Verification of Design Decisions in ForSyDe*

Tarvo Raudvere, Ingo Sander, Ashish Kumar Singh, Axel Jantsch
Royal Institute of Technology
Stockholm, Sweden
tarvo,ingo,ashish,axel@imit.kth.se

ABSTRACT

The ForSyDe methodology has been developed for system level design. Starting with a formal specification model that captures the functionality of the system at a high abstraction level, it provides formal design transformation methods for a transparent refinement process of the specification model into an implementation model that is optimized for synthesis. A transformation may be semantic preserving or a design decision. The latter modifies the semantics of the system level description and changes the meaning of the model. The main contribution of this paper is the incorporation of model checking to verify that refined system blocks satisfy the design specification. We illustrate the translation of the ForSyDe code to the SMV language and the verification of local design decisions with a case study of a ForSyDe equalizer model.

Categories and Subject Descriptors

B.6.3 [**Logic Design**]: Design Aids—*Verification*; C [**Computer Systems Organization**]: Systems specification methodology

General Terms

Design, Verification

Keywords

System Design, Design Refinement, Verification

1. INTRODUCTION

Future applications and architectures with extreme complexity can be implemented on a single chip since the capacity of integrated circuits is continually growing. In order to develop these applications we believe that a system design methodology has to start at a high abstraction level, where (1) functions should be separated from architecture and (2) computation should be separated from communication. Also the methodology has to make it possible to incorporate formal methods to verify systems at different levels of abstraction, since simulation alone is not sufficient.

These arguments have been a basis for the development of the ForSyDe (Formal System Design) methodology which addresses the transformational design of embedded systems. A series of transformations is applied to an initial abstract formal and functional specification model to refine the model into a final implementation model. In [14, 15] we have introduced transformations for ForSyDe.

The objective of this paper is to verify that blocks of the system model satisfy system specific properties after design decision transformations. Simulation techniques can not guarantee the correct behavior for all input stimuli. For complex designs producing a mathematical proof in order to prove the effect of design decisions on the system behavior requires high level of expertise and may be impossible in practice. An alternative approach to validate a system is to apply formal verification techniques to prove that the system satisfies the specification. We intend to use model checking in cooperation with abstraction to validate that after each design decision the refined system block locally satisfies system specific properties.

2. RELATED WORK

According to the tagged signal model developed by Lee and Sangiovanni-Vincentelli [10] the ForSyDe system model can be classified as a synchronous computational model. It is based on the synchrony hypothesis, that also forms the base for the synchronous languages. According to Benveniste and Berry ”the synchronous approach is based on a relatively small variety of concepts and methods based on deep, elegant, but simple mathematical principles” [3]. The synchronous assumption implies a total order of events and leads to a clean separation between computation and communication and gives a solid base for formal methods.

Two of the well known formal techniques for verification of the design correctness are theorem proving [7, 8] and model checking [6]. In the former case the correctness of a system is determined through mathematical proof composed by the designer which demands good knowledge and ingenuity. The latter method applies state exploration techniques to decide over the correctness of a system. The state space of a system must be finite for this approach. In order to have a finite state space abstraction techniques can be applied.

A good overview about program transformation in general is given in [12] and for transformation of functional and logical programs in [13]. One of the most well known transformation systems is the CIP (computer-aided, intuition-guided programming) project [2]. Inside CIP, program development is viewed as an evolutionary

*This research was supported by the Swedish Foundation for Strategic Research within the INTELECT program.

CODES+ISSS'03, October 1–3, 2003, Newport Beach, California, USA.

process that usually starts with a formal problem specification and ends with an executable program for the intended target machine. The individual transformations are done by semantic preserving transformation rules, which guarantees that the final version of the program still satisfies the initial specification.

Transformational approaches have been used mostly for development of software programs [13] but software transformational approaches do not deal with synchronous sub-domains and resource sharing which are required in ForSyDe. There are also a number of other transformational approaches targeting hardware design, e.g. [16], but none of them explicitly develops the concept of design decisions or addresses the refinement of a synchronous model into multiple synchronous sub-domains.

