

Model Checking of the Fairisle ATM Switch Fabric using FormalCheck

Leila Barakatain and Sofiène Tahar

Technical Report

September 1999

Concordia University, ECE Dept., Montreal, Quebec, H3G 1M8 Canada

E-mail: {l_baraka, tahar}@ece.concordia.ca

Abstract. *In this paper we present our experience on model checking of an Asynchronous Transfer Mode (ATM) network switch fabric using the FormalCheck tool. The switch we considered is in use for real applications in the Cambridge Fairisle network. It is composed of four input/output port controllers and a switch fabric. For the verification in FormalCheck, we used the same Verilog code as in [1] with some modifications. We verified an abstracted (1-bit) model of the switch fabric (this model was already verified, using VIS), then verified a 4-bit model, and an 8-bit model of the switch fabric. We did not find any errors in our ATM switch fabric design.*

1 Introduction

Verification is increasingly becoming the bottleneck in the design flow of communication networks systems. Conventional simulation is very expensive in terms of time while exhaustive simulation is virtually impossible. Therefore, formal verification of digital systems is gaining interest, as the correctness of a formally verified design implicitly involves all possible input values. Although ATM is hailed as the most important communication mechanism in the foreseeable future, there is currently little experience on the application of formal verification to ATM network hardware.

In this paper, we present our experimental results on the model checking of an ATM switch fabric using FormalCheck [1]. The switch we investigated is a part of a network which carries real user data: the Fairisle ATM network [4], designed and in use at the Computer Laboratory of the University of Cambridge. This switch is composed of four input/output port controllers and a switch fabric.

We first developed a 1-bit abstracted model of the Fairisle ATM switch fabric in Verilog HDL and verified it successfully. Then we wrote the Verilog RTL descriptions for a 4-bit and the full 8-bit switch fabric. We verified these models, and no errors were found.

The rest of the paper is structured as following: In Section 2, we describe the structure and behavior of the Fairisle ATM switch. In Section 3, we introduce 7 properties of the switch fabric and the experimental results of model checking. In Section 4 we present a comparison between the experimental results obtained with VIS [2] and FormalCheck. We conclude the paper in Section 7.

2 The Fairisle ATM Switch

The Fairisle ATM switch consists of three types of components: input port controllers, output port controllers and a switch fabric (Figure 1). The input port controllers receive ATM cells from

transmission lines, store them in queues, dequeue them, append *fabric header* and *output header*, and then transfer the cells into the switch fabric. An ATM cell consists of 48 data bytes plus a 4-byte header. The input port controllers also receive acknowledgment signals from the fabric and decide whether to send new data, to retransmit previous cell, or to stop sending data. The switch fabric transfers data cells from input port controllers to the output port controllers and passes the acknowledgment signals from the output port controllers to the input port controllers according to the fabric header. If cells on common destination clash, it arbitrates using round-robin allocation. The output port controllers decide whether to transfer data to transmission lines or loop back data to the input port controllers according to the output header. They also detect errors in received cells, and send the acknowledgment signals to the switch fabric. 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*). It ensures that the port controllers inject data cells into the fabric synchronously so that the fabric headers arrive at the same time. In this paper we are concerned with the verification of the switch fabric only.

