Model Checking and Refinement of ASM Models Using SMV

Meral Shirazipour, Yassine Mokhtari and Sofiène Tahar

Electrical and Computer Engineering Department
Concordia University, Montreal, Canada
Email: {m_shiraz, mokhtari,tahar}@ece.concordia.ca

Technical Report

January 2001

Abstract

Gurevich's Abstract State Machines (ASMs) are computational models of systems specified in a high level language (ASM-SL). In this paper, we investigate the pros and cons of the automatic verification of ASM models using the ASM-Workbench and its extention ASM-SMV, which supports computer-aided verification by model checking. The results obtained are based on the validation and verification of a design block of a communication chip. We conclude this work by some suggestions on the automated process of verifying ASM models.

Key words: Formal Hardware Verification, Abstract State Machines, Model checking, Refinement.

1 Introduction

Model checking is the most appealing approach to *formal verification* because of its full automation [13]. In spite of the successful use of this technique on systems of industrial complexity, model checking has its limitations. The problem with model checkers is that they cannot verify large designs due to *state explosion* [6]. In order to overcome this limitation, research is being conducted on the development of hybrid approaches to formal verification. Absract State Machines(ASMs) [12] are a recent formalism which could be presented as a way to overcome or at least delay the state explosion problem. A solution could be a reduction to an abstract model with a reduced state space. To start, an abstract ground model can be obtained using ASMs. This will give an overview of the problem. The key point will be the stepwise refinement of this abstract model in order to reach a version close to the actual system. The refined model must operate in a state space allowed by model checkers. Once expressed in the model checker language, requirements, e.g., safety and liveness properties can be verified.

Abstract State Machines (ASMs) have become a successful methodology for specifying and verifying complex systems. An ASM model describes the state space of the system by means of universes or functions, and the state transitions by means of transition rules. Recently, investigations are undertaken to evaluate the *automatic verifiability* of ASM models. We distinguish mainly two approaches. The first approach [5, 19] is based on the process of transforming an ASM model into the language of a symbolic model checker, SMV [15]. While in the other, more elaborated, approach [17], the ASM model is represented as a graph called CGL* where each node is a first-order structure and the properties are written in CTL*.

The present paper can be viewed as an investigation of the *automatic verifiability* of ASM models using the interface ASM-SMV in the context of hardware designs. Further, to overcome the limitation of the automatic verification, a number of techniques have been investigated. Using decomposition together with abstraction we are able to automatically verify a design block of a communication chip, an ATM (Asynchronous Transfer Mode) [10] switch fabric.

In this work, applying model checking on ASMs is analysed by using the example of the ATM switch's Timing block. The Timing block controls the timing of the arbitration decision of the switch fabric. The SMV model checker is used because it is proven effective when automatically verifying properties of combinational logic and interacting finite state machines. The behavioral and structural specifications of the Timing block are first expressed in the ASM language and simulated using the ASM-Workbench. This accounts for the validation of the model. Then the ASM is taken to the verification stage which is done by expressing the ASM in the SMV language using the ASM-SMV transformer [5, 18], a tool developped for the automatic conversion of ASMs to the SMV language. The requirements (safety and liveness properties) are then expressed in temporal logic and the verification is performed on SMV. Moreover, a refinement proof between the specification and implementation of the Timing block has been established in SMV.

2 Basic Concepts of Abstract State Machines

Abstract State Machines [12] bring together the specification methods with the computational models. Mathematical modelling of non-mathematical systems is achieved by the abstraction of states as algebras. The latter provides a powerful framework for the mathematical modelling of discrete dynamic systems. The idea consists of modelling the system with such a representative evolving algebra whose observation and experimentation will establish the correctness of the model.

2.1 Computational model

Computations. Abstract state machines are used for modelling systems as transition systems. By continuously executing transitions on an initial state S_0, finite or infinite sequences of States $\{S_i\}$ result. These sequences are called computations or runs of the abstract model of the system. The simulation of a system modelled as an abstract state machine is its computations.