Lava [5] is a hardware description language based on Haskell. Theorem proving is used for the verification of Lava programs. An abstract circuit should be constructed containing both a system and a property for a theorem proving method. In our approach we are performing a direct mapping of ForSyDe into a state machine description in SMV from which properties can be verified by automatic tools. Hence, little expertise is needed for constructing a proof and this makes it easier for transformations verification. Esterel [4] is a synchronous language for programming reactive systems. Both an Esterel and a ForSyDe model can be translated into a form of a finite state machine as an input for automata verification systems to perform behavior analysis and proofs. However the ForSyDe model may include more complex data-flow descriptions compared with Esterel which is mostly control oriented. Therefore the task of verification is more involved. Lustre [9] is a synchronous data-flow language for programming critical real-time systems. The advantage of the ForSyDe methodology is that a system model may have control and data-flow behaviors at the same time. The Lustre program includes a system description as a set of input/output relations, assumption about the behavior of the environment as a set of assertions and finally a set of properties which will be checked by a verification tool. The verification will be done similar to the symbolic model checking by using binary decision diagrams for state space exploration. In the ForSyDe methodology we are using a step-wise transformational approach where we need to verify each of these transformations against a design specification.

3. THE FORSYDE METHODOLOGY

3.1 The Design Process

The ForSyDe design process starts with the development of a formal, abstract, functional *specification model* that can be executed using the functional language Haskell [17]. This model is then refined inside the *functional domain* by a stepwise application of well defined design transformations into an *implementation model*. As the implementation model is a refined version of the specification model, the same validation and verification methods can be applied to both models. In the partitioning phase, the implementation model is partitioned into hardware and software blocks, which are mapped on architectural components. Only now, in the code generation phase, we leave the functional domain and enter the *implementation domain* to produce VHDL or C/C++ for the hardware and software parts as discussed in [11].

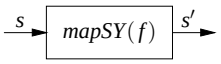
3.2 The Specification Model

The specification model is based on a synchronous computational model and uses ideal data types such as real numbers and infinite buffers. It abstracts from implementation details, such as low-level communication mechanisms and enables the designer to focus on the functional behavior of the system rather than structure and architecture. The specification model leaves a wide design space for further design exploration and design refinement, which is supported by our transformational refinement techniques (Section 4).

We describe our computational model in the sense of signals and processes where processes are executed concurrently and synchronous communication between them is modeled by signals. A signal is defined as a set of events where each event e_i has a value and a tag i. As we use the perfect synchrony hypothesis [3], all signals have the same set of tags. In order to model the absence of a value at a certain tag, a data type D can be extended into a data type $D_\perp$ by adding the special value $\perp$. Absent values are used to establish a total order of events when dealing with signals with different or aperiodic event rates. A system can be constructed as a network of processes and modeled as a set of equations.

We implement the synchronous computational model with the concept of *process constructors*. A process constructor is a higher-order function that takes *combinational functions* and *values* as input and produces a process as output. The ForSyDe methodology obliges the designer to use process constructors for the modeling of processes. This leads to a well defined specification model with a clean separation between *synchronization* (process constructors) and *computation* (combinational function). In addition each process constructor has a structural *hardware and software semantics* which is used to translate the implementation model into a hardware/software implementation [11].

The process constructor *mapSY* (Figure 1) takes a combinational function f and constructs a process with one input and output signal, where f is applied on all values of the input signal.



$$\begin{array}{lcll} mapSY(f) & = & P & \\ \text{where} & & P(s) & = & s' \\ & & f(e_i) & = & e'_i \end{array}$$

Figure 1: The Combinational Process Constructor *mapSY*

Figure 2 illustrates a $mooreSY_n$ process which models a n-input Moore state machine. The process constructor $mooreSY_n$ takes two combinational functions f and g as next state function and output function and m_0 as initial state. The process constructor $zipWithSY_n$ is similar to *mapSY* and applies a n-variable combinational function f to all events of the input signals. The process constructor $delaySY_1$ models one-cycle delay and emits m_0 as the first value of the output signal.

