Transformational System Design Based on a Formal Computational Model and Skeletons

Wenbiao Wu, Ingo Sander, Axel Jantsch
Royal Institute of Technology, Stockholm, Sweden
{wenbiao, ingo, axel}@ele.kth.se

Abstract

The Formal System Design methodology ForSyDe [1,2,3] is extended by a systematic refinement methodology based on transformations, which gradually transforms a high-level, function oriented system description into a synthesizable model. We group transformations according to three different criteria: (1) whether they preserve semantics or they constitute a design decision; (2) whether they are simple rewriting rules or complex patterns; (3) whether they transform combinatorial functions, skeletons or data types. We illustrate the use of transformations with three examples taken from an ATM based network terminal system.

1. Introduction

System level functional validation has been identified as one of the most severe obstacles to increased design productivity and quality. It is most likely that this will not fundamentally change unless a rich portfolio of validation techniques becomes an integral part of the system specification and design process. In addition to simulation, which is the main validation vehicle today, formal analysis and verification techniques must be available to the designer. Unfortunately, the full potential of formal techniques can only be exploited when they become an equal partner to simulation and design activities, which means that modeling and design techniques have to adapt and change significantly. Most current and proposed design languages such as VHDL, C++, Superlog [4], SystemC [5], etc., are foremost simulation languages. Considerations about formal analysis and verification are only taken up as an afterthought. This will always be insufficient because the semantics of these languages fundamentally limits the kind of formal analysis that can be performed.

We have proposed ForSyDe (Formal System Design) [1,2,3], a system modeling method based on a formal functional model and on skeletons with the objective to eventually develop both powerful formal analysis techniques and a full fledged refinement and synthesis process. In [1] we have argued that a carefully designed modeling method is the solid basis for efficient verification and refinement. Now we take the next step and propose a refinement technique, which takes formal verification into account up front. It is based on transformations, which allows to split the system validation into numerous small steps, one separate verification step for each applied transformation. Thus, the seemingly impossible task of verifying the functionality of an entire complex system becomes feasible because it is reduced to many smaller and simpler steps.

The main contribution of this paper is the extension of ForSyDe methodology with a transformational refinement process of the system model. We outline different types of transformations, which together are powerful enough to allow refinement of a system level functional specification into a synthesizable description. The technique is illustrated by applying selected transformations on a system model of an ATM switch.

We review the related work in this area in section 2, introduce our system model in section 3, discuss transformational design in section 4,

illustrate transformation examples in section 5 and conclude our paper in section 6.

2. Related work

Transformational system design supports the idea of designing at a higher level of abstraction which implies describing the behavior of the required system in high level languages and then transforming this into a description at a lower 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. A great deal of the pioneering work in transformation systems was undertaken by Burstall and Darlington [6]. Their system transformed applicative recursive programs to imperative ones and their ideas have heavily influenced today's transformation systems.

Haskell [7] 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 the Hawk project [8], Hawk is a library of Haskell functions that are appropriate building blocks (or structural units) for describing the micro-architecture level of a microprocessor. In their work, Hawk has been used to specify and simulate the integer part of a pipelined DLX microprocessor. Lava [9] is a collection of Haskell modules. It assists circuit designers in specifying, designing, verifying and implementing hardware. O'Donnell [10] 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. Möller [11] 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 behavior specification. Gofer/Haskell (Gofer is an interpreter for 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 designs which differ significantly from our approach. Reekie [12] used Haskell to model digital signal processing applications. He modeled 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 a more effective representation. However, in Reekie's work, the target applications are data flow networks and the final representation is not synthesized. In our approach, we targeted at both data flow and control intensive applications and the final system is synthesized to a VHDL and C model.

There are also some other languages to support hardware design. The Ruby language, created by Jones and Sheeran [13], 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 [14] 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.

3. System model

Today, system design often starts with a description of concurrent processes communicating with each other by means of complicated mechanisms (shared variable, remote procedure call or asynchronous message passing). We have argued [15] that such a system model does not serve as a good starting point for system design, because many design decisions are already taken. In particular, it is difficult to change the process structure of the system model, i.e. to move functions from one process to another, since this will require the redesign of the communication structure between the processes.