States. Abstract state machines have a vocabulary (signature) which is a finite set of function names with a fixed arity. Each vocabulary must contain the nullary function names *true, false, undef* along with the equality sign and the boolean operators ($\neg, \wedge, \vee$, etc.). Function names are declared as static or dynamic. Static functions have the same interpretation in any state S of a particular execution of an ASM, while dynamic functions' interpretations are changed by the transitions. The nullary functions and boolean operators are static. A state S of an ASM with a given vocabulary is a non-empty set, called the superuniverse of S, along with interpretations of each function name in that vocabulary over the non-empty set. A state can also be reffered to as static algebra. Algebras are structures of first-order logic without relations (relations are treated as boolean-valued functions).

Transitions. force a state S into a successor state S' by applying a dynamic function on S. The transition occurs when S goes through a finite update set. This is done by simultaneously firing all the members of the update set at the state S. An update set $\triangle = (f, \bar{x}, y)$ where $f : T_1 \ldots T_n \rightarrow T$ is a dynamic function name and T_i is a type. $\bar{x} \in T_1 \times \ldots \times T_n$ and $y \in T$. S' is obtained by firing $\triangle$ at S. For example after running the transition T bellow, the value of c is equal to 20 and not 4 since all the members of the update set are thrown simultaneously.

```
dynamic function a:  INT initially 10
dynamic function b:  INT initially 10
dynamic function c:  INT initially 0
transition T ==
   block
      a := 1
      b := 3
      c := a + b
   endblock
```

2.2 The ASM Language

Terms. are used to refer to admissible values. The syntax of terms is defined recursively as in first-order logic. The same recursion is used to define Boolean terms.

- A variable is a term. If a variable is Boolean, the term is also Boolean.
- If f is a function name of arity r in a given vocabulary and $t_1, \ldots, t_r$ are terms, then $f(t_1, \ldots, t_r)$ is a term. The composed term is Boolean if f is relational.

Ground terms are terms without variables. By analogy, other syntactical objects without variables will be called ground. *Atomic Boolean terms* are terms of the form $f(\bar{t})$, where f is a relation name. *Boolean terms* are built from atomic Boolean terms by means of the Boolean operations.

Transition rules. define the changes over time of the states of abstract state machines. The *update instruction* is the simplest transition rule. It has the following form: $f(\bar{a}) := b$ where f is a dynamic function name, $\bar{a}$ is a tuple of terms and b is a term. For example, if $\bar{x}$ and y are the values of $\bar{a}$ and b in the present state, $f(\bar{x}) := y$ in the next state. Another transition rule is the *Block rule* which is the combination (sequence) of transition rules. The execution of a block rule is the simultaneous execution of the sequence of the transition rules. The *Conditional rule* is also a transition rule. It is written as

```
if a_0 then R_0
elseif a_1 then R_1
   ⋮
elseif a_k then R_k
endif
```

a_i are first-order Boolean terms. R_i are the transition rules. The conditional rule is executed in state S by evaluating a_i in S. If it evaluates to true then R_j is executed, for a_j true and a_i false ($i < j$).

The ASM-SL Ennvironment. The ASM language, including all constructors described above, is supported by the "ASM Workbench" [4], which provides type checking of ASM specifications as well as their simulation and debugging. The specification language ASM-SL [3] includes some additional features which are necessary for practical modelling tasks: constructors for defining types, functions, and macros, as well as a set of predefined data types (Booleans, integers, tuples, lists, finite sets, etc.).

3 Case Study: Timing Block

The Fairisle network is a real ATM (Asynchronous Transfer Mode) [10] switching network designed and in use at University of Cambridge for multimedia applications [14].