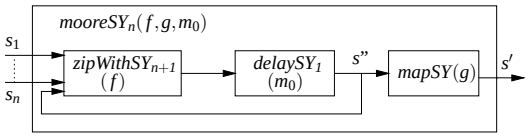


Figure 2: The $mooreSY_n$ **Process Constructor**

3.3 Implementation Model

The implementation model is the result of the refinement process (Section 4). In contrast to the specification model which is a network of concurrent synchronous processes it may also include synchronous sub-domains with a different signal rate. Synchronous sub-domains violate the synchronous assumption since not all signals share the same set of tags. Thus they are not allowed in the

specification model, but are introduced by well-defined transformations during the refinement process. Inside a synchronous subdomain the synchronous assumption is still valid and the same formal techniques can be used as for the specification model.

4. REFINEMENT OF THE SYSTEM MODEL

The initial specification model is stepwise refined through the use of well defined design transformations into a final implementation model.

There are two classes of transformation techniques:

Semantic Preserving Transformations do not change the meaning and the behavior of the model, rather these are mainly used to optimize the model for synthesis.

Design Decisions change the meaning of a model. A typical design decision is the refinement of an infinite buffer into a fixed-size buffer with *n* elements. While such a design decision clearly modifies the semantics, the transformed model may still behave in the same way as the original model. For instance, if it is possible to verify that a certain buffer will never contain more than *n* elements, the ideal buffer can be replaced by a finite one of size *n*.

The designer applies transformations to a system model by choosing transformation rules from the transformation library. The transformation rules are characterized by a name, the required format and constraints of the original process network, the format of the transformed process network and the implication for the design, i.e. the relation between original and transformed process network is expressed by the characteristic function. For a more elaborate discussion see [15].

Since the system properties are preserved by semantic preserving transformations the verification is needed only for design decisions. Design decisions can be put into various categories, for instance *a clock domain refinement* which introduces multiple clock domains in the original synchronous model or *communication refinement* which can be used to partition the system into hardware and software parts and the communication interface between them [14]. We will take up an example of communication refinement of a synchronous channel into an asynchronous protocol which is suitable to model hardware-software interfaces (Figure 3). For this transformation the channel is required to be type of $V_\perp$. First an identity process is introduced, and then this process is refined into the handshaking protocol by introducing the processes *FIFO*, *Send* and *Receive*. When *Send* is idle it tries to read data from *FIFO*. If the reading was successful then *Send* emits the message DataReady to *Receive* and after receiving the message Ready, it sends the data. After the data is received the *Receive* sends a message Ack to *Send*.

The timing behavior of the refined interface is different from the original interface since the handshake protocol implies a delay of several cycles for each event, as *Send* and *Receive* are synchronous processes. Also the sub-system in the *Receive* side of the channel will not process exactly the same combination of values in each event cycle as in the specification model. These consequences have to be taken into account, when interfaces are refined. In order to validate that the refined system block satisfies the design specification for given assumptions about the input characteristics of the channel and size of the FIFO buffer, we need to incorporate verification.

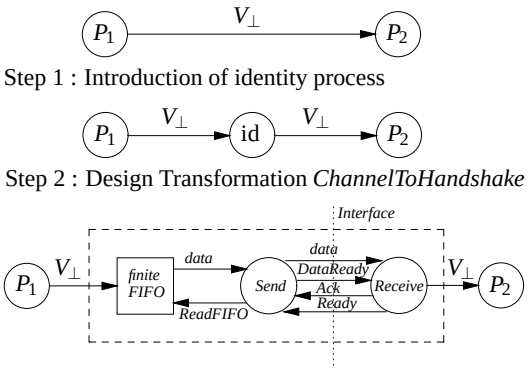


Figure 3: Refinement into a Handshake Protocol

5. VERIFICATION IN FORSYDE

The design flow starting from the development of a specification model M_0 to an implementation model M_n through transformations T_i is shown in Figure 4. A transformation $T(M,R,PN) \rightarrow M[R(PN)/PN]$ refines the process network PN, which is a part of the entire process network inside the model M, according to the transformation rule R. The result of the transformation is an intermediate system model M' where in contrast to the model M the process network PN is replaced with $R(PN)$. In order to verify the correctness of system blocks after design decisions we incorporate model checking techniques.

