ROYAL INSTITUTE OF TECHNOLOGY

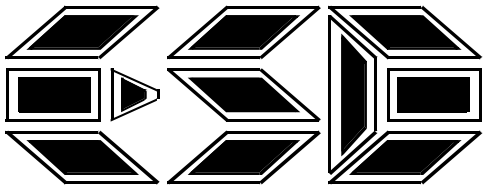
TRITA-ESD-99-2

ISSN 1104-8697

ISRN KTH/ESD/FOU--99/2--SE

A System Design Methodology Based on a Formal Computational Model

Wenbiao Wu
Axel Jantsch



ELECTRONIC SYSTEMS DESIGN LABORATORY
ROYAL INSTITUTE OF TECHNOLOGY
ESDLAB/KTH-ELECTRUM
ELECTRUM 229
S-164 40 KISTA, SWEDEN

A SYSTEM DESIGN METHODOLOGY BASED ON A FORMAL COMPUTATIONAL MODEL

Wenbiao Wu, Axel Jantsch
ESD Lab, Department of Electronics
Royal Institute of Technology
S-164 42 Kista, Sweden

Abstract

The modern electronic system design has become extremely complex due to the increasing demand of higher performance and improved functionality. In this paper, we address this problem with a system design methodology which is based on functional language and uses skeletons for design exploration. Transformational design method is also introduced during various design exploration stages. The final design is synthesized into a VHDL and C model which can be further implemented.

1. INTRODUCTION

The modern electronic system design has become extremely complex due to increasing demand of higher performance and improved functionality, which requires an immense investment of time and efforts to create and verify from the high-level abstract descriptions downwards. One of the characteristics of such systems is the co-existence of HW and SW blocks. With the complexity of system design, Intellectual Property blocks (IPs) such as embedded processor cores, DSP cores, filter components, interface units, real time kernels, etc. are also becoming increasing popular. An IP based design approach uses these building blocks and connects them together. The main problem here is the validation of the whole system design and the design of custom software and hardware for connecting the building blocks.

Traditional hardware system design forces the designer to partition hardware and software in an early stage and design them separately which results in a implementation difficult to verify and change. Usually a concurrency-process based system model follows the system requirement with complicated synchronous or asynchronous communication mechanism. In this process model, functions can't be easily moved from one process to another since this will require the redesign of communication structure between processes.

A fundamental aspect of system design is the specification process. We argue that there should be an intermediate layer between the system requirement and the concurrency-based process model. A specification that is written in a language whose syntax and semantics are formally defined. The development of a formal specification model will provide insights into and an understanding of both the system requirement and the implementation. This will on one hand reduce the errors in requirement, while on the other hand can be conveniently used to capture the essential functionality of a system, hence provides a good basis for future design and implementation.

Because of the formal nature of the specification model, which can be regarded as consisting of mathematical entities, it can be further analyzed by using some

mathematical methods which in turn can also be mechanized. For example, this formal model can be integrated with theorem provers, model checkers and other formal analysis and verification techniques, which can in some way guarantee the correctness of the specification model.

The preliminary result of specification language evaluation suggests that specification languages based on a functional paradigm can be more suitable than other languages based on communicating processes because the partitioning into concurrent processes is not determined in a functional model and therefore the communication need not be defined in detail. [JKS98]

Another advantage of formal specification is that this methodology can also be employed together with transformation techniques, i.e. the specification can be transformed through a series of correctness preserving steps to a final synthesizable system model which meets the system requirement and constraints imposed by designers (See Fig 1). The transformation result of each step is sufficiently close to the previous expression that the effort of verifying the transformation is not excessive. It can therefore be guaranteed that the final implementation is a true implementation of the specification. A transformation approach made up of a sequence of small steps is very effective and much more efficient than a theorem prover which is usually very difficult because the gap between implementation and a specification is much larger than the gaps between transformation steps. Of course, there are still difficulties in applying a purely transformational approach to large system design. However, the incorporation of this approach into the design methodology will offer the opportunities to improve the design process. For example, in HW/SW co-design, transformation techniques can be employed in the system's custom parts that are not covered by pre-designed building blocks (IPs).