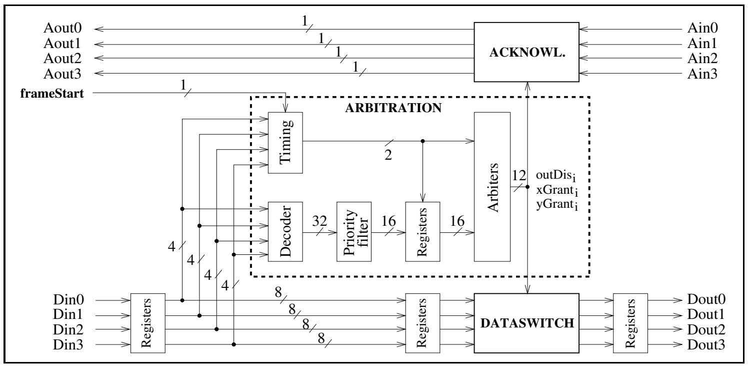

Figure 1: Fairisle ATM Switch Fabric

The Fairisle switch fabric (see Figure 1) forms the heart of the Fairisle switch. The fabric is composed of three modules, namely Acknowledgement, Arbitration and Data Switch. Further dividing the Arbitration module, we have the blocks Timing, Decoder, Priority Filter and Arbiters as sub-modules. In our example, we have taken the Timing block to be our case study for specification and verification using ASMs.

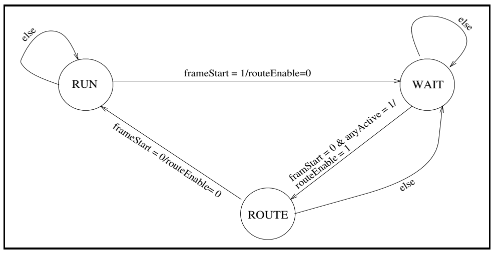

Figure 2: Timing block: behavior model (FSM)

The Timing block controls the timing of the arbitration decision based on the *frameStart* signal (see Figure 1) and the time the routing bytes arrive. The *frameStart* signal ensures that the environment of the Timing block injects data (cells) into the fabric synchronously for the cells to arrive at the same time. After receiving the *frameStart* signal, the Timing block waits for cells. These cells could be either *active* or *inactive*, i.e., containing the same or different destinations respectively. The signal *anyActive* captures the active cells. If these cells are active, the Timing block sends a *routeEnable* signal to the Arbiration block to route the data to the appropriate channels. Figure 2 illustrates the abstract behavior model (Finite State Machine) of the Timing block and Figure 3, its hardware implementation.

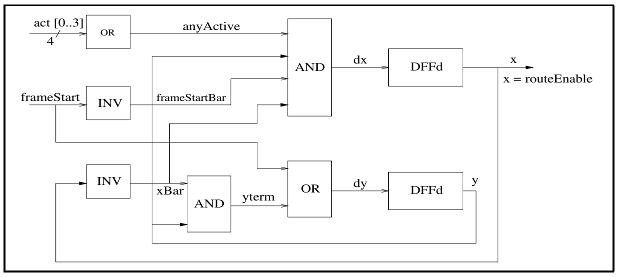

Figure 3: Timing block: hardware implementation

3.1 Modelling the Timing Block's Specification in ASM

The state of the machine is represented by *dynamic functions*. The behaviour of the machine can be modelled by means of nullary functions. First, the nullary function *routeEnable* is used to represent the outgoing signal. Then, the nullary function *state* is an internally updatable function used to guide the dynamic control, namely: *run, wait, and route*, i.e:

```
dynamic function routeEnable: BOOL initially false
dynamic function state: {run, wait, route} initially run
```

External functions are used to represent environment operations, i.e., incoming signals.

```
external function frameStart: BOOL
external function anyActive: BOOL
```

During each cycle of execution, the state machine evaluates the incoming signals with oracle functions, that is, uses the inputs provided by the environment to compute the new states and the outgoing signals. This timing behavior is modelled by the following timing rules:

```
transition T1 ==
   if state = run and frameStart then
      state := wait
      routeEnable := false
   endif


transition T2 ==
   if state = wait and not(frameStart) and anyActive then
      state := route
      routeEnable := true
   endif
```

```
transition T3 ==
   if state = route and not(frameStart) then
      state := run
      routeEnable := false
   else
      state := wait
   endif
```

