GLS-VLSI'98
IEEE Great Lakes Symposium on VLSI
Lafayette, Louisiana, USA, February 1998
IEEE Computer Society Press, pp. 368-373

Practical Approaches to the Automatic Verification of an ATM Switch Fabric Using VIS

Jianping Lu and Sofiène Tahar

Concordia University, ECE Dept., Montreal, Quebec, H3H 1M8 Canada
Email: {jianping, tahar}@ ece.concordia.ca

Abstract

In this paper we present several practical methods for formally verifying an Asynchronous Transfer Mode (ATM) network switching fabric using the Verification Interacting with Synthesis (VIS) tool. We produced Verilog RTL behavioral and netlist structural descriptions of the switch fabric at different levels of hierarchy and established several abstracted models of the fabric. Using various techniques presented in the paper, we provided a number of relevant liveness and safety properties expressible in CTL, and accomplished their verification in reasonable CPU time. Moreover, we performed equivalence checking between the structural and behavioral descriptions of each submodule of the implementation hierarchy.

1. Introduction

Verification is increasingly becoming the bottleneck in the design flow of communication networks systems. Simulation is very expensive in terms of time and exhaustive simulation is virtually impossible. As a result, formal verification of digital systems is gaining interest, as the correctness of a formally verified design implicitly involves all possible input values.

ATM (Asynchronous Transfer Mode) is a network technology for addressing the variety of needs for new high-speed, high-bandwidth applications. It has been hailed as the most important communication mechanism in the foreseeable future. However, there is currently little experience on the application of formal verification to ATM network hardware. For instance, Curzon [4] formally verified the 4 by 4 fabric of the Cambridge Fairisle switch using the HOL theorem prover [6]. Tahar *et al.* [11][7] verified the same switch fabric in an automatic fashion using the MDG (Multiway Decision Graphs) tools [3] by property checking and equivalence checking. Another case of formal verification of an ATM circuit was made by Chen *et al.* [2] at Fujitsu Digital Technology Ltd., where the authors identified a design error in an ATM circuit using the tool SMV (Symbolic Model Verifier) by verifying some properties expressed in CTL (Computational Tree Logic)[10].

In this paper, we present our results of formally verifying an ATM network component using VIS [12]. VIS (Verification Interacting with Synthesis) is a tool developed at the University of California, Berkeley, which integrates the verification, simulation and synthesis of finite-state hardware systems. The device we investigated is the Fairisle 4 by 4 switching fabric. It performs the actual switching of data cells in the ATM Fairisle communication network [8], designed and in use at the Computer Laboratory of the University of Cambridge.

We wrote the Verilog RTL behavioral description of the switch fabric at different levels of abstraction, and translated the original Qudos HDL [5] netlist structural description of the switch fabric to Verilog. We achieved property checking on abstracted models of the switch using several approaches and performed equivalence checking between the behavioral and structural descriptions of submodules. In addition, we succeeded in detecting several injected design errors using both property checking and equivalence checking.

The rest of the paper is structured as follows: In Section 2, we describe the switch fabric in terms of behavior and structure implementation. In Section 3, we investigate property checking of the switch fabric. In Section 4, we present novel techniques for model checking enhancement. In Section 5, we introduce the methodology and results of the equivalence checking. The error detection using VIS is presented in Section 6 and Section 7 concludes the paper.

2. The ATM Switch Fabric

The Fairisle ATM switch consists of input port controllers, output port controllers and a switch fabric (Figure 1).