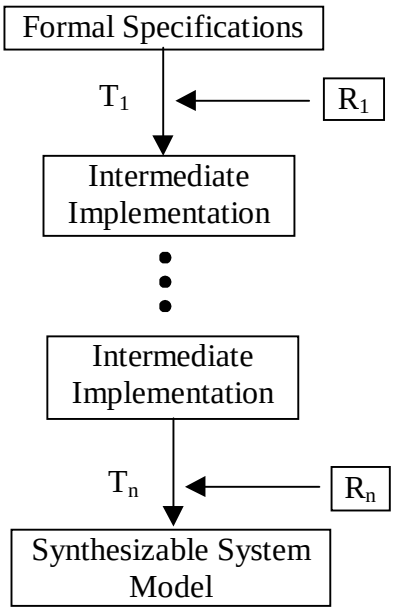


Fig 1: Transformational System Design

Our second assumption to system design is that system development is an iterative process. The specifications of large systems are not written as one homogenous document, but they are developed and improved over a long period of time by different people trying to meet changing requirements and constraints. So, this process should be supported by a technique, which allows the designer to add, modify and remove the entities or functions of his concern without a large impact on the rest of specifications.

2. RELATED WORKS

As we have argued in the first part that the design approach should be based on the use of formal models to describe the behavior at a high level of abstraction before a

decision on its decomposition into hardware and software components is taken. For a comprehensive overview of the computational models available see Edwards et al [ELLS97]. Many real-time systems are specified by means of concurrent processes which communicate asynchronously. However, since this model's closeness to the real architecture, we argue that this is not a good choice for a functional system model. There are mainly two drawbacks in this model. One is that many design decisions are already present in such a model (for example, the partitioning into processes). It is very difficult to correct a wrong design decision in the later design phases. The other drawback is that the complexity of the communication mechanism in the model is a major difficulty for the functional design exploration and the subsequent implementation.

The synchrony hypothesis [BB91] 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 [HCRP91], Signal [GGBM91]), the other targeting control oriented applications (e.g. ESTEREL [BS91], Statecharts [HAR87]). However, there is no language, which is good in both areas as elaborated in [BB91]. 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.

Transformational system design [WJ99] supports the idea of designing at a higher level of abstraction which implies describing the behavior of the required system and then transforming this into a structural and behavioral description at a low level, possibly in a hardware description language. Such kind of design methodology can reduce the overall design time and ensure the correctness of the implemented system. In [WJ99], transformational design is divided into four categories based on the system specification notations or languages. That is: imperative language based transformation systems, functional language based transformation systems, logic language based transformation systems and concurrent process based transformation systems. As pointed out in [WJ99], functional language as its name implies is based on the mathematical notation of functions. In contrast to the logic programming language where underlying model of computation is the relation, functional language has a more efficient operational behavior.

Haskell [HPF99] is a modern functional language that provides a common ground for research into functional languages and functional programming. However, now it has also been used in hardware design. In Hawk project [MCL98], Hawk is embedded into Haskell. In their work, Hawk has been used to specify and simulate the integer part of a pipelined DLX microprocessor. Lava [BCSS98] is a collection of Haskell modules. It assists circuit designers in specifying, designing, verifying and implementing hardware. O'Donnell [DON95] has developed a Haskell library called Hydra that models gates at several levels of abstraction ranging from implementations of gates from using CMOS and NMOS, up to abstract gate representations using lists to denote time-varying values. Bernhard Möller [MOL97] provided a deductive hardware design approach for basic combinational and sequential circuits. The goal of

his approach is the systematic construction of a system implementation starting from its behaviour specification. *Gofer/Haskell* has been used in his work to formulate a functional model of directional, synchronous and deterministic systems with discrete time. However, all of the above approaches are targeted on gate-level or circuit design which differ significantly from our approach. Reekie [REE95] used Haskell 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.

There are also some other languages for specifying hardware design. The Ruby language, created by Jones and Sheeran [JS90], is a circuit specification and simulation language based on relations, rather than functions. 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 [LL95] is a hardware modeling language based on the functional language 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. A direct translation of HML to VHDL is described in their paper [LL95], 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.

3. THE DESIGN METHODOLOGY

3.1 The Computational Model

Our computational model is based on the synchronous hypothesis which assumes the outputs of a system are synchronized with the system inputs, while the reaction of the system takes no observable time. In this methodology, we use the denotational framework of Lee and Sangiovanni-Vincentelli [LS97]. They defined a signal as a set of events, where an event has a tag and a value. A signal is shown in Fig 2. Here events are totally ordered by their tags. Events with the same tag are processed synchronously. A special value '⊥' (“bottom”) is used to model the absence of an event. Absent events are necessary to establish a total ordering among events for real time systems with variable event rates. Compared with finite state machines which are more operational in their nature, this approach leads to a computational model with a definite denotational flavor.