3.2 Modelling the Timing Block's Implementation in ASM

The implementation of the Timing block is illustrated in Figure 3. The behavior of the circuit can be modelled with dynamic nullary functions representing the internal signals:

```
dynamic function routeEnable:  BOOL initially undef
dynamic function x:  BOOL initially undef
dynamic function y:  BOOL initially undef

dynamic function dx:  BOOL initially undef
dynamic function dy:  BOOL initially undef

dynamic function frameStartBar :  BOOL initially undef
dynamic function xBar:  BOOL initially undef
dynamic function yterm:  BOOL initially undef
```

The nullary function routeEnable is the output signal of the circuit. The input signals are provided by the environment and are therefore represented by *External functions*:

```
external function frameStart:  BOOL
external function anyActive:  BOOL
```

The functionalities of the gates can be represented by means of static functions. The timing delays of the gates are not considered.

```
static function orgate2(in0,in1) ==
   if in0 or in1 then
      true
   else
      false
   endif


static function orgate4(in0,in1,in2,in3) ==
   if in0 or in1 or in2 or in3 then
      true
   else
      false
   endif


static function andgate2(in0,in1) ==
   if in0 and in1 then
      true
   else
      false
   endif


static function andgate4(in0,in1,in2,in3) ==
   if in0 and in1 and in2 and in3 then
      true
   else
      false
   endif


static function invgate(in0) ==
   if in0 then
      false
   else
      true
   endif
```

The value of the internal signals and the output are obtained by running two transitions (gates and DFFd). The second transition (DFFd) is necessary for modelling the time delay between the signals dx and x, and similarly dy and y, through registers.

```
transition gates ==
   block
      frameStartBar := invgate(frameStart)
      dx := andgate4(anyActive, y, frameStartBar, xBar)
      xBar := invgate(x)
      yterm := andgate2(xBar, y)
      dy := orgate2(frameStart,yterm)
   endblock
```

```
transition DFFd ==
   block
      x := dx
      y := dy
      routeEnable := dx
   endblock
```

3.3 Validating the Timing Block

Validation is the process of identifying and recording the functional behavior of a design. In order to get the correct verfication results from the models developped above, it is crucial to make sure that the abstraction corresponds to the actual design. Moreover, formal verification by stepwise refinement requires well-defined models of the design at the implementation level as well as the specification level. This brings the need for the validation of the ASM models. This validation is performed by means of simulation. The ASM models of the Timing block are simulated using the *ASM Workbench* [4] which provides type checking of ASM specifications as well as their simulation and debugging (see Figure 4 for a snapshot of the tool).

Validation is carried out by tests to ensure that desirable scenarios are allowed, and undesirable ones forbidden by our model. The key success of validation is the executability of our model, since ASM descriptions are transition systems.

