System Synthesis Based on a Formal Computational Model 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 used 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 synthesis method which bridges the gap between a highly abstract functional model and an efficient hardware implementation. The functional model is strictly formal and based on formal semantics, a pure functional language, and the synchrony hypothesis. However, the use of skeletons in conjunction with a proper computational model allows a hardware interpretation, where the structure is given by skeletons and the combinatorial logic by elementary functions. Thus, without compromising the formal properties we offer an effective modelling technique on a high abstraction level which is still natural for hardware designers, and is the basis for synthesis into an efficient implementation. Furthermore, we describe a design methodology which uses the modelling concepts and the synthesis method. It contains a design exploration phase and defines how and when design decisions are formally introduced into the synthesis process. Finally, we illustrate design space exploration and synthesis with a FIFO component taken from an ATM switch.

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. 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 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 a hardware implementation.

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 abstract and 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 complicated asynchronous or synchronous communication mechanisms. 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.

Our synthesis method is based on skeletons which provide a structural hardware 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 a VHDL model suitable for processing by logic synthesis tools.

The rest of the paper is organized as follows. Section 2 discusses related work, Section 3 gives an overview of the design methodology, Section 4 introduces the computational model and skeletons, and Section 5 illustrates the design space exploration phase. In Section 6 we present the synthesis method with a FIFO from an ATM switch serving as illustrating example. In Section 7 we discuss what research remains to be done to make this approach usable in practice, but we 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 therefore we do not expect insurmountable problems.

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 [23], VHDL, or SpecCharts [14]. 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 or Erlang [17], 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.

Skillicorn and Talia [19] present a hierarchy of models for computation on parallel architectures. This is very relevant also for specification and implementation of hardware systems because many of the problems and challenges are similar or even identical. Depending on what information is explicit in a model they distinguish 6 levels, i.e. (1) "nothing explicit", (2) "parallelism explicit", (3) "parallelism and decomposition explicit", (4) "parallelism, decomposition, and mapping explicit", (5) "parallelism, decomposition, mapping, and communication explicit", and (6) "parallelism, decomposition, mapping, communication and synchronization explicit". According to this scheme our modelling approach falls into the "nothing explicit" level, with parallelism, mapping and communication implicit in the model and therefore left to be decided by the synthesis and design process. 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 [19].

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, one group targeting data flow applications (e.g. Lustre [5], Signal [8]), the other targeting control oriented applications (e.g. ESTEREL [3], Statecharts [6]). 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 [11] used the functional language Haskell [9] 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, 10].

The parallel programming community has used functional languages to derive parallel programs from a functional specification [12, 13]. They use skeletons to structure a problem. This formulation is then transformed into an efficient implementation for a chosen parallel computer architecture.

Only few attempts to synthesize hardware from an abstract functional specification have been published. All of them differ significantly from our approach. Ruby [21] 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. In contrast our approach is based on a functional language, addresses data flow and control dominated applications, uses a fully fledged functional language, and links to commercial logic synthesis tools rather than dealing with layout directly. HML [22] is a hardware description language based on Standard ML, which is a functional language similar to Haskell used in our approach. However, HML attempts to replace VHDL or Verilog as hardware description languages, while we propose a hardware and system specification concept on a significantly higher abstraction level with a very different computational model. In [22] 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

Our design methodology is illustrated in Fig. 1. System design starts with the 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. 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