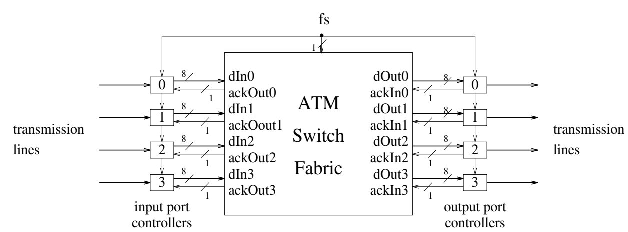

Figure 1. Structure of the Fairisle ATM switch

The port controllers synchronize incoming and outgoing data cells, appending control information in the front of the cells in a routing tag (Figure 2).The routing bits in the routing tag indicated the destined output port of data cells. The priority bit is used for arbitration, where the high priority cells are given precedence. For those cells with the same priority, round-robin arbitration is performed. The output controllers are informed of whether their cells were successful or not through the acknowledgments generated by the output ports. The port controllers and switch fabric all use the same clock, and they also use a higher-level cell frame clock: the *frameStart* signal (*fs*).

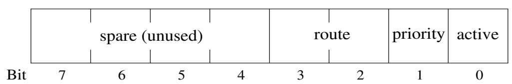

Figure 2. Routing tag of a Fairisle ATM cell

The behavior of the switch fabric is cyclic. In each frame, the fabric waits for cells to arrive, reads them in, processes them, sends successful ones to the appropriate output ports and sends acknowledgments. It then waits for the next round of cells to arrive. The cells from all the input ports start when a particular bit (the active bit) of any input port goes high.

Figure 3 shows a block diagram of the switch fabric implementation. It consists of an arbitration unit, an acknowledgment unit and a dataswitch unit. The arbiters make arbitration decisions for each output port *i* by setting values for the corresponding *outputDisable[i], xGrant[i],* and *yGrant[i]* boolean signals, according to which the dataswitch switches data from input ports to expected output ports. And the acknowledgment unit passes appropriate acknowledgment signals to the input ports according to these signals.

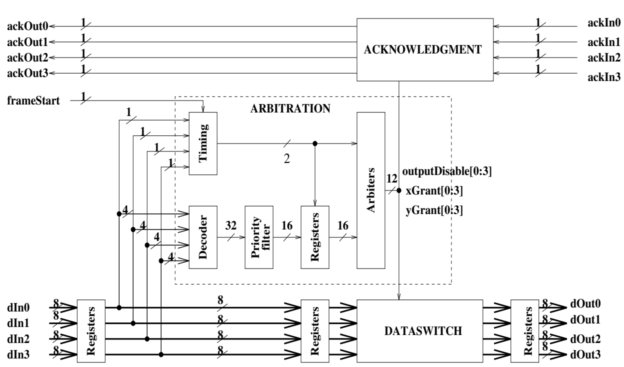

Figure 3. Fairisle switch fabric implementation

3. Property Checking

State space explosion is a well-known problem in FSM-based verification approaches. Although the number of states that can be handled has been increased dramatically since the introduction of BDDs [1] as a symbolic representation of a set of states, BDDs still have their limit and cannot handle designs with a lot of state-holding elements. There are 210 latches in the switch fabric that we are considering. In this section, we describe our methodology and results to deal with the state space explosion we faced in property checking.

3.1. Environment for the port controllers

Within VIS properties are expressed in CTL. However, in order to give an explicit value for an input signal in a CTL expression, we have to declare input signals (variables) as non-deterministic register variable. Furthermore, in order to express explicit time points in certain properties, we need to represent them via explicit states in the corresponding CTL expressions. Therefore, we established an environment state machine which imitates the behavior of the port controllers and also constraints the number of possible inputs to the switch fabric.

We modeled the port controllers as a finite state machine. Figure 4 represents an abstracted environment state machine describing the behavior of the port controllers. Arrows denote state transitions, and t_s, t_h and t_e denote start of a frame, start of an active cell (header arrival) and end of a frame (which is the start of the next frame), respectively. States S1 to S7 represent the cyclic behavior of the fabric, where one cycle corresponds to one frame.