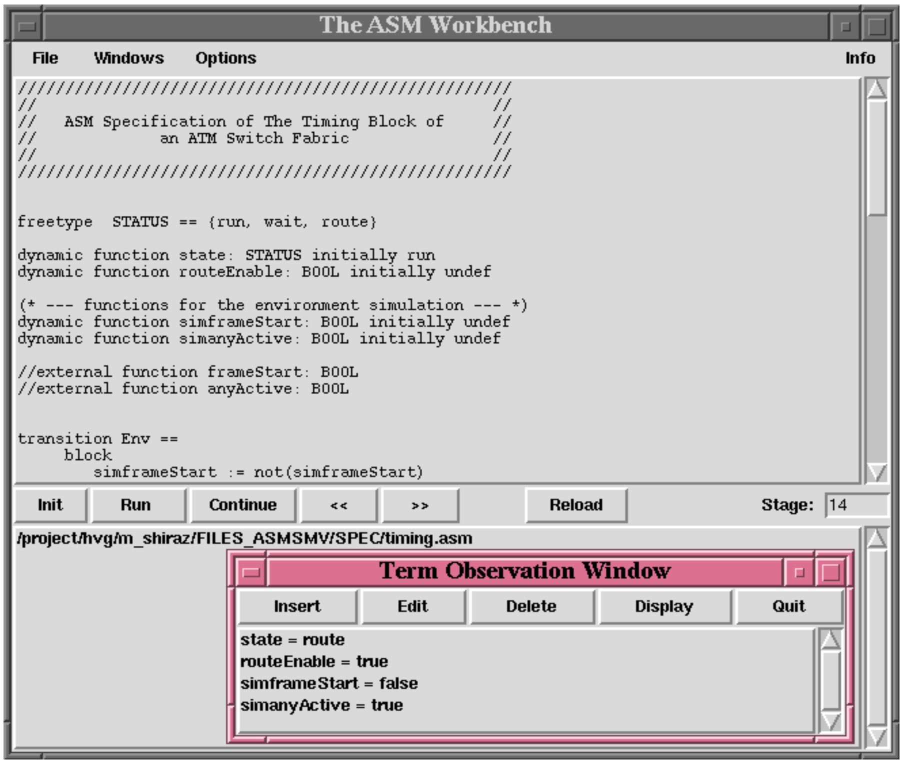

Figure 4: The ASM Workbench

3.3.1 Validating the Timing Block's Specification

The specification of the Timing block is provided by the finite state machine of Figure 2. This three-state machine is what we used to validate the ASM model of the specification. The transitions are run in a random sequence and the terms are observed from the Term Observation Window of the ASM Workbench (Figure 4). The input values are provided by two extra transitions that are not part of the actual model. The model was successfully validated as the state transitions and their corresponding output values given by the finite state machine were obtained.

3.3.2 Validating the Timing Block's Implementation

The implementation of the Timing block presented in Figure 3 is used to validate its ASM model. An extra transition is needed to simulate the environment. This transition assigns eight specific set of values to the inputs *frameStart* and *anyAc-*

tive. The following table shows these inputs and their corresponding output values. These are sufficient for the validation because the *frameStart* signal changes as many times as necessary for the implementation to go through all relevent states.

Time (clock)	0	1	2	3	4	5	6	7
frameStart	0	0	1	0	0	0	1	0
anyActive	0	1	1	0	1	1	0	1
routeEnable	0	0	0	0	0	1	0	0

3.4 Verification of the Timing Block

Winter [18] introduced an interface from the ASM Workbench to SMV based on an ASM-to-SMV transformation. SMV [15] is a symbolic model checker based on binary decision diagrams (BDDs) [2]. It extracts a finite-state model from an SMV program and uses an exhaustive reachability analysis to determine whether the model satisfies the specifications expressed in temporal logic CTL [9]. If the model does not satisfy some specification, SMV will produce an execution trace that shows why the specification is false. Besides partial specification (properties), SMV can check an implementation against a complete specification through a refinement mapping process.

The ASM-to-SMV transformations [19] are based on the finite size of states. Consequently, the subset of ASM translated into SMV has finite and fix universes, i.e., funtions' s domains and ranges are finite and may not be extended by transition rules. The basic tranformation schema deals only with (i) external and nullary dynamic functions, (ii) data types including booleans, integers and enumerated types, and (iii) static functions that have corresponding predefined operations in SMV. In [5] the schema is extended to cope with n-ary functions. Hence, the transformation approach can tackle much broader applications.

3.4.1 Model Checking of the Specification Model

CTL properties. Once an ASM model is transformed into the SMV language, we benefit from model checking properties concerning safety and liveness. For illustration purposes using our Timing block, we show the verification of a liveness property stating that arrived cells are eventually routed. We formalize this requirement in CTL:

$$\mathbf{AG}((\neg frameStart \wedge anyActive) \rightarrow \mathbf{AF}\, routeEnable)$$

Proving that an ASM model satisfies its requirement does not depend only on states and transition rules but also on some additional assumptions on which we give some examples during our Timing block verification below.

Fairness Constraint. Verifying the correctness of the abstract model, i.e., comparing the abstract model against the CTL property stated above, needs a fairness requirement. Fairness conditions are used to constrain the nondeterministic choices present in abstract models. For example, one possible behavior of the transition system (see Figure 2) may cause the system to stay forever in the *run* state if the transition condition is never satisfied. Fairness conditions assert that a transition that is enabled often enough will eventually be executed. Hence by adding a fairness requirement stating that the abstract model must visit infinitely often the state *route*, we are able to prove that our abstract model satisfies our CTL requirement. Formally, we will add to the ASM-SMV model the following condition in SMV notation:

$$\begin{array}{c}\text{FAIRNESS} \\ state = route\end{array}$$

SMV did not generate any counter-examples. Therefore it was proved that the abstract model satisfied the CTL requirement.