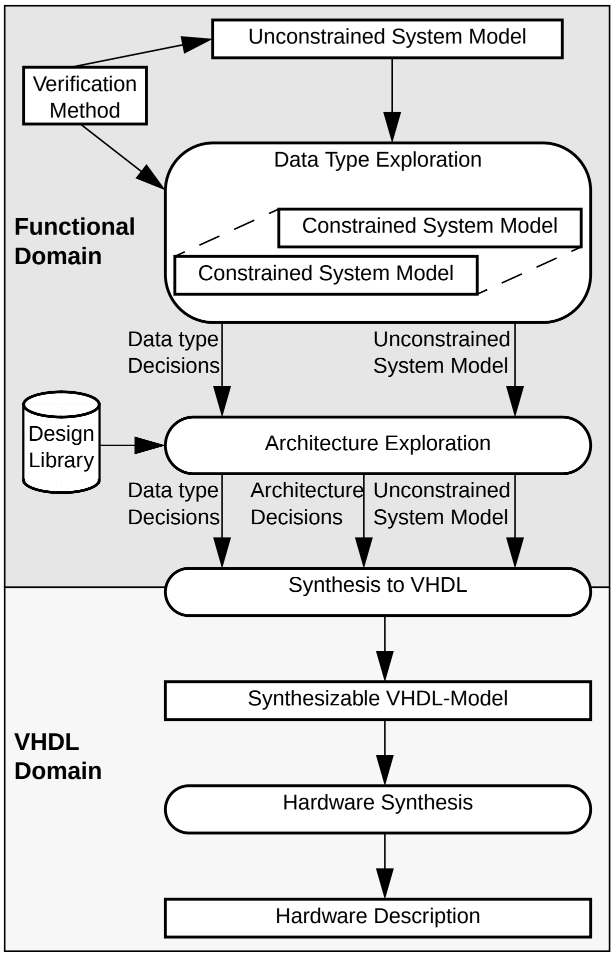


Figure 1. Design Methodology

C++.

During *data type exploration* unconstrained system functions are constrained by replacing infinite data types with fixed size data types. In this way we obtain several *constrained functional* system models which are simulated and analysed to determine the suitable constraints, such as the size of buffers. Also, they can be verified with the same method as used for the unconstrained system model to guarantee that the functional requirements are fulfilled. As part of the future work we want to connect our system to available formal verification and analysis tools, similar to the approach taken in the HAWK project [18]. The result of the data type exploration process are *data type decisions* which describe the specific constraints established during this phase. 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. Architecture exploration is part of our future work, but is planned to be done by means of cost models and estimation techniques. The result of this phase are *architecture decisions*, which are an input to the *synthesis process*. Architecture decisions define the details of interfaces and implementation necessary to generate the details in a VHDL model. Examples of architecture decisions relating to the data rate and the sequential-parallel trade-off of interfaces are discussed in Section 5.

Synthesis is done in two steps. First the unconstrained system model, guided by data type and architecture decisions, is synthesized into a synthesizable VHDL-model on RTL-level. In the second step the VHDL-model is synthesized with a logic synthesis tool into a netlist for a chosen technology.

We discuss the computational model in Section 4, design space exploration in Section 5, and synthesis in Section 6.

4. Computational Model

4.1. Definition

For a formal definition of the computational model we use the denotational framework of Lee and Sangiovanni-Vincentelli [7]. 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.

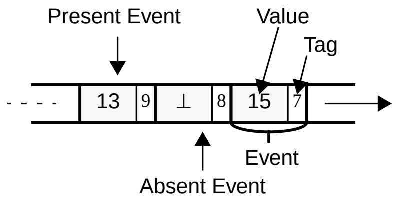


Figure 2. A signal is a set of events

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.

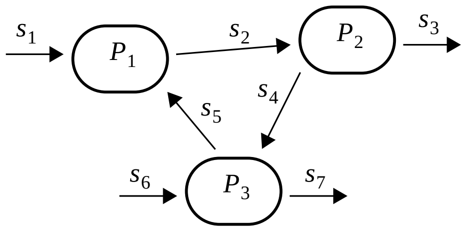


Figure 3. A system is modelled with concurrent processes

4.2. Modelling Language

We have chosen the functional language Haskell as our modelling language, as it

- is based on *formal semantics* and *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
- is executable to allow the simulation of the system model

In addition Haskell offers some other versatile facilities as *pattern matching* and a powerful *type system,* which can be used for verification purposes.

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.

We introduce some of the properties of Haskell with the higher-order function **`map`**, which applies a function to 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`).

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 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.

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 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, 10] 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`** (Section 4.2), which recursively applies a function `f` on all elements of a list. **`mapS`** can be interpreted as a combinatorial component with one input.