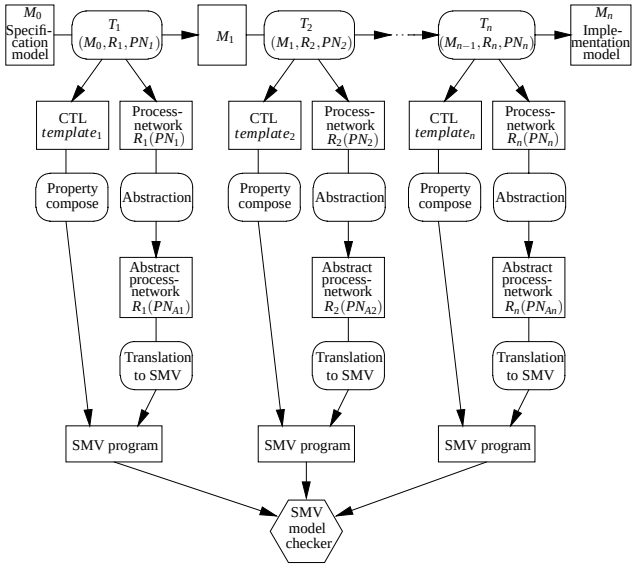


Figure 4: Verification of Design Transformations

For every design transformation we have a set of predefined specification templates which helps the designer to construct a CTL specification. For example a decision may be taken based on assumptions about system environment. We can assume some certain data rate in the input of a FIFO buffer and according to the rate to set the size of the buffer. To verify that the size of the FIFO buffer will not be exceeded we offer a configurable input pattern generator and a template for a CTL specification.

In contrast to theorem proving techniques model checking can be applied only to systems with a finite state space. Since a ForSyDe specification model is allowed to describe a system with an infi-

nite state space the designer has to determine the finite state space through abstraction. The design flow continues with the translation of the refined part of the system model with the abstract state space into the input language of a model checking tool. We use the Cadence version of SMV (Symbolic Model Verifier) since the tool has been used successfully for model checking and there is a straightforward mapping from the ForSyDe language to the SMV language as described in Section 5.2.

The translation from ForSyDe into SMV can be fully automated and we give the guideline of properties which should to be verified. Based on the latter we estimate that the verification flow without abstraction takes much less time than the designer needs for selecting proper design decisions.

5.1 The SMV Tool

SMV [1] is a tool for the formal verification of finite state systems. The tool is based on a technique called symbolic model checking and can be used to check whether a system satisfies a specification given in the temporal logic CTL. CTL makes it possible to define various properties composing liveness, fairness, safety and deadlock freedom which can describe very complex relations of signals in terms of timing and values. If the system does not satisfy the given specification then the tool gives a counter example. The counter example is a trace from the system initial state to a state where the verified property does not hold. Systems can be expressed in the SMV language which offers modular hierarchical descriptions and reusable components of a system.

5.2 Translation from ForSyDe to SMV

The system mapping from ForSyDe to SMV entails datatype definitions, translation of function and process definitions, and specification of connections between processes. Each of these steps is described in further detail in the following subsections.

The translation is illustrated with a receiver which is a part of the refined equalizer model (Section 6). The equalizer specific ForSyDe code of a part of the receiver is given in Figure 5. The state of the receiver is modeled as a tuple of two elements, the first element tells if the receiver is waiting for data or has received the data and the second element is used to keep the values of events received from the sender. The first component of the state may have values *WaitDataReady*, *WaitData* or *OutputData* which are defined through the scalar datatype *RecState*. The receiver is constructed with the process constructor *moore2SY* which has the state function *recState*, the output function *recOutput* and the initial state value $(WaitDataReady, 0)$ as arguments. The state transition function and the output function are defined using pattern matching. The last three lines in Figure 5 defines the output function *recOutput*. If the first element of the current state is *WaitDataReady* then the first output has value *Prst Ack* and the second has value *Abst*.

```
data RecState = WaitDataReady | WaitData | OutputData

receiver = moore2SY recState recOutput (WaitDataReady,0)

recOutput (WaitDataReady,_) = (Prst Ack      ,Abst)
recOutput (WaitData     ,_) = (Prst Ready    ,Abst)
recOutput (OutputData   ,v) = (Abst          ,Prst v)
```