We have addressed this problem in our design methodology by adopting a purely functional specification of the system based on data dependencies and written in the functional language Haskell, for which a formal semantics

exists.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. For a formal definition of the computational model, we use the denotational framework of Lee and Sangiovanni-Vincentelli [16]. It defines a *signal* as a set of *events*, where an event has a tag and a value. A signal is shown in Figure 1. Here events are

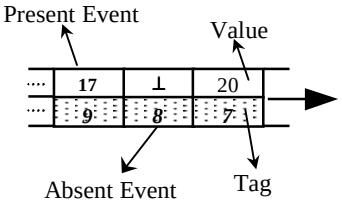

Figure 1. A signal is a set of events

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.

The system is modeled by means of concurrent processes which are supported by the use of *skeletons*. Skeletons are higher-order functions whose inputs and outputs are signals. *Elementary functions* are the basic components inside the skeleton. In terms of a clocked system, elementary functions can model the behavior of system components in one clock cycle. On the other hand, each skeleton has a representation in the design library and can be directly mapped into a hardware or software component based on the information from a design library.

Due to its formal nature of the system model, which can be regarded as consisting of mathematical entities, it can be further analyzed by using mathematical methods, which in turn can also be mechanized. For example, the 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.

4. Transformational design

4.1. Introduction

With transformational design, we mean starting the system design with a formal specification model and then applying transformation techniques on this model to get a synthesizable system model, i.e. the specification can be transformed through a series of steps to a final synthesizable system model which meets the system requirement and constraints imposed by designers (Figure 2). These steps can be either semantics preserving or non-semantics preserving. The later introduces design decisions imposed by the designer. The transformation result of each step is sufficiently close to the previous expression that the effort of verifying the transformation is not excessive. The benefits of this approach are:

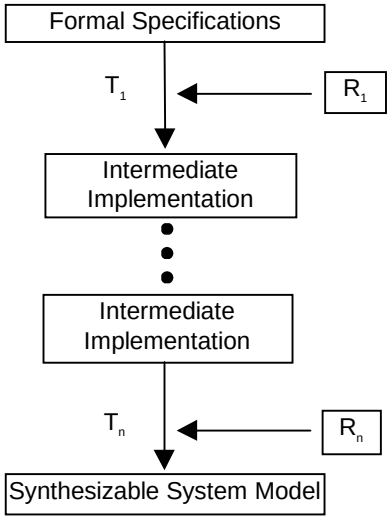

Figure 2. Transformational system design

- Each step in the system refinement is closer to the final implementation.
- The final implementation is either a true implementation of the initial specification which is guaranteed by the use of semantics preserving transformation rules in the library, or can be proofed correct for certain assumptions w.r.t non-semantics preserving transformations which are introduced by design decisions.
- The design process and design decisions are documented, thus the whole process is repeatable.
- A transformation approach made up of a sequence of small steps is very effective and much more efficient than theorem proving which is usually very difficult because the gap between implementation and

specification is much larger than the gaps between each 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).

4.2. Refinement of the system model by transformation

We distinguish two levels of transformations:

- ***Transformation rules*** are primitive transformations as developed over the last three decades [6,17]. They include
  - Definition: Introduce new functions to the system model;
  - Unfolding: Replace a function call with its definition;
  - Folding: the opposite of unfolding;
  - Abstraction: Abstract common expressions used in system models into one separate function to be reused;
  - Algebraic rules;
- ***Transformation patterns*** describe more complex transformations which in many cases can be application and implementation related. For instance, if a finite state machine model contains a large and regular structured state, a transformation pattern can transform this into a simpler finite state machine and a memory skeleton. In the synthesis step such a model can later be implemented as a memory with memory controller, which is far more suitable than an FSM-implementation with many registers. Essentially, transformation patterns capture intelligent design techniques.

Orthogonal to this we classify transformations in the following way:

- ***Semantic preserving transformations*** do not change the functionality of the system, but they may have an impact on non-functional properties such as performance and cost. Semantic preserving transformations have the beautiful property, that they can be freely applied without changing the functionality. This greatly alleviates the validation and allows for fast design space exploration. Transformation rules are always semantic preserving, but transformation patterns may or may not be.
- ***Decision transformations*** inject a design decision into the refinement process. During the refinement process from a high abstraction level down to the implementation a number of decisions must be taken which also change the functionality. For instance, the high-level functional model will in many cases use infinite buffers (FIFO) for communication between entities. However, during refinement the infinite buffers must be reduced to finite buffers which invariably changes the behavior, because the finite buffers may overflow for certain input sequences. Decision transformations may be proved by adding additional assumptions. E.g. if we assume that the input sequences are constrained in such a way, that the finite buffers will never overflow, we can proof that the two descriptions behave identically. This has the additional benefit of making all these assumptions explicit and part of the system specification.