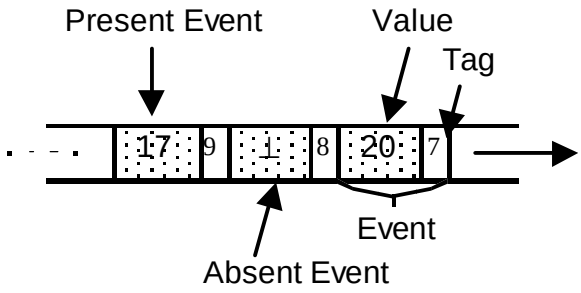


Fig 2: A signal is a set of events

3.2 The Modeling Language

We choose Haskell, a purely functional language, as our modeling language. The reasons for this are the following:

- Haskell is based on formal semantics and purely functional;
- Supports higher-order functions;
- Polymorphism allows generic formulations and hence supports reuse;
- All the specification is executable which allows the simulation of the system model.

3.3 Skeletons Used in the Functional Model

A skeleton is a higher-order function which is used to model elementary process. It takes elementary functions and signals as input parameters and produces signals as output. In design, we can define an elementary function as a function that is combinatorial and does not include any timing behaviour.

For example, we can define add :: (Int, Int) -> Int as a simple adder. The definition of map add :: [(Int, Int)] -> [Int] will be its clocked equivalent. Here map is defined as:

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

3.4 Transformation Rules and Patterns

The system specification is written in a functional language. Transformation-based design exploration is employed to transform the initial specification into a synthesizable architecture model during the process. A design library is used to facilitate this process.

The research of program transformation has been active for several decades since the pioneering work of Burstall and Darlington [BD75]. Previous works in this area will provide a solid basis for our design methodology.

In the design process, each small step of transformation is the result of applying one transformation rule to the previous expression. Transformation rules include:

- Definition: Introduce new functions to the system model.
- Unfolding: Replace the application of a function with its definition.
- Folding: This is the opposite of unfolding.
- Abstraction: Abstract common expressions used in system models into one separate function to be reused.
- Algebraic Rules

Transformations are also guided by transformation patterns which correspond to different design exploration stages and purposes. They are:

- Combinatorial function transformations
- Data type transformations
- Skeleton-based transformations

We will give some transformation examples in the following sections.

3.5 Intermediate Design Steps

3.5.1 The Unconstrained Functional Model

System design (Fig. 3) starts with the development of an unconstrained functional system model which is based on a synchronous computational model, a functional modelling language (Haskell) and the use of skeletons. 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 system model is unconstrained in the sense that it uses unconstrained data types, such as infinite lists or trees. The nature of the unconstrained system model leaves a wide design space. During this stage, functions can be added or removed as needed. The whole system functionality is acquired through the use of skeletons.