Figure 5: Receiver in ForSyDe

5.3 Datatypes

The datatypes offered by the SMV language are Boolean, Scalar, Struct and Array. This is a proper base to construct datatype definitions for all the ForSyDe datatypes with some minor restriction.

- Integers are defined as Scalars which demands the user to specify the range of all reachable values. The SMV tool treats them as Integers and provides arithmetic and logic operation on them. In the following we give an example of datatype definition of *Int_0_7* which covers Integers values from 0 to 7:

  ```
  typedef Int_0_7 0..7;
  ```

- A definition of a Scalar datatype contains the name of the new datatype and a set of all possible values. An example of the RecState type definition is the following:

  ```
  typedef RecState = {WaitDataReady ,
              WaitData , OutputData};
  ```

- Constructor based datatypes will be translated to structural datatypes. For example an absent extended Integer with possible values *Abst* and *Prst Int_0_7* has the following definition:

  ```
  typedef AbstExt {Prst,Abst};
  typedef Abst_Int_0_7 struct {
      Con : AbstExt ; Val : Int_0_7};
  ```

- A list is defined as a pair of an Array and an Integer value where the Integer value is employed to store the count of elements in the list. We assume that lists have a finite size defined by the user.

- Each element of a tuple will be defined as an independent variable with its own datatype.

5.4 Functions

Functions which are arguments for process constructors in ForSyDe will be expressed as modules in SMV. In ForSyDe a function has the following shape:

```
function_name::input1_type->...->inputN_type->output_type
function_name  condition1 = expression1
...
function_name  conditionM = expressionM
```

The first line of a function definition express input and output datatypes. Each of the following lines consists of one condition and one expression. If the condition is satisfied then the corresponding expression evaluates the function output. The SMV module generated according to the upper function has the following style:

```
MODULE function_name(input1,...,inputN){
   out : datatype;
   out := case {
              condition1 : expression1;
              ...
              conditionM : expressionM;}}
```

The line *out* : *datatype*; defines a new variable *out* and the next line assigns a value to the variable through a conditional case expression. To address the variable one has to write *function_name*.*out*.

5.5 Processes

Process constructors in the ForSyDe library are classified as constructors for predefined processes and constructors for user defined processes. For the former we have predefined SMV modules since their functionality is fixed for example memories. The latter process constructors are higher order functions which take a set of user defined functions as arguments. The definition of a process P which has behavior of a Moore state machine is $P = mooreSY_n(f, g, m0)$ where f is a next state function, g is an output function and *m0* is an initial state value of the state machine. The process has the following translation into SMV:

```
MODULE Pmoore(inp1,...,inpN){
   state       : datatype;
   output      : datatype;
   stateFunc   : f(state,inp1,...,inpN);
   outputFunc  : g(state);
   init(state) := m0;
   next(state) := stateFunc.out;
   output      := outputFunc.out;}
```

In the SMV language *init* denotes the initial value of a variable and *next* denotes its value in the next state.

5.6 Netlists

A system is modeled as a network of processes in ForSyDe where signals are used to connect processes with each other. The network is expressed as a netlist. In SMV we construct a similar netlist in the main module. For example the process network in Figure 6 is expressed in ForSyDe as the following:

```
system s1 = s5
  where
  (s2,s3) = P1 s1
  s4      = P3 s3
  s5      = P2 s2 s4
```

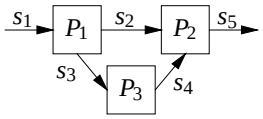


Figure 6: A Network of Processes

and the corresponding definition in SMV is the following:

```
s23 : P1(s1);
s4  : P3(s23.out2);
s5  : P2(s23.out1,s4.out);
```

In the SMV code the signals *s23.out1* and *s23.out2* correspond to the ForSyDe signals *s2* and *s3* respectively.