In addition, we can group the transformations in the following way: skeleton-based transformations on skeletons, local transformations on elementary functions which are usually the essential part of a skeleton, and data-type transformations for data types. This is also orthogonal to other classifications.

- ***Skeleton-based transformations*** are done on a higher level than local transformations on elementary functions. They are usually employed on the architecture model and will depend heavily on the design library which contains implementations for skeletons and library elements. In our design methodology, this kind of transformation will include merge and split skeletons, move functions between skeletons to minimize the communications between skeletons etc. and share elementary functions between skeletons.
- ***Local transformations*** are employed on the elementary functions inside the skeleton. They can be introduced by skeleton-based transformations, for example merging and

splitting skeletons usually give rise to the reconstruction of internal elementary functions which permits further optimization.

- ***Data type transformations*** are usually employed to transform one list to multi-lists or vice versa, split a signal into several signals to be processed by several processes simultaneously to acquire high efficiency, transform data structures (e.g. trees and lists) to meet the specialized system implementation and transform unbounded data types to bounded data types.

During design exploration, these transformations are employed on the system model to optimize the system and thereby to meet the specified constraints. The system model is successively refined by means of applying small transformational steps to the model in order to receive a synthesizable system model, which meets the specified constraints. The refinement process is also supported by estimation values for possible transformations.

4.3. Synthesis of the synthesizable system model

After the refinement process the HW parts of the synthesizable system model are synthesized into VHDL, while the SW parts are synthesized into C. Here we use the HW and SW interpretations of the skeletons. The VHDL model is then further processed by logic synthesis tools while the C-code is compiled for a specific processor [2,3].

5. Experiments in the ATM case study

In this section we illustrate the transformational design methodology by several examples from our case study, an ATM switch. A more detailed description of the design of this ATM switch can be found in [1]. The ATM switch has operation and maintenance functionality. This case study has the following aspects:

- The original specification model is written completely in Haskell with skeletons used.
- The aforementioned transformation rules and patterns are employed manually during the design explorations.
- The design exploration utilizes the results from library estimation as one of the exploration criteria.
- The final system is a functional model which can be easily synthesized into a VHDL and C model.

Example I

First we introduce the skeletons used in these examples:
mapT is the skeleton that applies a function *f* on to the values of all events in a signal. It is based on the function *map* which maps a function on a list. It is defined like this in the functional specification:

```
mapT :: (a -> b) -> TimedSignal a -> TimedSignal b
mapT f (Sig xs) = Sig (map f xs)
```

Figure 3 shows skeleton *mapT* with elementary function *inc* which increments the value in each event.

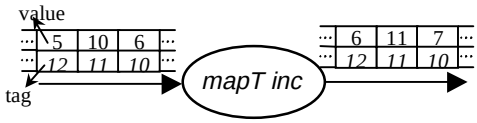

Figure 3. Skeleton mapT

filterT is the skeleton that filters out the events in a signal which doesn't satisfy property *p*. Here is its definition:

```
filterT :: (a -> Bool) -> TimedSignal a -> TimedSignal a
filterT p = mapT (check p)
where check p x = if p x then x else Absent
```

Figure 4 shows the skeleton *filterT* with elementary function *even* which filters out events with odd values.

Figure 4. Skeleton filterT

In the ATM system, we use the following functional description which checks the status of a virtual path and generates OAM cells if a certain condition is met. This is also shown in the first part of Figure 5.

```
sendDownstream = mapT action.filterT isSendCondition
where
isSendCondition (Present (vpi, (VPI_OK, _))) = False
isSendCondition (Present (vpi, (VPI_AIS, 1000))) = True
isSendCondition (Present (vpi, (_, _))) = False

action (Present (vpi, (VPI_AIS, time))) =
        Present (F4_OAM (VPI vpi) VPI_RDI)
action _ = Absent
```