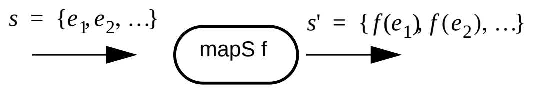
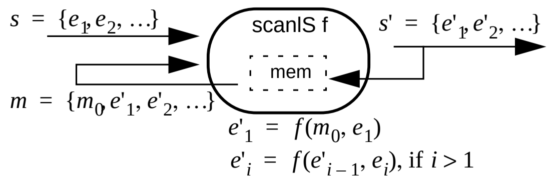


Figure 4. The skeleton mapS

The skeleton **`scanlS`** applies a function `f` on the events of a signal and an internal state `mem`. The result of the function `f` 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 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 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 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. 6).

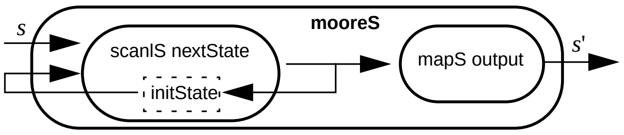


Figure 6. The composite process mooreS

We use **`mooreS`** to constitute a new process **`unconstrainedFifoT`**, that is used to model an *unconstrained FIFO*, which 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 which carries values (3). We model the unconstrained FIFO by means of the function **`mooreS`** and two elementary functions `fifoState` and `fifoOutput`.

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

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

5. Design Space Exploration

Starting with an unconstrained system model of an ATM Switch with operation and maintenance functionality [15, 16] (Fig. 7), we illustrate the design exploration process, which includes data type exploration and architecture exploration.

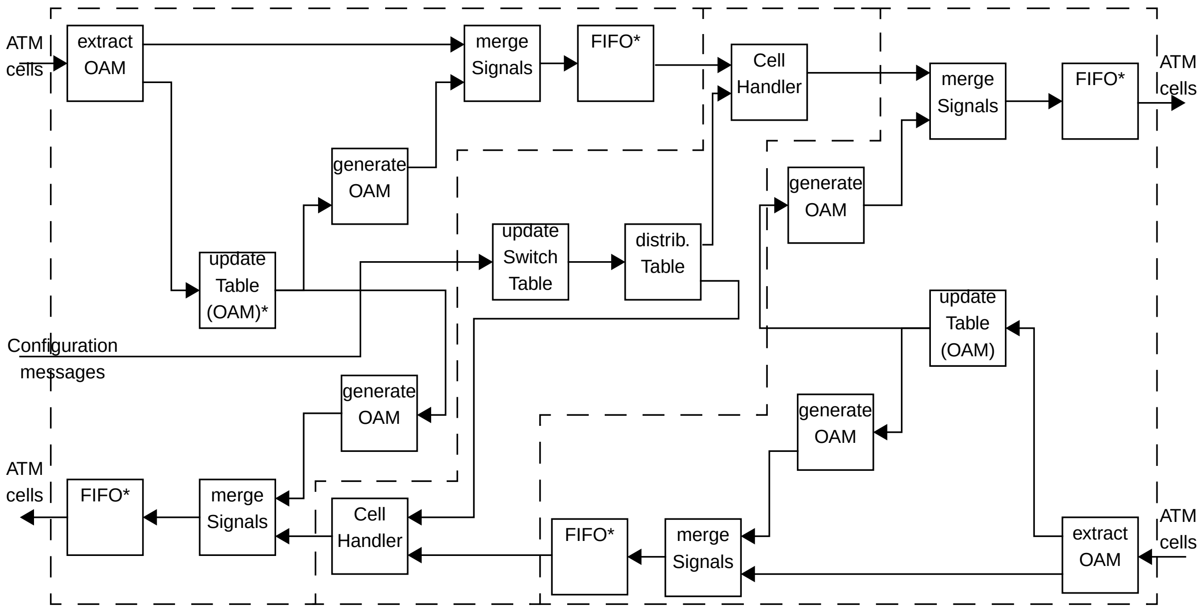


Figure 7. Unconstrained system model of an ATM switch with operation and maintenance functionality

During *data type exploration* we replace unconstrained system functions by constrained functions. The constrained system model is then verified with the same verification method, which proved the correctness of the unconstrained system model; today this is done by simulation. We illustrate this with a FIFO in the ATM switch. The unconstrained FIFO is replaced by constrained FIFOs, which

- have a fixed buffer size
- can only consume a fixed number of items during one event cycle

The constrained FIFO is modelled similar to the unconstrained FIFO (Section 4.3). We obtain a template by replacing the function `fifoState` with a new function `constrainedFifoState` with two additional parameters `b` for the buffer size and `i` for the number of parallel inputs.

```
constrainedFifoTemplate :: Int -> Int
                        -> Timed [a] -> Timed a
constrainedFifoTemplate b i
= mooreS (constrainedFifoState b i) fifoOutput []
```

The function `constrainedFifoState` behaves as `fifoState`, except that it uses bounded lists, which means that the lists are cut, if they have more elements as specified in the parameters `b` and `w`.