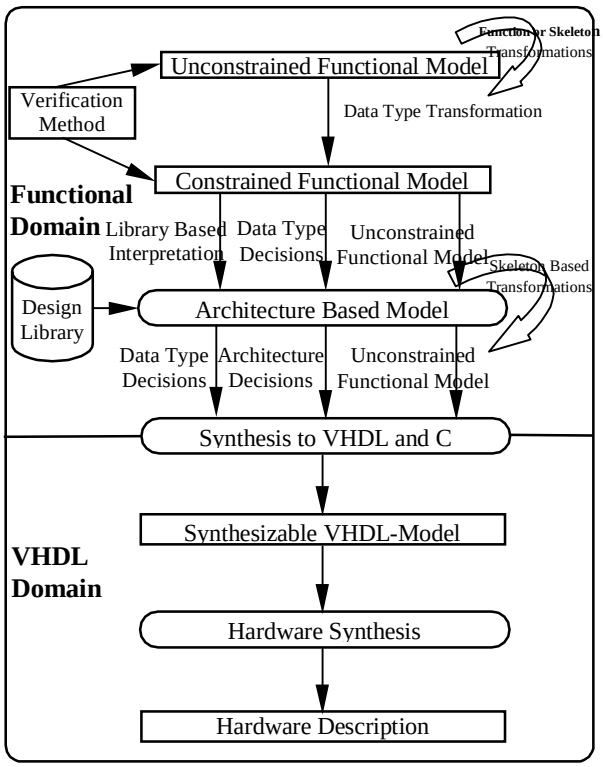


Fig 3: System Design Methodology

Functional transformation is employed on this model to meet the user specified constraints or to optimise system design. So, at first the designer just provides an initial correct but maybe inefficient system model. In the following design explorations in this stage, the designer can transform this system model using the transformation rules to get a new functional model which is not only correct but also satisfies the system requirements. Here we present some transformation examples.

Example 1: As described, a signal is a list of events. Now, we want to process the particular events which satisfy some property. To simplify this, let's say the property be P which is a function p :: Event -> Bool and the operation we want to do be sum :: Signal -> int which returns the sum of the event's values in the signal.
So, the functions can be:

```
sumByProperty :: Signal -> Int
sumByProperty s = sum (properList (s))                    (1)

properList :: Signal -> Signal
properList [] = []
properList (x:xs) = if p x then x : properList xs else properList xs

sum :: Signal -> Int
sum [] = 0
sum (x:xs) = x +  sum xs
```

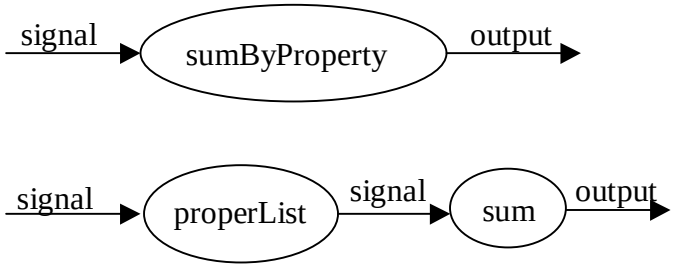


Fig 4: Function sumByProperty

Here, properList is a function used to generate the list where the event satisfies property P. sum is a function used to sum the events in a signal. This is also shown in Fig 4.

By using function composition strategy, we can avoid the construction of an intermediate list. This list is passed to sum in (1).

First, we define a new function :

```
newSum :: Signal -> Int
newSum = sum•properList

newSum []       = sum•properList [] = sum (properList [])  = sum [] = 0
newSum (x:xs)  = sum•properList (x:xs) = sum (properList (x:xs))
                = sum (if p x then x : properList xs else properList xs)
                = if p x then sum (x : properList xs) else sum (properList xs)
                = if p x then x + sum•properList xs else sum•properList xs
                = if p x then x + newSum xs else newSum xs          (2)
```

Fig 5: Function newSum

So, here we have transformed the old functions into a new function newSum (Fig 5) which doesn't need to generate an intermediate list and is much more efficient. This will reduce the communication and buffer size if implemented in hardware design later.

Example 2: The final function in the previous example can be further optimised if it is to be implemented in SW part. Function newSum gets the final result through successive recursive calls which requires O(n) space to remember the arguments of operator '+'. Also, the sum of the lists can only start until the recursion is completely unrolled. To solve this problem, we can rewrite the function as:

```
newSum s = newSum' s 0
And,
newSum' :: Signal -> Int -> Int
newSum' [] result = result
newSum' (x:xs) result
```

```
| p x          = newSum' xs (result+x)
| otherwise  = newSum' xs result
```

The execution of these two functions is shown in Fig 6 and Fig 7. In this example, the