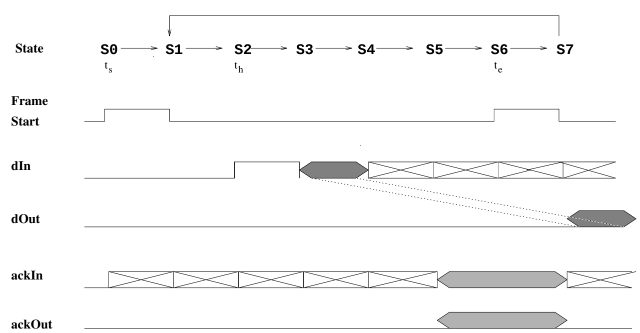

Figure 4. Abstracted environment state machine with related timing diagrams

3.2. Properties description

After establishing the environment state machine, we consider several properties of the fabric including liveness and safety properties. In following CTL expressions, “!”, “−>”, “*” and “+” denote logical *“not”, “imply”, “and*” and *“or*”, respectively.

Property1: At time t_h (state S2), if input port 0 chooses output port 0, potentially the data in input port 0 will be transferred to output port 0. In CTL, this *liveness* property is expressed as follows.

```
AG (dIn0[0] = 1 * dIn0[2] = 0 * dIn0[3] = 0 * state = S2
-> EF (dIn0^S3 == dOut0));
```

where $dIn0^{S3}$ stores the value of *dIn0* in state S3.

Next, we consider several *safety* properties. In following, we present four such properties of the fabric along with their CTL expressions. An extensive set of further liveness and safety properties is reported in [9].

Property 2: The data bytes in a cell are transferred from input port 0 to output port 0 sequentially with 4 clock cycle delay.

```
EG (state = S3 -> AX AX AX AX (dOut0 = dIn0^S3));
```

Property 3: From time t_h+1 (state S3) to time t_h+4 (state S6), the default value (zero) is put on the data output ports.

```
AG ((state = S3 + state = S4 + state = S5 + state = S6)
-> dOut0 = 0 * dOut1 = 0 * dOut2 = 0 * dOut3 = 0);
```

Property 4: If the input port 0 chooses output port 0 with priority, the data cells of input port 0 will be transferred to output port 0 with 4 clock cycles delay.

```
AG (dIn0[3:0]=0011 * dIn1[1]=0 * dIn2[1]=0 * dIn3[1]=0 *
state = S2 -> AX AX AX AX AX ( dOut0 == dIn0^S3 ));
```

Property 5: If input port 0 chooses output port 0 with priority, acknowledgment signal will be passed from output port 0 to input port 0.

```
AG (dIn0[3:0]=0011 * dIn1[1]=0 * dIn2[1]=0 * dIn3[1]=0 *
state = S2 -> AX AX AX ( ackOut0 == ackIn0 ));
```

3.3. Abstracted fabric

Due to state space explosion, we did not succeed in checking the properties on the original fabric. To cope with this problem, we reduced the datapath of the dataswitch to 1 bit. Because the behavior and structure of 1-bit datapath is exactly the same as that of other 7 bits, this abstraction is valid. The arbitration and acknowledgment unit remained as in the original design. The abstracted model is shown in Figure 5.

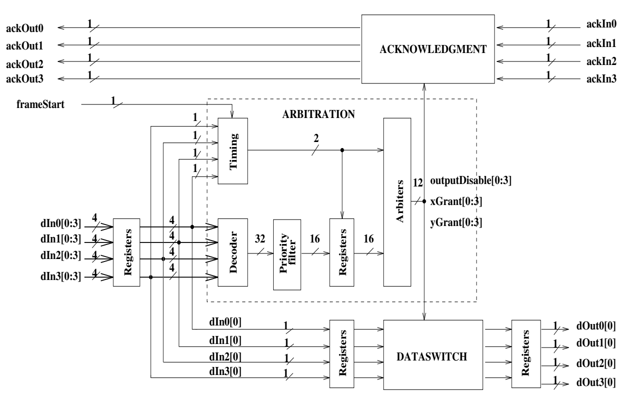

Figure 5. Abstracted switch fabric

Based on this abstracted model, we checked the five properties described above. The CPU time (elapse time), memory usage and nodes allocated for property checking are shown in Table 1. This experiment as well as all the results in this paper were done on SUN Sparc 20 workstation (55MHz/256MB).