6. VERIFICATION OF DESIGN TRANSFORMATIONS

We illustrate refinement and verification in ForSyDe by means of the system model of an equalizer (Figure 7). The main task of the equalizer is to adjust the audio signal according to the *Button Control*.

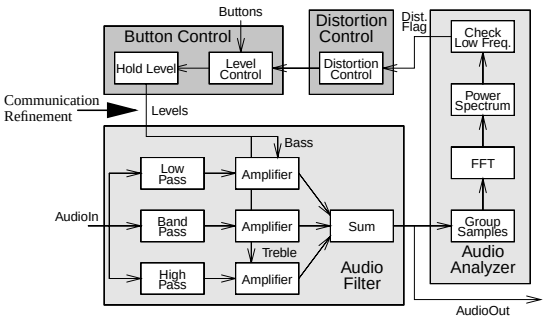


Figure 7: Subsystems of the Equalizer

The *Button Control* subsystem monitors the button inputs and the override signal from the subsystem *Distortion Control* and adjusts the current bass and treble levels. Since the aperiodic data rate of the *Button Control* and the *Distortion Control* subsystem is much lower than the data rate of the *Audio Filter* and *Audio Analyzer*, the *Button Control* and *Distribution Control* are implemented in software and the *Audio Filter* and *Audio Analyzer* in hardware. We use the design transformation *ChannelToHandshake* (Figure 3) [15] to refine the communication between the *Button Control* and the *Audio Filter* in Figure 7.

We translated the refined handshake protocol into SMV. In order to verify the correctness of the refined system block and to estimate the size of the FIFO buffer we checked the following properties:

Property 1 The implementation of the handshake protocol includes a finite size FIFO buffer. It is obvious that any higher data rate of input values than the buffer size was dimensioned for, will cause buffer overflow and the loss of data. We can verify that any data entering the channel when there was at least one empty slot in the FIFO buffer will be transmitted into the channel output. If this property holds we can say that there is no data lost other than caused by overflow. The specification of this property expressed in CTL is the following:

```
SPEC AG ((input_stream.Con=Prst &
        input_stream.Val = 0  &
        fifoOutput.st2 < SIZE-1) ->
        AF (recOutput.out2.Con = Prst
          & recOutput.out2.Val = 0));
```

The signal *input_stream* is the input of the channel and may have any value type of *AbstExt_Int_0_7*. The state variable *fifoOutput.st2* represents the current number of elements in the FIFO buffer. The constant SIZE is defined as maximum number of elements the FIFO buffer can store. *recOutput.out2* is the output signal of the channel. The given specification defines the following property: Always if the FIFO buffer has at least one empty slot and the channel input holds an event *Prst* 0 then the channel always in the future emits value *Prst* 0. In a similar way we can verify the property for any other value instead of 0.

Property 2 We specified the handshake protocol so that it takes seven clock cycles to transport a data from the channel input to the output if the sender process is in the initial state when the data enters into the channel. The CTL specification of this property is the following:

```
SPEC AG ((input_stream.Con = Prst &
          fifoOutput.st_2 = 0 &
          sender.st1 = ReadFifo) ->
         (AX AX AX AX AX AX AX
          recOutput.out2.Con = Prst));
```

The variable *sender.st1* represents the state of the sender process and initial value of the sender according to the system model is *ReadFifo*.

Property 3 Based on the last property which says that it takes seven clock events to transport data through the channel we expected that if an input stream is composed of sub-streams length of eight and containing at most one present value in every sub-stream then the buffer size stays finite. In order to verify this property we defined in SMV a non-deterministic FSM which generates these sub-streams and we checked the following property:

```
SPEC AG (fifoOutput.st2 < SIZE);
```

The SMV tool reported that the given specification is incorrect and gave a trace which lead to the state where the property was not satisfied. Later we increased the length of sub-streams from eight to nine and the proposed specification was

true. The first specification did not hold because it takes two event cycles for the receiver to ask for the next data after it has delivered the last data.