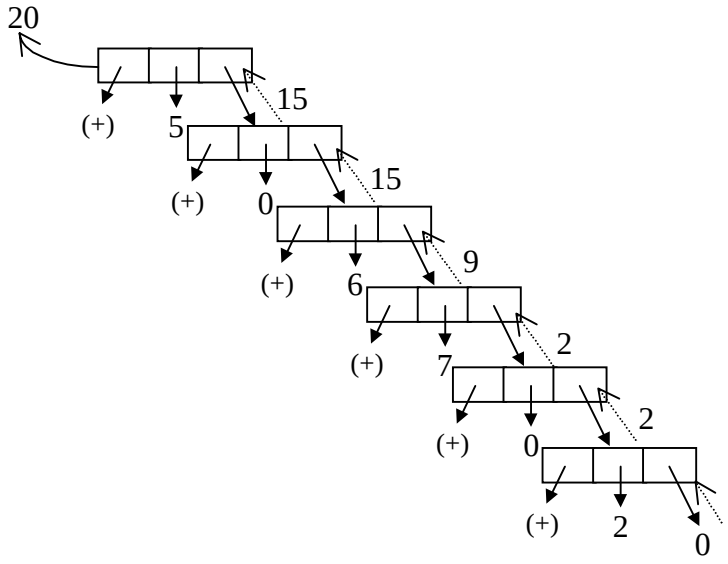


Fig 6: The execution of function newSum

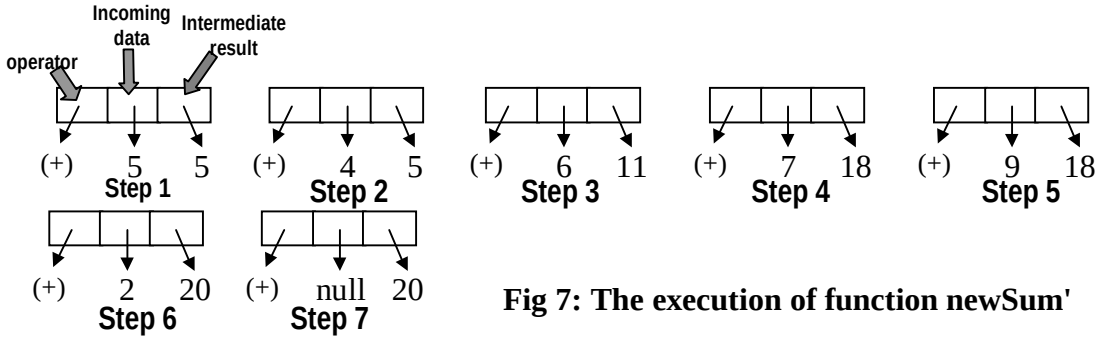


Fig 7: The execution of function newSum'

input is ***5***, 4, ***6***, ***7***, 9, ***2***. The events in bold and italic types satisfy property P. (Here we have ignored the tags.)

This transformation will reduce the space needed to O(1) and is much more efficient since we already get the final result when the recursive calls is completed.

Usually, it will be tedious to do such transformations by hand. However, the mathematical nature of transformation decides that these steps can be mechanized and proved correct by tools.

3.5.2 The Constrained Functional Model

The constrained functional model is developed from the unconstrained model using data type exploration. This intermediate stage is needed in order to synthesize the functional model because it's difficult and inefficient to implement infinite data types in the final design. Some other data type related transformations are also included in this stage. The followings are the main transformations will be used in this stage.

- Transform lists to multi-lists or vice versa

Sometimes, we need to split a signal into several signals to be processed by several processes simultaneously to acquire high efficiency or vice versa. This is illustrated in Fig 8.

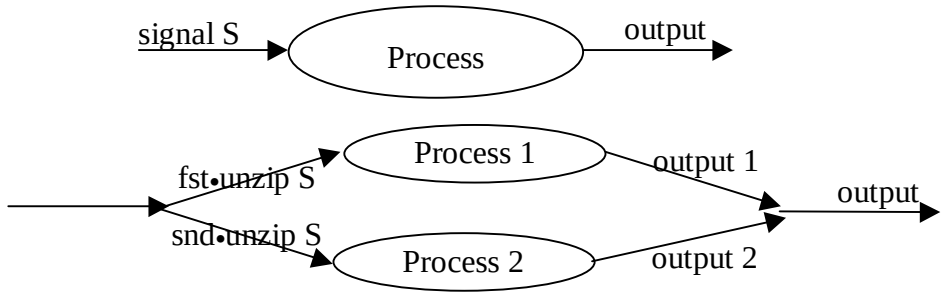


Fig 8: Transform list to two lists

Here, the signal is a list of pairs. And unzip is the function used to split the signal. Fst return the first element in the pair. Snd returns the second element in the pair.

```
fst :: (a, b) -> a                    snd :: (a, b) -> b
fst (x, y) = x                        snd (x, y)  = y

unzip :: [(a, b)] -> ([a],[b])
unzip [] = ([], [])
unzip ((x, y):ps) = (x:xs, y:ys) where (xs, ys) = unzip ps
```

We can also modify unzip to meet some special splitting criteria.