Table 1. Property checking on abstracted fabric

Properties	CPU time (sec.)	Memory (MB)	Nodes allocated
Property 1	3933.9	40.3	84,199,139
Property 2	4550.7	43.0	90,371,031
Property 3	3593.4	32.4	93,073,140
Property 4	3679.7	40.9	79,687,784
Property 5	414.8	5.3	4,180,124

4. Enhancement of property checking

Although we succeeded in checking all the above properties on the abstracted fabric, we found that almost all the properties are checked with unreasonable time (many hours machine usage time). This would be worse if the circuit under investigation is larger than the current switch fabric. Here, we discuss several approaches we adopted to speed-up property checking.

We noticed that most of the properties can be divided into several sub-properties which are much easier to verify in model checking. If all the sub-properties pass the model checking, and we show that the conjunction of these sub-properties implies the property, then we conclude that the property passes the model checking too. We call this *Property Division*. In following we propose two approaches to divide a property: *Cascade Property Division* and *Parallel Property Division*.

4.1. Cascade property division

Cascade property division is to divide a property into several sequentially related sub-properties. For checking some sub-properties, new environment machines will eventually be required. We use property 5 to explain this approach. The original CTL expression of property 5 is:

```
AG (dIn0[3:0] = 3 * dIn1[1] = 0 * dIn2[1] = 0 *
dIn3[1] = 0 * state = S2
-> AX AX AX (ackOut[0] == ackIn[0]));
```

To divide this property, we introduced the intermediate signals (variables) *xGrant[0], yGrant[0]* and *outputDisable[0]*, yielding the following two sub-properties:

Sub-property 1:
```
AG (dIn0[3:0] = 3 * dIn[1] = 0 * dIn2[1] = 0
* dIn3[1] = 0 * state = S2
-> AX AX AX (state = S5 * xGrant[0] = 0 *
yGrant[0] = 0 * outputDisable[0] = 0));
```

Sub-property 2: `AG (state = S5 * xGrant[0] = 0 * yGrant[0] * outputDisable[0] = 0 -> ackOut0 == ackIn0);`

For sub-property 2, some input signals (variables) like *xGrant[0], yGrant[0]* and *outputDisable[0]* are not in the environment state machine of the abstracted fabric. We hence established a new environment state machine where the behaviors of the signals *xGrant[i], yGrant[i]* and *outputDisable[i]* are given.

Table 2 gives the comparison between the property checking with cascade property division and the property checking without it. The CPU time for checking property 5 is enhanced by 41 times through cascade property division.

Table 2. Cascade division in property checking

Property 5		CPU time (sec.)	Memory (MB)	Nodes allocated
no cascade	prop. division	414.8	5.3	4,180,124
cascade property division	sub-property1	5.1	1.9	109,438
	sub-property2	4.9	1.7	78,743
	Total	10.0	-	-

4.2. Parallel property division

While cascade property division introduces sequentially related intermediate variables to divide a property, parallel property division splits a property into several parallel sub-properties without introducing any intermediate variable. It checks every sub-property by an abstracted model that is stripped from a design regularly. In this approach, the design structure must be disassembled at some specific location. We use property 3 to illustrate this approach.

The original CTL expression of property 3 is:

```
AG (state = S3 + state = S4 + state = S5 + state = S6
-> dOut0[0] = 0 * dOut1[0] = 0 * dOut2[0] = 0 *
dOut3[0] = 0);
```

To verify this property, we separated it into four parallel sub-properties as follows:

Sub-property 1: `AG (state = S3 + state = S4 + state = S5 + state = S6 -> dOut0[0] = 0);`

Sub-property 2: `AG (state = S3 + state = S4 + state = S5 + state = S6 -> dOut1[1] = 0);`

Sub-property 3: `AG (state = S3 + state = S4 + state = S5 + state = S6 -> dOut2[2] = 0);`

Sub-property 4: `AG (state = S3 + state = S4 + state = S5 + state = S6 -> dOut3[3] = 0);`

For each sub-property, we established from the abstracted fabric an abstracted fabric *unit* for each port. Table 3 gives a comparison between property checking with parallel division and property checking without it for property 3. From this table, the CPU time for property checking has been enhanced by 73 times.

