 Mhz | 680 | 692 | 1.76% |
| 50 Mhz | 692 | 758 | 9.54% |

## 6. CONCLUSION

We presented a novel design methodology for system design. We combine 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. This means, that the system model abstracts from implementation issues such as detailed communication mechanisms and its formal nature supports formal methods and verification. However, despite of its high abstraction level, the use of skeletons makes it possible to interpret the system model as hardware, software or a mix of both leading to an efficient implementation.

The design flow consists of a design exploration and a synthesis phase. The design exploration results in design decisions which are input to the synthesis process. We discussed mainly the synthesis of hardware from the system model, which is organised in three layers. Our method reflects these layers and uses suitable synthesis strategies for each of them. We illustrated the feasibility of our method by the synthesis of the OAM subsystem of an ATM switch.

We will focus our future work on (1) software synthesis, (2) communication synthesis between heterogeneous components under consideration of memory structures (message passing, shared memory), (3) architecture exploration and (4) the connection of formal verification methods to our design methodology.

## 7. REFERENCES

[1] A. Benveniste and G. Berry, "The Synchronous Approach to Reactive and Real-Time Systems", *Proceedings of the IEEE*, Sept. 1991.

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

[3] F. Boussinot and R. de Simone, "The ESTEREL Language", *Proceedings of the IEEE*, pp. 1293-1304, September 1991.

[4] S. Edwards, L. Lavagno, E. A. Lee and A. Sangiovanni-Vincentelli, "Design of Embedded Systems: Formal Models, Validation and Synthesis", *Proceedings of the IEEE*, pp. 366-390, March 1997.

[5] N. Halbwachs, P. Caspi, P. Raymond and D. Pilaud, "The Synchronous Data Flow Programming Language LUSTRE", *Proceedings of the IEEE*, Vol. 79, No. 9, pp. 1305-1320, September 1991.

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

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

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

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

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

[11] I. Sander and A. Jantsch, "Formal System Design Based on the Synchrony Hypothesis, *Functional Models, and Skeletons", Proceedings of the IEEE International Conference on VLSI Design*, 1999.

[12] D.B. Skillicorn and D. Talia, "Models and Languages for Parallel Computation", *ACM Computing Surveys*, pp. 123-169, June 1998.

[13] Y. Li and M. Leeser, "HML: An Innovative Hardware Description Language and its Translation to VHDL", *Conference on Computer Hardware Description Languages and Their Applications (CHDL)*, 1995.

[14] G. Jones and M. Sheeran, "Circuit Design in Ruby", in *Formal Methods for VLSI Design*, North Holland, ed. J. Staunstrup, 1990.