- Transform data structures (e.g. trees and lists.)

It's true in both hardware design and software design that the organization of data structure has a big influence on the system efficiency. In some cases, it's desirable to transform the input data structure to accommodate the specific HW or SW structure. For example, in the design of IP filter in a router, how the IP address table is organized is an important issue because searching the table costs most of the time. The IP address table can be organize either in lists or trees. However, from the algorithm and implementation point of view, it's better to implement the data structure in trees. Specific operations can then be performed on this structure. Usually, this kind of transformation will be based on the implementation the designer has in mind and thus is very powerful and efficient.

- Transform unbounded data types to bounded data types

We can use templates to transform unbounded data types into bounded data types [SJ99].

For example, an unconstrained FIFO is defined as:

```
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.

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 []
```

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
```

3.5.3 Architecture Based Model

The design library contains implementations for skeletons and library elements. Based on the unconstrained functional model, data type decisions and the design library, architecture based model can be obtained through a library-based interpretation. Some skeleton-based transformations are introduced here which is used to optimize system design.

- Merge separate skeletons

We can merge separate skeletons into one skeleton (see Fig 9). Here, skeletons are represented as FSMs, i.e. they have next state functions f_1 and f_3 and output functions f_2 and f_4. The final skeleton is also an FSM whose next state function is f_5 and output function is f_6. Skeleton merging is usually only a re-grouping of the existing

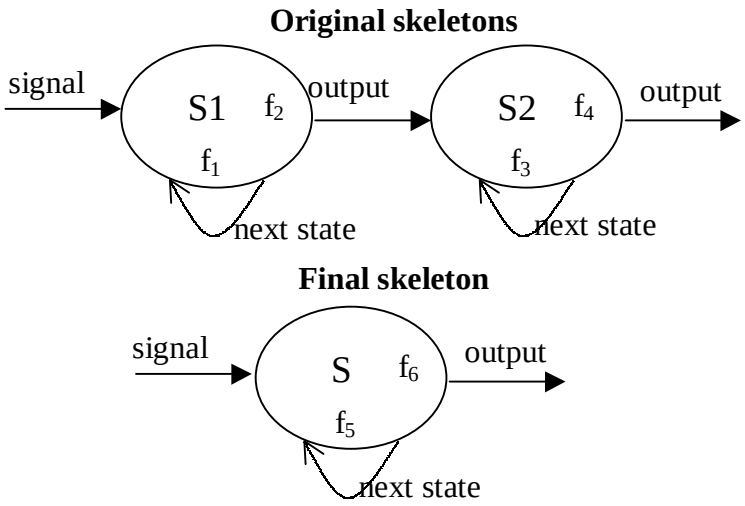


Fig 9: Combine two skeletons together

skeletons. However, as far as the communication between skeletons is concerned, regrouping the skeletons can sometimes minimize the communication overhead. So, this kind of transformation is usually based on library estimation.

- Split a skeleton into several skeletons

This transformation is opposite to the previous one and may be used when the desired skeleton doesn't have a direct hardware representation in the design library while the skeletons resulted can be found in the library. It also supports skeleton reuse, i.e. common sub-skeletons may be reused.

- Move functions between skeletons

The functions in skeletons can be easily moved in the functional model. Usually, the moving of functions is based on library estimation or user decisions. For example, in Fig 10, there are two skeletons. Skeleton B will use data C from skeleton A. Since C is a very large table in skeleton A, it's desirable to move the function get_entry in skeleton B to skeleton A to minimize the communication between two skeletons. The final skeletons are also shown in Fig 10.

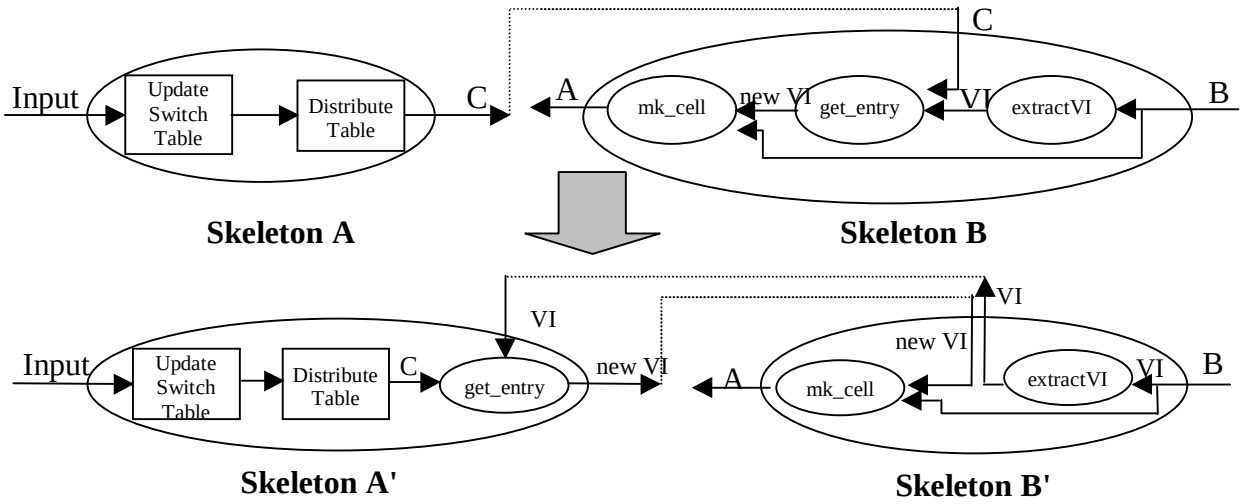


Fig 10: Move Functions between Skeletons