3.4.2 Model Checking of Implementation Model

Verifying the correctness of the Timing block's implementation requires an additional assumption about the *environment*. Indeed, we have abstracted the environment by means of external functions in the abstract ASM model. In the next lower level ASM, these functions are assumed to satisfy some logical properties. These assumptions must be *reasonable* in order to guarantee the correctness. Back to our example, the environment of our implementation is as follows:

- The *frameStart* and *anyActive* signals are low initially.
- The *frameStart* signal remains high until it becomes low.
- After the point where the *frameStart* signal becomes low and the *anyActive* signal is low, eventually the cells will be routed.

We formalize these requirements in CTL:

$$(\neg frameStart \wedge \neg anyActive) \wedge \\ \textbf{AG}\left(\begin{array}{l}(frameStart\ \textbf{AU}\ \neg frameStart)\wedge \\ \textbf{AF}\ frameStart \wedge \\ \textbf{AF}\ (\neg frameStart \wedge anyActive)\end{array}\right)$$

Finally, we use an *assume-guarantee* paradigm [11] to prove the correctness of our implementation by splitting the proof in two parts. This modular reasoning must be used in order to avoid state explosion.Here is an example of how assume-guarantee reasoning works in general. Given that a system S is composed of modules M1 and M2, where there is no feedback from M2 to M1: $S = M1||M2$, in order to prove with a simple assume-guarantee that $(M1||M2) \models \alpha$, we first must prove that $(\beta||M2) \models \alpha$, and then $M1 \models \beta$.

In our model checking example, we first must show that the implementation satisfies our CTL requirement (*guarantee*) assuming that the behavior of the environment corresponds to our CTL environment (*assumption*). Then, this assumption is discharged by proving that the actual environment does indeed meet the CTL assumption. Using SMV, the first step is proved while we keep the assumption about the environment until we tackle the actual implementation of the environment:

using CTL-environment *prove* CTL-requirement
assume CTL-environment

Once again, SMV did not find any counter-examples. Therefore it was proved that the implementation also satisfies the CTL requirement.

3.4.3 Verification by Refinement Mapping

The last step of our verification consists of proving that our abstract model is correctly implemented by our ASM implementation. This is usually done by introducing *refinement maps* [1] that translate the behavior of the abstract model into the behavior of given interfaces and structures in the low design. SMV supports this methodology by introducing a new construct called *layer*. A layer defines low-level implementation signals as a function of abstract model signals and thus provides a refinement map. In our case study, the low-level implementation of *routeEnable* signal must be simultaneously consistent with its abstract definition thus the refinement map is an identity. However, it could be a sophisticated one in other modules of the design. The following code is the refinement map that relates our abstract model to the implementation.

```
MODULE main(){
VAR
act0 : boolean;
act1 : boolean;
act2 : boolean;
act3 : boolean;
frameStart : boolean;
routeEnable: boolean;
c : circuit(frameStart,anyActive, routeEnable);
b : behavior(frameStart,anyActive);
/** refinement mapping */
layer spec : {
 c._routeEnable := b._routeEnable;
}
DEFINE
anyActive := act0 | act1 | act2 | act3;
```

The SMV module *circuit* corresponds to the implementation ASM, whereas the *behavior* module corresponds to the abstract ASM model. This code was successfully verified, i.e, no counter-examples were produced by SMV. Therefore it was proved that the abstract model is correctly implemented by the ASM implementation.

4 Conclusion

In this paper we report our experience with model checking and refinement of ASM models using the ASM-SMV interface as applied to the Timing block of an ATM switch fabric as case study. Althougth, this case study is simple, the process of the verification involves many interesting techniques, namely model abstraction, fairness constraints, assume-guarantee paradigm and refinement mapping.