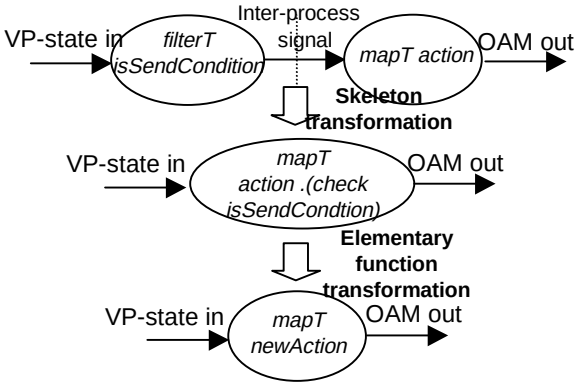

Figure 5. sendDownstream

sendDownStream first filters out the events in incoming signal according to *isSendCondtion*, then it takes *action* on the outgoing signal from the previous process.

This specification can be transformed to:

newSendDownstream
= mapT action . filterT isSendCondition (0)
= mapT action . mapT (check isSendCondition) (1)
= mapT (action . (check isSendCondition)) (2)
= mapT newAction (3)

newAction Present (vpi, (VPI_AIS, 1000))
= Present (F4_OAM (VPI vpi) VPI_RDI)
newAction _ = Absent

Step (0) is the definition of *newSendDownStream*. The transformation step from (1) to (2) is a skeleton-based transformation which is based on the rule *mapT f . mapT g = mapT (f.g)*. However, the transformation step from (2) to (3) is an elementary-based transformation, which is aimed to optimize the elementary function inside the skeleton. This example also shows that transformation allows the optimization of internal functions. The advantage of this transformation is apparent if we compare them in Figure 5. The original structure will usually be implemented in two processes, either HW or SW. There is intensive communication between these two processes. However, in the new specification, only one process is needed in the final implementation.

Example II

To get the system work reliably under some critical environments, we need some fault tolerance in the system design. In the ATM switch we can duplicate the switch to achieve this goal. This is shown in Figure 6, where *atmSwitch* is the original switch module and *newSwitch* is the resulting system which is derived from *atmswitch*. The definition of *newSwitch* is as following:

newSwitch atmin = (atmOut1, atmOut2)
where
atmOut1 = zipwith3T selectF atmUpOut1
atmMiddleOut1 atmDownOut1
atmOut2 = zipwith3T selectF atmUpOut2
atmMiddleOut2 atmDownOut2
(atmUpOut1, atmUpOut2) = atmSwitch atmUpin
(atmMiddleOut1, atmMiddleOut2) = atmSwitch atmMiddlein
(atmDownOut1, atmDownOut2) = atmSwitch atmDownin
(atmUpIn, atmMiddleIn,atmDownIn) = fan3T id id id atmin

fan3T f g h x = (f x, g x, h x)
zipWith3T :: (a -> b -> c) -> TimedSignal a ->
TimedSignal b -> TimedSignal c
zipWith3T f (Sig (x:xs)) (Sig (y:ys)) (Sig (z:zs))
= Sig (f x y z: zipWith3 f xs ys zs)

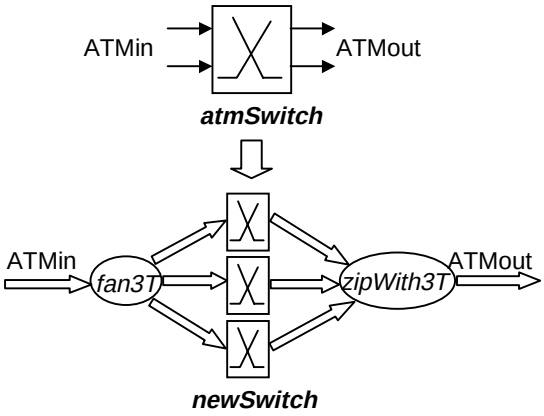

Figure 6. Transform one module to three

Function *fan3T* fans one ATM stream into three which are then fed into three ATM switches in *newSwitch*. The results of all switches are later sent to *(zipWith3T selectF)* which selects one value as output. Function *selectF* can be defined in different ways. A common implementation can be majority vote, which compares the inputs and selects the value, which is equal in two or three of the inputs. The correctness of this transformation can be easily proved. As a result, the object system is more reliable than the original one. Similar transformations to make the design more parallel can be beneficial also for performance reasons.