The designer can in a similar way estimate the required buffer size according to any other input stream by defining a corresponding FSM to generate input sub-streams.

The wrong presumption about the safe input data rate is a typical mistake which shows that in order to validate a system we need to incorporate formal techniques instead of using only the designer intuition or simulation techniques.

Property 4 Finally we checked that present values in the output preserves the same order they have in the input.

The CPU time of a Sun Ultra 5 (192 MB RAM) required to verify the properties and the number of BDD nodes created by the SMV tool are given in Table 1.

Property	CPU time (sec.)	BDD nodes
Property 1	2.54	61961
Property 2	0.28	0
Property 3	0.44	3739
Property 4	9.05	127040

Table 1: Verification Time and Number of BDD Nodes

We have shown that the design transformation *ChannelToHandshake* can be used for the communication refinement from a synchronous channel into an asynchronous protocol.

7. CONCLUSION

In the ForSyDe methodology the design flow starts from the development of a specification model to an implementation model through design transformations. The contribution of this paper is the integration of a model checker for verification of refined system blocks by introduction of mapping rules from ForSyDe to state machine descriptions in SMV. For the transformations presented in the transformation library we offer a set of specification templates that helps the designer to construct a CTL specification.

At present we have not incorporated a methodology to reduce the state space of SMV specifications and thus the abstraction has to be done by the user. Therefore we plan to incorporate state space abstraction techniques in order to elaborate a methodology which helps the user to create an abstract system model with finite and reduced number of states.

8. REFERENCES

[1] The SMV model checker. http://www-cad.eecs.berkeley.edu/~kenmcmil/smv/.

[2] Friedrich Ludwig Bauer, Bernhrad Möller, Helmut Partsch, and Peter Pepper. Formal program construction by transformations – computer-aided, intuition guided programming. *IEEE Transactions on Software Engineering*, 15(2), February 1989.

[3] Albert Benveniste and Gérard Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, September 1991.

[4] Gerard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[5] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in Haskell. In *International Conference on Functional Programming*, 1998.

[6] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 343 – 354. ACM Press, 1992.

[7] David Cyrluk, S. Rajan, Natarajan Shankar, and Mandayam K. Srivas. Effective theorem proving for hardware verification. In *Theorem Provers in Circuit Design (TPCD '94)*, volume 901 of *Lecture Notes in Computer Science*, pages 203–222, Bad Herrenalb, Germany, sep 1994. Springer-Verlag.

[8] Michael J.C. Gordon and Tom F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, 1993.

[9] Nicolas Halbwachs, Fabienne Lagnier, and Christophe Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. *Software Engineering*, 18(9):785–793, 1992.

[10] Edward A. Lee and Alberto Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, December 1998.

[11] Zhonghai Lu, Ingo Sander, and Axel Jantsch. A case study of hardware and software synthesis in ForSyDe. In *Proceedings of the 15th International Symposium on System Synthesis*, Kyoto, Japan, October 2002.

[12] Helmut A. Partsch. *Specification and Transformation of Programs*. Springer-Verlag, 1990.

[13] Alberto Pettorossi and Maurizio Proietti. Rules and strategies for transforming functional and logic programs. *ACM Computing Surveys*, 28(2):361–414, June 1996.

[14] Ingo Sander and Axel Jantsch. Transformation based communication and clock domain refinement for system design. In *39th Design Automation Conference (DAC 2002)*, New Orleans, USA, June 2002.

[15] Ingo Sander, Axel Jantsch, and Zhonghai Lu. Development and application of design transformations in ForSyDe. In *Design, Automation and Test in Europe Conference (DATE 2003)*, Munich, Germany, March 2003.

[16] Tiberiu Seceleanu. *Systematic Design of Synchronous Digital Circuits*. PhD thesis, University of Turku, Finland, 2001.

[17] Simon Thompson. *Haskell The Craft of Functional Programming Second Edition*. Addison-Wesley, 1999.