We can then build instances of constrained FIFOs by specifying the parameters b and w.

```
constrainedFifoT_b8_i4 :: Timed [a] -> Timed a
constrainedFifoT_b8_i4
   = constrainedFifoTemplate 8 4
```

Note, that the type of the constrained FIFO is identical with the data type for the unconstrained FIFO, which makes it possible to replace the unconstrained FIFO by a constrained FIFO without adjustment, and an arbitrary mix of constrained and unconstrained functions can be simulated. If verification shows, that the constrained system model still fulfils the system requirements, we obtain the following *data type decisions* for the FIFO:

- the buffer has to be dimensioned for eight items
- at maximum four items can be consumed during an event cycle

A FIFO with the named design characteristics can be implemented with different architectures. During *architecture exploration* possible implementations for skeletons and library elements from the *design library* are compared in order to get the most effective implementation, the result of this process are *architecture decisions*. Fig. 8 shows two possible implementations for the FIFO. The first one uses parallel inputs, while in the second implementation the inputs are consumed at a higher data rate compared with the event data rate. Other intermediate solutions are also possible. Fig. 8 illustrates how the unconstrained FIFO is transformed into a hardware implementable architecture by data type and architecture decisions.

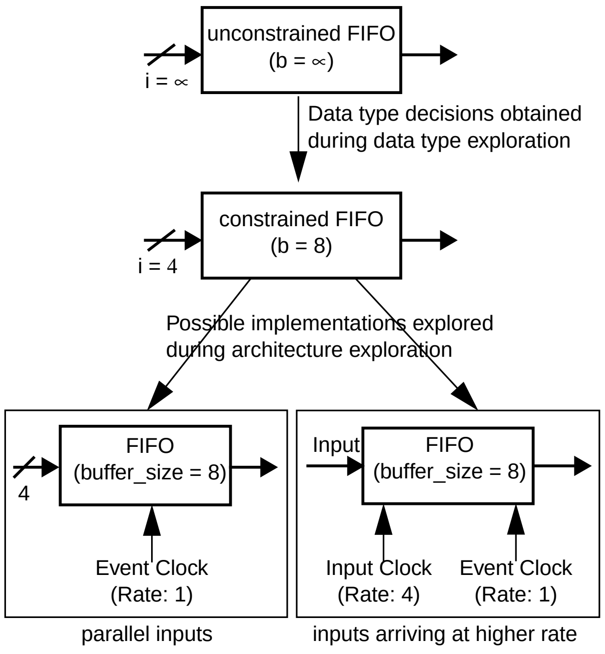


Figure 8. Data type and architecture exploration

We point out that the functional system model for the FIFO does not restrict the choice of an architecture since the unconstrained model does not imply an architecture.

6. Synthesis

Synthesis is done in two steps. First we transform the unconstrained system model into a synthesizable VHDL model. Here the synthesis process uses the results of the design exploration, the data type and architecture decisions. Fig. 9 shows how the synthesis process is divided into several sub-tasks. Finally the VHDL model is synthesized with a logic synthesis tool.

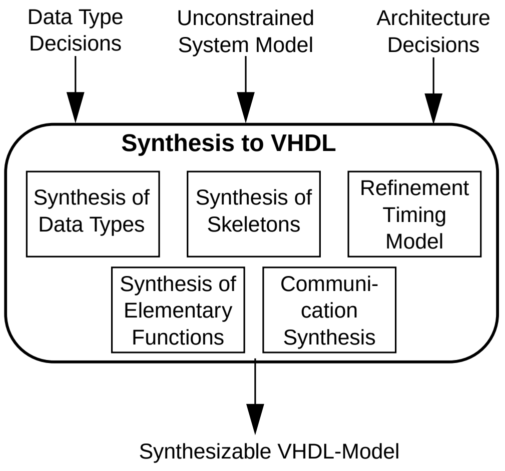


Figure 9. Synthesis to VHDL

We illustrate this through the synthesis of the process `unconstrainedFifoT`. In particular we focus on the refinement of the timing model (Section 6.1), skeleton synthesis (Section 6.2), the synthesis of lists (Section 6.3), and the synthesis of elementary functions (Section 6.4).