Table 3. Parallel division in property checking

Property 3		CPU time (sec.)	Memory (MB)	Nodes allocated
no parallel	prop. division	3593.4	32.4	93,073,140
parallel property division	sub-property1	10.9	2.6	158,652
	sub-property2	13.8	2.5	148,649
	sub-property3	13.4	2.5	165,660
	sub-property4	10.8	2.5	153,735
	Total	48.9	-	-

4.3. Latches Reduction

Since latches introduce states, reducing the number of latches in a model will greatly speed-up the property checking. For instance, the switch fabric uses latches to pause data for 1 clock cycle in its primary inputs and outputs. Ignoring these latches will not influence the state transitions within the design, but overall timing behavior has to be re-evaluated. However, just ignoring the latches will speed-up property checking dramatically. Table 4 shows how the CPU time of checking property 2 has been enhanced by nearly 100 times if we use latch reduction. In this example, the data output latches that are used to delay output data for 1 clock cycle (Figure 3) were ignored in the case of "abstracted fabric with latches reduction".

Table 4. Latches reduction in property checking

Property 2	CPU time (sec.)	Memory (MB)	Nodes allocated
no latch reduction	4550.7	4.32	90,371,031
latch reduction	23.1	3.15	235,004

Table 5 collects the number of latches among the original fabric, the abstracted fabric and one abstracted fabric unit (all including few additional latches used for the environment state machine). From previous experiments, we found that property checking is almost impossible using the original fabric, and it is very slow using the abstracted model as shown in Table 1. However, using the abstracted fabric unit, acceptable CPU time of property checking is achieved. Through more experiments of property checking in VIS, we found that the model which has around 50 latches can be used for a property checking within an acceptable CPU time in VIS. This experimental result was successfully used to established the models while using above approaches so that we were able to check a set of twenty-eight properties within reasonable CPU time [9].

Table 5. Number of latches in different models

	Original fabric	Abstracted fabric	Abstracted fabric unit
Number of latches	210	85	54

5. Equivalence Checking

Besides property checking, VIS supports combinational and sequential equivalence checking of two circuits. We attempted sequential equivalence verification between the Verilog structural description (which we translated from the original Qudos HDL implementation), and the Verilog behavioral description of the fabric based on its FSM specification. If both descriptions are equivalent, the correction of the fabric is proved. We first provided a behavior description of the whole switch fabric as one module and tried to verify its equivalence against the implementation of the whole fabric including all connections of submodules. However, we could not succeed in verifying it in VIS after three days continuous run on a SUN SPARC 20 workstation due to state space explosion, even though we used different techniques of model abstraction, combination with environment state machine, dynamic ordering, etc. We hence followed a second approach that modularizes the fabric to several parts that are similar to the hierarchical modules of the structural description, where each module will be, in addition, described in terms of its behavior specification. This second approach has the shortcoming that while we are able to check the correctness of separate submodules of the fabric structure, it is difficult to ensure the correctness of the network connecting all the submodules. This shortcoming will be complemented by simulation and property checking. This second approach, however, has the advantage that the developed behavioral descriptions of the submodules are close to that used in industry design synthesis, and hence may fit some kind of on-the-fly verification.