- Share elementary functions between skeletons

Some elementary functions in one skeleton may already exist in another skeleton. Thus, it will be more efficient to reuse the elementary function that has already been defined. For example in Fig 11, two skeletons s1 and s2 use the same elementary function 'mapS' at the end which is based on the higher-order function map and recursively applies a function f on all elements of a list. Based on library estimation, it is possible that we combine these two functions into one.

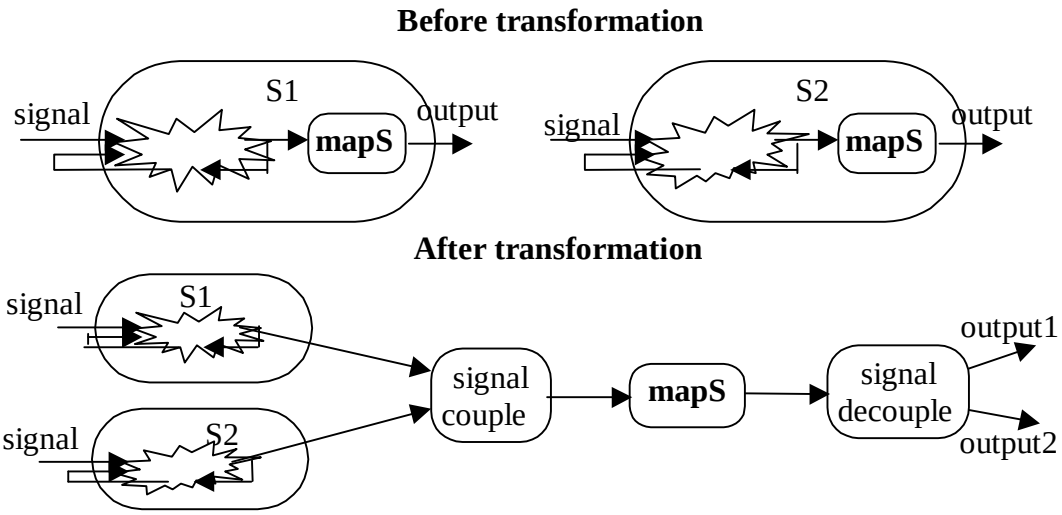


Fig 11: Elementary function reuse in skeletons

- Transform elementary functions inside skeletons

Besides the above transformations, we can also transform the elementary functions inside the skeleton as the methods described in section 3.5.1. This will also improved the performance of the skeletons. At the same time, some of the skeleton-based transformations can also be used in the unconstrained functional model.

3.6 Summary of Design Steps and Transformations

In the previous sections, we have introduced several design phases in our methodology and presented some design exploration examples. Since our model is based on the functional language, it's very natural and effective to employ transformation based design space exploration on this model. The main challenge here is how to combine different transformation rules and patterns with library estimation which will largely depend on the implementation of the design library.

Elementary function transformations and skeleton based transformation can be employed in all the design stages. However, the former is mostly used in early stages because in early stages design library is still not introduced and the designer usually have much wider exploration space for elementary function transformation. On the other hand, skeletons have hardware or software representations in the design library. It is much easier and more profitable to employ skeleton based transformtion in later stages.

4. SYNTHESIS INTO VHDL AND C MODEL

With the intermediate information we get during the design exploration, the synthesis can be done in a structural way. The information useful in synthesis stage includes data type decision, architecture decision and so on. In this methodology, the synthesis can be divided in two steps. First we transform the unconstrained functional model together with the design exploration results into a synthesizable VHDL and C model. Then the VHDL model can be further synthesized with a logic synthesis tool. The C code is implemented as the software part of the final implementation.

The first step mainly consists of the following parts:

- Synthesis of data types
  We map the data types used in Haskell to types in VHDL or C.
- Synthesis of skeletons
  The skeleton has a hardware interpretation in the design library. So, it can be directly modeled in VHDL.
- Synthesis of communications model
  We synthesize the synchronous timing model into the clocked signal in VHDL.
- Synthesis of combinatorial functions

5. CONCLUSION AND FUTURE WORK