The process `unconstrainedFifoT` is defined as follows:

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

fifoState :: [a] -> Token [a] -> [a]
fifoState []     Absent       = []
fifoState (x:xs) Absent       = xs
fifoState []     (Present ys) = ys
fifoState (x:xs) (Present ys) = xs ++ ys

fifoOutput :: [a] -> Token a
fifoOutput []     = Absent
fifoOutput (x:xs) = Present x
```

The design is synthesized with the following *data type decisions*:

1. the buffer is dimensioned for eight items
2. at maximum four items are consumed during an event cycle

Our *architecture decision* is to choose the FIFO-architecture with parallel inputs.

6.1. Refinement of the Timing Model and Signals

We use a synchronous timing model, where a signal carries events, which can either be present and have a value or absent. The time period between two events is called an *event cycle*. We use an *event clock* to partition a timed signal into a set of events. This event clock is the clock signal used in the VHDL model to define a synchronous hardware implementation. To indicate if an event is present or absent we use additional *control signals*. Thus a timed signal is transformed into a record, where the first part is a value of the signals data type and the second part is a control signal, indicating if an event is present or not. The data type `Timed a` (Section 4.3) is transformed into the following record in VHDL.

```
TYPE Timed_A IS RECORD
                  value : A;
                  is_present : boolean;
                END RECORD;
```

6.2. Skeleton Synthesis

The process `unconstrainedFifoT` is based on the library element **`mooreS`**. All skeletons have a hardware interpretation and this means that `mooreS`, which is composed of the skeletons **`mapS`** and **`scanS`**, can be interpreted as a Moore-FSM. Fig. 10 shows the structure of a Moore-FSM, which can be directly modelled in VHDL with three parallel processes. The processes for the next state decoder and the output decoder are combinatorial, while the process for the memory elements includes registers.

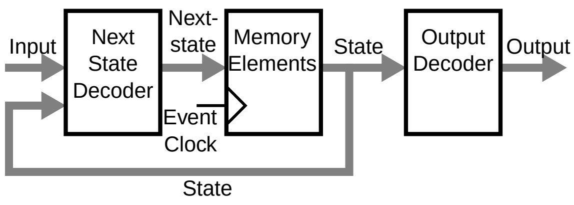


Figure 10. Finite State Machine of Moore-type

The library element `mooreS` has two functions and one state value as parameters. The first function, `fifo-State`, models the *next state decoder*, while the second function, `fifoOutput`, models the *output decoder*. The synthesis task is to transform these functions into a synthesizable VHDL process. A register process for the *memory elements* is inferred as a direct consequence since **`mooreS`** contains state information. The state parameter, here the empty list, models the initial state and is interpreted as the reset state of the Moore-FSM.

6.3. Synthesis of Lists

A list is implemented as a bounded list, which means that the list has a maximum number of items. This number is a result of the data type exploration. In VHDL a bounded list is implemented as a record with an array and a positive number; the number specifies how many items are valid in the list (Fig. 11).

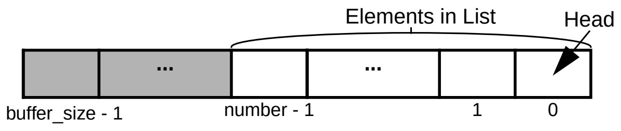


Figure 11. Implementation of a bounded list

The following data type is synthesized for the state of the FIFO, where the number of elements is given by the first data type decision, i.e. the buffer is dimensioned for eight items.

```
TYPE List_A_Buffer_Size IS RECORD
        number : natural RANGE 0 TO buffer_size;
        item : Array_A_Buffer_Size;
     END RECORD;
```

6.4. Synthesis of Elementary Functions

The functions `fifoState` and `fifoOutput` are transformed into combinatorial VHDL processes. We illustrate the transformation method with the function `fifoState` which is transformed into the process `OUTPUT_DECODER`.