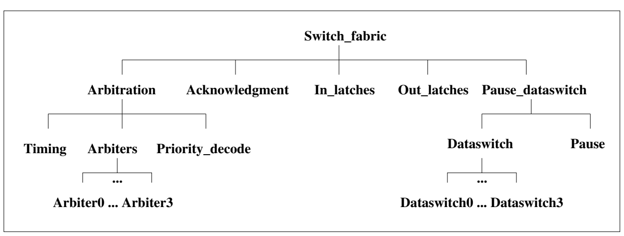

Figure 6. Modular structure of the switch fabric

Figure 6 represents the hierarchical structure of the fabric for which we provided behavioral descriptions. For the Fairisle switch fabric, we verified the sequential equivalence of the modules *In_latches, Out_latches, Pause, Timing, Priority_decode* and *Arbiters* easily. In addition, we succeeded checking the combinational equivalence of the *Acknowledgment* module. After using dynamic ordering, we checked the equivalence for *Dataswitch_i* and *Arbitration* modules, but they consumed too much CPU time (see Table 6). We failed to verify in VIS the modules *Dataswitch, Pause_dataswitch* and *Switch_fabric*. Table 6 gives the CPU time and number of latches for the modules investigated by equivalence checking.

Table 6. Equivalence checking of submodules

Component	CPU time (sec.)	# of latches
Acknowledgment	1.4	0
In_latches	4.2	32
Out_latches	4.2	32
Pause	4.0	32
Arbiter_i	1.4	3
Arbiters	13.3	12
Priority_decode	26.9	16
Timing	0.3	2
Dataswitch_i	1855.8	16
Arbitration	67860.0	30
Dataswitch	-	64
Pause_dataswitch	-	96
Switch_fabric	-	190

6. Design error detection

No errors were discovered in the above property and equivalence checking verification. For experimental purposes, however, we injected several design errors into the implementation: (1) We exchanged the inputs to the JK flip-flop that produces the *outputDisable* signal. This prevented the circuit from resetting. (2) We used the priority information of the input port 0 to control the input port 2. (3) We used an AND gate instead of an OR gate within the acknowledgment unit producing a faulty *ackOut0* signal. These three errors were detected by property checking and equivalence checking, and VIS generated counterexamples that exhibit the incorrect behavior of the corresponding signals. Experimental results are reported in Table 7, where the CPU time includes the time for property checking and counterexample generation. More details on further injected design errors are reported in [9].

Table 7. Detection of injected errors

Experiment	Property checking		Equivalence checking	
	Properties used	CPU time (sec.)	Affected submodules	CPU time (sec.)
Error 1	Property 3	82.5	Arbiters	20.6
Error 2	Property 4	49.3	Priority_dec.	24.0
Error 3	Property 5	15.4	Acknowledg.	1.7

An important advantage of formal verification using VIS is counterexample generation whenever equivalence or property checking fails. However, some counterexamples are difficult to analyze directly. In this work, we converted the counterexamples into Verilog-XL and analyzed them graphically. Figure 7 represents the procedure we adopted for verification using VIS in a general digital design flow. In [9] we report how we applied this design flow to reimplement the switch fabric using automatic synthesis and succeed in interacting VIS with Synopsys and Verilog-XL in an automatic fashion.

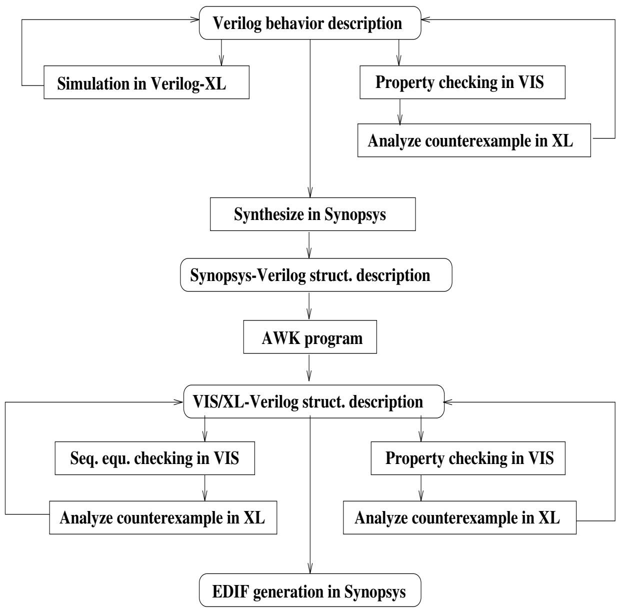

Figure 7. Digital design flow using VIS

7. Conclusions

In this study, we have explored the formal verification for a real ATM switch fabric. The main contribution of our work is the establishment of different environment and component abstraction techniques and the development of several approaches to avoid the state space explosion occurring in property checking and equivalence checking. The proposed abstraction and enhancement techniques are tool independent and can be applied to a wide range of regular design.