Example III

The previous examples are semantics preserving transformation. There are also non-semantics preserving transformations. We will illustrate

this with the FIFO design in this ATM case study.

In the ATM example, an unconstrained FIFO is defined as:

```
unconstrainedFifoT :: TimedSignal [a] -> TimedSignal 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.

In the final implementation, the FIFO will only have a fixed buffer size and can only consume a fixed number of items during one event cycle. To solve this, 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 ->
                    TimedSignal [a]  -> TimedSignal a
constrainedFifoTemplate b i
  = mooreS (constrainedFifoState b i) fifoOutput []
```

We can then build instances of constrained FIFOs by specifying the parameters *b* and *i*. For example: (Figure 7)

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

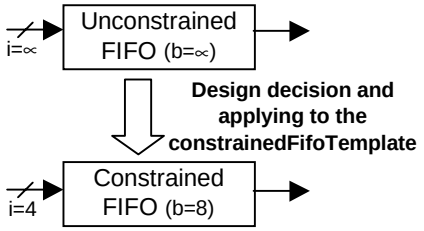

Figure 7. Decision transformations on FIFO

The result of this transformation is much closer to the final synthesizable model and this transformation can be used in several parts of the case study.

We have illustrated some transformation techniques with the ATM case study. The purpose of this case study is to validate our transformational system design methodology. Furthermore, through this case study, we have obtained a better understanding of the required transformations and how they can be mechanized to solve real industry problems. However, this is only a small subset of the complete transformation library which will eventually include a rich set of transformation patterns. Besides the transformations in this case study, it also includes memory-based transformations for FSM, communication transformations and other transformations on skeletons. This set of transformation patterns combined with the various transformation rules should be sufficient to derive a synthesizable model from the functional specification.

6. Conclusion and future work

We introduced transformational refinement to our design methodology for HW/SW co-design which starts with a formally defined functional system model and abstracts from the implementation details. The system model is stepwise refined by formally defined transformations leading to a synthesizable system model. Due to the use of skeletons the synthesizable model can be synthesized into an efficient implementation incorporating HW and SW parts.

As the next step of our current work, we will focus on how to mechanize some of the transformations we have proposed and how to verify these transformations.

7. References

[1] I. Sander and A. Jantsch, "Formal Design Based on the Synchronous Approach", *Proceedings of the Twelfth International IEEE Conference on VLSI Design*, pp. 318-324, Goa, India, January 7-9, 1999.

[2] I. Sander and A. Jantsch, "System Synthesis Utilizing a Layered Functional Model", *Proceedings of the 7th International Workshop on Hardware/Software Codesign*, pp. 136-141, Rome, Italy, May 3-5, 1999.

[3] 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, Orlando, Florida, USA, April 8-9, 1999.

[4] P. L. Flake and Simon J. Davidmann, "Superlog, a unified design language for system-on-chip", *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 583 -586, 2000.

[5] S. Y. Liao, "Towards a New Standard for System-Level Design", *Proceedings of the Eighth International Workshop on Hardware/Software Codesign (CODES)*, pp. 2-6, May 2000.

[6] R. M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM, 24(1)*, pp. 44-67, January 1977.

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

[8] J. Mattews, B. Cook and J. Launchbury, Microprocessor Specification in Hawk, In *Proceedings of IEEE International Conference on Computer Languages*, Chicago, May 14-16, 1998.

[9] P. Bjesse, K. Claessen, M. Sheeran and S. Singh, Lava: Hardware Design in Haskell. In *ACM International Conference on Functional Programming*, Baltimore, Maryland, 1998.

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

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

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

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

[14] Y. Li and M. Leeser, "HML: A Novel Hardware Description Language and Its Translation to VHDL", *IEEE Transactions on VLSI Systems, Vol 8(1), pp. 1-8, Feb. 2000*.

[15] A. Jantsch and I. Sander, "On the Roles of Functions and Objects in System Specification", in *Proceedings of the International Workshop on Hardware/Software Codesign*, 2000.

[16] E. A. Lee and A. Sangiovanni-Vincentelli, "A Framework for Comparing Models of Computation," IEEE Transactions on CAD, Vol. 17, No. 12, December 1998.

[17] A. Pettorossi and M. Proietti, "Rules and Strategies for Transforming Functional and Logic Programs", *ACM* Computing Surveys, Vol. 28, No. 2, pp. 360-414, June 1996.