It is worth to notice that for now there is no feature in the ASM language to express fairness assumptions, assume-guarantee paradigms or refinement. Of course, the two last items could be expressed mathematically but we are interested here in proving the correctness of ASM models automatically. Hence, the ASM language should be extended in this direction and equipped with a formal specification language like CTL.

The case study of the Timing block that is used here is also investigated by other approaches. Curzon [8] formally verified the refinement of the Timing block using the theorem-prover HOL. In order to overcome the cumbersome proof process of interactive theorem proving (HOL), a hybrid approach [16] which combines HOL and MDG was also used where the verification of the Timing block deals with the equivalence checking of the abstract model and its implementation. In future work, we intend to extend our case study to verify the whole ATM switch fabric.

Multiway Decision Graphs (MDGs) [7] represent the state space in a more compact structure than ROBDDs for designs containing a large data path. An MDG is a finite, directed acyclic graph (DAG) where each node is a first-order structure, hence it is suitable for representing ASM models. Furthermore, the verification procedures cover combinational and sequential equivalence checking. The combinational verification provides the equivalence checking of two combinational circuits. The sequential verification provides invariant checking and equivalence checking of two state machines. Further future work will concentrate on the combination of ASMs and MDGs based on the progress work of Winter [18].

Acknowledgment

This work has been partially supported by an undergraduate scholarship from NSERC and GRIAO. The authors are greatful to Kirsten Winter and Egon Börger for their comments and suggestions.

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 81(2):253–284, 1991.

[2] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.

[3] G. Del Castillo. ASM-SL, a Specification Language based on Gurevich's Abstract State Machines: Introduction and Language Definition. Heinz Nixdorf Institut Paderborn, Germany, 1999.

[4] G. Del Castillo. The Asm Workbench: an Open and Extensible Tool Environment for Abstract State Machines. Heinz Nixdorf Institut, Paderborn, Germany, 1999.

[5] G. Del Castillo and K. Winter. Model checking support for the ASM high-level language. *International Conference for Tools and Algorithms for the Construction and Analysis of Systems, TACAS'2000*, April 2000.

[6] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, January 2000.

[7] F. Corella, Z. Zhou, X. Song, M. Langevin, and E. Cerny. Multiway Decision Graphs for Automated Hardware Verification. *Formal Methods in System Design*, 10(1):7–46, 1997.

[8] P. Curzon. The Formal Verification of the Fairisle ATM Switching Element. Techical Report 329, Computer Laboratory, University of Cambridge, U.K., March 1994.

[9] E.A. Emerson. *Temporal and modal logic, Handbook of theoretical computer science*. Elsevier Sciences B.V. J. van leeuwn north holland edition, 1990.

[10] D. Ginsburg. *ATM Solutions for Enterprise Internetworking*. Addison Wesley, 1996.

[11] O. Grumberg and D. E. Long. Model Checking and Modular Verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, May 1994.

[12] Y. Gurevich. ASM Guide. Technical Report CSE-TR-336-97, Michigan University, 1997.

[13] C. Kern and M.Greenstreet. Formal Verification in Hardware Design: A Survey. *ACM Transactions*, 4(2):123–193, April 1999.

[14] I.M. Leslie and D.R. McAuley. Fairisle: An ATM Network for the Local Area. *ACM Communication Review*, 19(4):327–336, 1991.

[15] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[16] V.K. Pisini, S. Tahar, P. Curzon, O. Ait-Mohamed, and X. Song. Formal Hardware Verification by Integrating HOL and MDG. *Proc. ACM 10th Great Lakes Symposium on VLSI (GLS-VLSI'00)*, pages :23–28, March 2000.

[17] M. Spielmann. Automatic verification of abstract state machines. LNCS 1633:431–442, July 1999.

[18] K. Winter. Model checking for abstract state machines. *Journal for Universal Computer Science*, 3(5):689–702, 1997.

[19] K. Winter. Towards a Methodology for Model checking ASM: Lessons learned from the FLASH Case study. *International Workshop on Abstract State Machines (ASM'2000)*, March 2000.