Although the current VIS is limited to circuits of moderate size whose BDDs can be constructed within the available memory space, we demonstrated that VIS can partially verify large circuits properly by using modular verification and abstract models. In this paper, we presented several novel methods such as the use of environment machines to facilitate property checking, property division and latch reduction which made VIS succeeding in checking properties on large circuits.

In [4] it is reported that time spent on the simulation of the Fairisle switch fabric would have been in the order of several weeks. However, using all the techniques described in the paper, a verifier is able to perform the verification within a couple of days.

In order to check the correctness of large designs by means of formal verification, the user must modularize and abstract the design. The design cannot always be viewed as a black box by the verifier, and so users must have a thorough knowledge of the design in order to verify properties in model checking. Therefore, it is the designer in person who is in the best position to verify the design, and formal verification should be used in the design flow as much as possible.

References

[1] Bryant R.: Graph-Based Algorithms for Boolean Function Manipulation; *IEEE Transactions on Computers,* Vol. C-35, No. 8, August 1986, pp. 677-691.

[2] Chen, B.; Yamazaki, M.; Fujita, M.: Bug Identification of a Real Chip Design by Symbolic Model Checking; *Proc. International Conference on Circuits And Systems* (ISCAS'94), London, UK, June 1994, pp. 132-136.

[3] Corella, F.; Zhou, Z.; Song, X.; Langevin, M.; Cerny, E: Multiway Decision Graphs for Automated Hardware Verification; *Formal Methods in System Design,* Vol. 10, No. 1, 1997, pp. 7-46.

[4] Curzon, P.: *The Formal Verification of the Fairisle ATM Switching Element;* Technical Reports No. 328 & No. 329, University of Cambridge, Computer Laboratory, March 1994.

[5] Edgcombe, K.: *The Qudos Quick Chip User Guide;* Qudos Limited.

[6] Gordon, M.; Melham, T.: *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic;* Cambridge, University Press, 1993.

[7] Langevin, M.; Tahar, S.; Zhou, Z.; Song, X.; Cerny E.: Behavior Verification of an ATM Switch Fabric using Implicit Abstract State Enumeration; Proc. *IEEE International Conference on Computer Design* (ICCD'96), Austin, Texas, USA, October 1996; pp. 20 - 26.

[8] Leslie, I. and McAuley, D.: Fairisle: An ATM Network for the Local Area; *ACM Communication Review,* Vol. 19, No. 4, September, 1991, pp. 327-336.

[9] Lu, J. and Tahar, S.: *On the Formal Verification and Reimplementation of an ATM Switch Fabric Using VIS;* Technical Report No. 401, Concordia University, Dept. of ECE, September 1997.

[10] McMillan, M.: *Symbolic Model Checking;* Kluwer Academic Publishers, Boston, Massachusetts, 1993.

[11] Tahar, S.; Zhou, Z.; Song, X.; Cerny, E.; Langevin, M.: Formal Verification of an ATM Switch Fabric using Multiway Decision Graphs; *Proc. IEEE Sixth Great Lakes* Symposium on VLSI (GLS-VLSI'96), Ames, Iowa, USA, March 1996; IEEE Computer Society Press, pp. 106-111.

[12] Brayton, R. et. al.: VIS: A System for Verification and Synthesis; Technical Report UCB/ERL M95, Electronics Research Laboratory, University of California, Berkeley, December 1995.