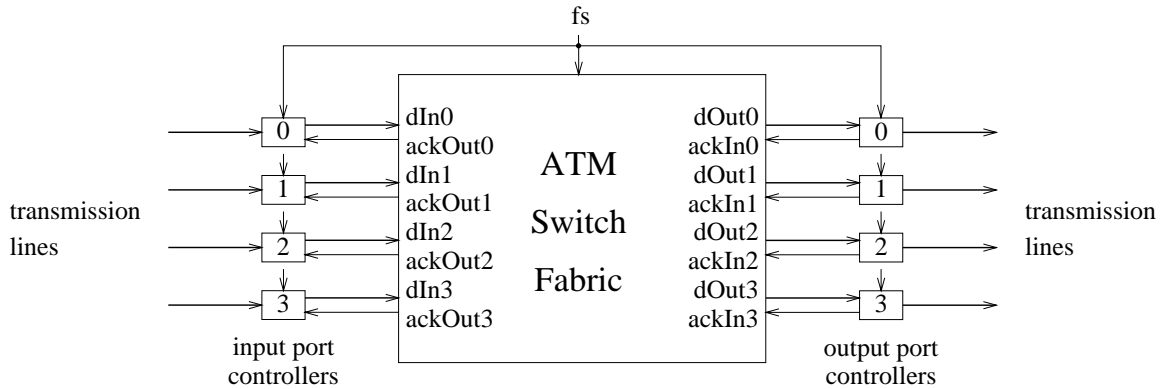


Figure 1: Structure of the Fairisle ATM switch

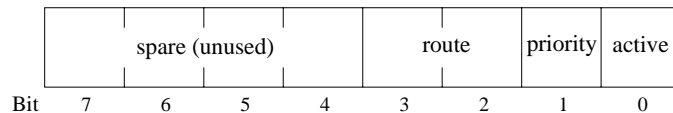


Figure 2: Fabric header of a Fairisle ATM cell

The port controllers synchronize incoming and outgoing data cells, appending control information in the front of the cells in a routing tag (Figure 2). This header is stripped off before the cell reaches the output stage of the fabric. The fabric switches ATM cells from the input ports to the output ports according to the fabric header. If different port controllers inject cells destined for the same output port (which is indicated by the route bits) into the fabric at the same time, then only one will succeed, and the others must re-try later. The priority bit in the fabric header is used for arbitration, and the high priority cells are given precedence. For those 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 fabric passes the acknowledgment from the requested output port to the successful input port, and does not forward the acknowledgment to unsuccessful input ports or forwards the negative acknowledgment when the output port controllers are running short of buffer space.

2.1 Switch Fabric Behavior

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 boundaries of separate cycles are determined by the *frameStart* signal. Whenever it goes high, a new cycle commences. The cells from all the input ports start when a particular bit (the active bit) of any input port goes high; the fabric does not know when this will happen. However, all the input port controllers must start sending cells at the same time within the frame. If no input port raises the active bit through the frame then the frame is inactive. Otherwise it is active. In order to initialize the fabric correctly for the forthcoming frame, the active bits must be low in the 2 cycles prior to the arrival of the *frameStart* signal. Because the decision is completed 3 clock cycles after the header time (arrival of routing tag), the fabric begins to send acknowledgment at least 3 clock cycles after that. In [6], the overall behavior of the switch fabric (including constraints from the port controllers) is expressed in form of one state machine composed of 14 states.

2.2 Switch Fabric Implementation

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 arbitration unit is composed of a timing unit, a decoder, a priority filter and a set of arbiters.

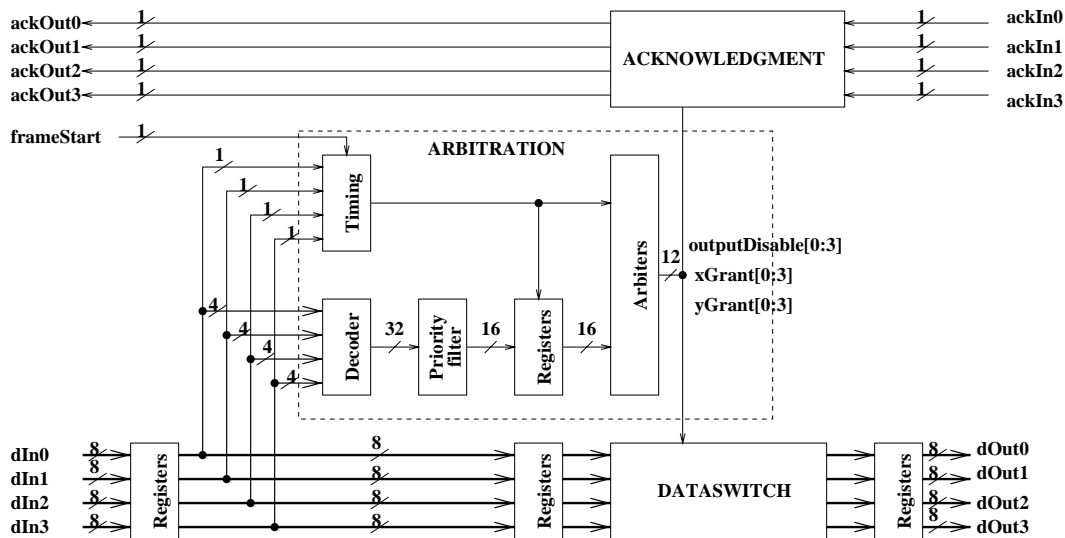


Figure 3: Fairisle switch fabric implementation

The decoder reads the fabric headers of the cells and decodes the port requests with low priority and those from inactive inputs, and passes the actual request situation for each output port to the arbiters. 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. The dataswitch switches data from input ports to expected output ports according to the signals xGrant[i], yGrant[i] and outputDisable[i]. The acknowledgment unit passes appropriate acknowledgment signals to input ports also according to these same signals.

3 Model Checking

In our design, we used the Verilog behavioral description of the switch fabric that was verified using VIS [5] as the base design for the abstracted (1-bit) fabric, and then we made some modifications to the code. After succeeding in verifying the abstracted model of the fabric, we modeled and verified a 4-bit and an 8-bit model of the ATM switch fabric.

In order to express explicit time points in certain properties, we need to represent them via explicit states in the corresponding properties. Therefore, an environment state machine was established, which imitates the behavior of the port controllers and also constraints the number of possible inputs to the switch fabric.