```
fifoOutput []     = Absent      -- Pattern 1
fifoOutput (x:xs) = Present x   -- Pattern 2
```

The function `fifoOutput` uses *pattern matching*, which are transformed into `IF`- or `CASE`-statements in VHDL. The first pattern matches if the list, modelling the state of the FIFO, is empty. In this case the output is an absent event. Otherwise the second pattern matches, and the output is the first element of the FIFO buffer (`state`).

```
OUTPUT_DECODER : PROCESS(state)
BEGIN
   IF state.number = 0 THEN
      output.is_present <= false;    -- Pattern 1
      output.value  <= 0;
   ELSE
      output.is_present <= true;     -- Pattern 2
      output.value <= state.item(0);
   END IF;
END PROCESS;
```

6.5. Synthesis Results

We have manually transformed the Haskell model into a synthesizable VHDL model according to the described method. The VHDL model was validated by simulation and synthesized with the Synopsys Design Compiler using the LSI_10K library with different timing constrains. We compared the results with a manually written VHDL design for the FIFO. 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 example

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%

This synthesis of the FIFO is a very important step towards synthesizing the entire ATM switch because the FIFO is one of the most demanding blocks. It also shows that the method is fully automatable when all necessary design decisions are provided.

7. 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 as 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 a hardware structure 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 have discussed the synthesis method and illustrated it with a FIFO. The FIFO is a demanding example because it has a large internal state, which is traditionally a challenge for functional languages, and it serves as a buffering and synchronization device between producer and consumer blocks running potentially at different rates. The example shows how the unconstrained functional system model, with a very large design space, is subsequently and systematically transformed into a synthesizable VHDL model.

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.

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”, *Proceedings of the IEEE*, Vol. 79, No. 9, 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*, Vol. 85, No. 3, 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] D. Harel, “STATECHARTS: A Visual Approach to Complex Systems”, *Science of Computer Programming*, 8-3, pp. 231-275, 1987.

[7] 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.

[8] P. Le Guernic, T. Gautier, M. Le Borgne and C. de Marie, “Programming Real-Time Applications with SIGNAL”, *Proceedings of the IEEE*, Vol. 79, No. 9, pp. 1321-1335, September 1991.

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

[10] Alberto Pettorossi and Maurizio Proietti, “Rules and Strategies for Transforming Functional and Logic Programs”, *ACM Computing Surveys*, vol. 28, no. 2, pp. 361 - 414, June 1996.

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

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

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

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

[15] Wolfgang Horn, *Modelling of an ATM Multiplexer in a Network Terminal for a Mixed Hardware/Software Implementation*, Electronic Systems Design Laboratory, Department of Electronics, Royal Institute of Technology, Stockholm, report no. TRITA-ESD-1998-06, May 1998.

[16] Ingo Sander, Axel Jantsch, “Formal System Design Based on the Synchrony Hypothesis, *Functional Models, and Skeletons”, Proceedings of the IEEE International Conference on VLSI Design*, 1999.

[17] Joe Armstrong, Robert Virding, and Mike Williams, *Concurrent Programming in Erlang*, Prentice Hall, 1993.

[18] John Matthews, John Launchbury, and Byron Cook, “Microprocessor Specification in Hawk”, *International Conference on Computer Languages*, 1998.

[19] D.B. Skillicorn and D. Talia, “Models and Languages for Parallel Computation”, *ACM Computing Surveys*, vol. 30, no. 2, pp. 123-169, June 1998.

[20] Yanbing Li and Miriam Leeser, “HML: An Innovative Hardware Description Language and its Translation to VHDL”, *Conference on Computer Hardware Description Languages and Their Applications (CHDL)*, 1995.

[21] G. Jones and M. Sheeran, “Circuit Design in Ruby”, in *Formal Methods for VLSI Design*, North Holland, edited by J. Staunstrup, 1990.

[22] Yanbing Li and Miriam Leeser, “HML: An Innovative Hardware Description Language and its Translation to VHDL”, *Conference on Computer Hardware Description Languages and Their Applications (CHDL)*, 1995.

[23] A. Olsen, O Faergemand, B. Moeller-Pedersen, R. Reed, and J.R.W Smith, Systems Engineering with SDL-92, North Holland, 1995.