We have provided a design methodology for HW/SW co-design system. The methodology is based on the synchronous hypothesis and use functional language as the specification language. System design is accompanied by transformational design exploration. The system model abstracts from implementation issues such as

communication mechanisms and its formal nature supports formal methods and verification. The use of skeletons also makes it possible to interpret the system model as a hardware or software structure which will lead to an efficient implementation.

As the next step of our project, we will apply the methodology to a real industrial example. And we will focus on the connection of formal verification methods to our design methodology.

REFERENCES:

[BB91] 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.

[BCSS98] Per Bjesse, Koen Claessen, Mary Sheeran and Satnam Singh, Lava: Hardware Design in Haskell. In ACM Int. Conf. on Functional Programming, 1998.

[BD75] R. M. Burstall and John Darlington. Some Transformations for Developing Recursive Programs. In Proceedings of International Conference on Reliable Software (Los Angels) IEEE, New York, pp. 465-472, 1975.

[BS91] F. Boussinot and R. de Simone, The ESTEREL Language, Proceedings of the IEEE, Vol. 79, No. 9, pp. 1293-1304, September 1991.

[DON95] J. O'Donnell, From Transistors to Computer Architecture: Teaching Functional Circuit Specification in Hydra, In Symposium on Functional Programming Languages in Education, July 1995.

[ELLS97] 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.

[GGBM91] 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.

[HAR87] D. Harel, “STATECHARTS: A Visual Approach to Complex Systems”, Science of Computer Programming, 8-3, pp. 231-275, 1987.

[HCRP91] 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.

[HPF99] P. Hudak, J. Peterson and J. H. Fasel, A Gentle Introduction to Haskell. At http://www.haskell.org/tutorial.

[JKS98] A. Jantsch, S. Kumar, I. Sander, B. Svantesson, J. Öberg, and A. Hemani, P. Ellervee, M. O’Nils, "Comparison of Six Languages for System Level Descriptions of Telecom Systems", Proceedings of the Forum on Design Languages, vol. 2, pp. 139-148, 1998.

[JS90] G. Jones and M. Sheeran, “Circuit Design in Ruby”, in Formal Methods for VLSI Design, North Holland, edited by J. Staunstrup, 1990.

[LL95] 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.

[LS97] 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.

[MCL98] J. Mattews, B. Cook and J. Launchbury, Microprocessor Specification in Hawk, In Proceedings of ICCL'98, May 1998, Chicago.

[MOL97] B. Möller, Deductive Hardware Design: A Functional Approach, Report 1997-09, Institute of Computer Science at the University of Augsburg, 1997.

[REE95] H. J. Reekie, Realtime Signal Processing, PhD Thesis, University of Technology at Sydney, Australia, 1995.

[SJ99] I. Sander and A. Jantsch, System Synthesis Based on a Formal Computational Model and Skeletons, Proceedings of IEEE Workshop on VLSI’99 (WVLSI’99), pp. 32-39, April 8-9, Orlando, Florida, USA, 1999.

[WJ99] W. Wu and A. Jantsch, A Survey of Design Transformation Techniques, Internal report, TRITA-ESD-99-1, ISSN 1104-8697, ISRN KTH/ESD/FOU--99/1--SE, Royal Institute of Technology, Department of Electronics, ESDlab, Stockholm, Sweden, 1999.