3.1 Environment for the Port Controllers

The switch fabric's interface with the port controllers consists of the signals *frameStart*, 32-bit data inputs, 32-bit data output, 4-bit acknowledgment inputs and 4-bit acknowledgment outputs (Figure 3). In our design, the port controllers are modeled as a finite state machine. Since the *frameStart* signal is cyclic every 64 clock cycles, the port controllers could be expressed as a 68-state environment state machine (Figure 4). This 68-state environment state machine is inspired from the work described in [6]. In Figure 4, there are 68 states enumerated by integers. 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. *fs*, *h* and *d* above the states mean that the *frameStart* signal, the routing tag (header) of an active cell and the data, respectively, are generated in that state. States 1 to 5 are related to the initialization of the fabric. States 6 to 68 represent the cyclic behavior of the fabric, where one cycle corresponds to one frame [6].

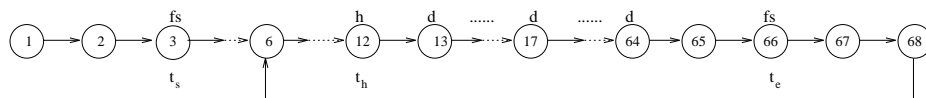


Figure 4: 68-state environment state machine

In this design, the states with the same behavior were combined to one state. To simplify the definition of properties and also to decrease the CPU time of the property checking; for instance, because states 13 to 64 of Figure 4 have the same behavior, which is that data are input to the fabric, they were combined to one state that is state S3 in Figure 5.

Figure 5 is the abstracted environment state machine and represents the behavior of the port controllers. t_s , t_h and t_e also correspond with states S0, S2 and S6, respectively. States S1 to S7 represent the cyclic behavior of the fabric, where one cycle corresponds to one frame. This environment state machine represents the main features of the port controllers and has the minimum states [5].

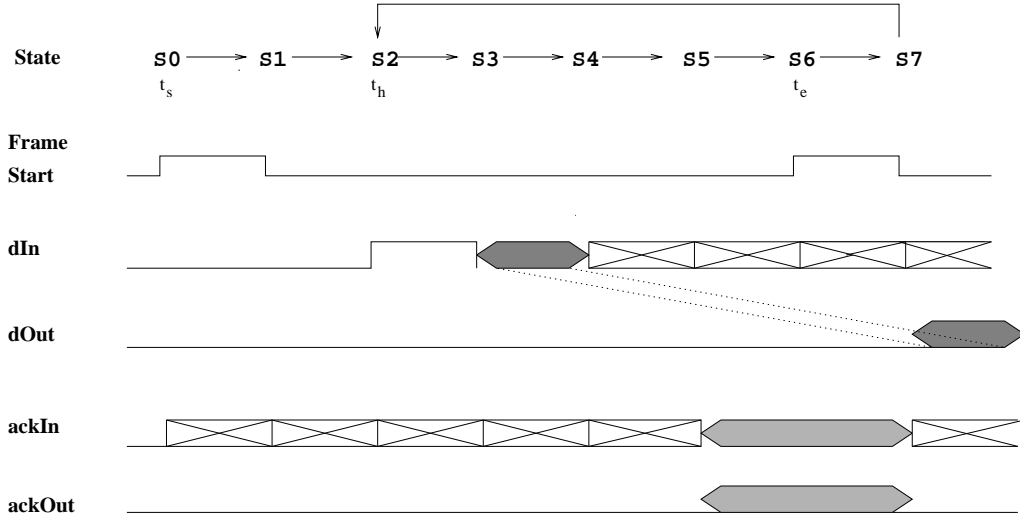


Figure 5: Abstracted environment state machine with related timing diagrams

3.2 Properties Description

We considered 7 properties of the fabric including liveness and safety properties, which are presented as following. The properties are expressed in both CTL expressions, and FormalCheck formulas. In the following CTL expressions, “ \neg ”, “ $\text{fi}\Rightarrow$ ”, “ \wedge ” and “ \vee ” denote logical “not”, “imply”, “and” and “or”, respectively. Also in the following FormalCheck properties, “ $\&\&$ ”, “ $\|$ ”, and “ $==$ ” denote logical “and”, “or”, and “equal”.

Note also in the FormalCheck properties that we make use of the “Fulfill Delay” and “Duration” options provided by the tool. Generally, the “Fulfilling Condition” is checked after the “Enabling Condition” is true. A delay can be added between the “Enabling Condition” and the checking of the “Fulfilling Condition”. This delay is specified as a integer that counts the occurrences of an event. This event is specified by the rising edge of “Clock” in our experiment. The verification window begins after the delay. The verification windows terminate after a given “Duration” or the “Discharging Condition” becomes true, whichever comes first. This duration is specified as an integer that counts the occurrences of an event, which in our case is the rising edge of “Clock”. In the following properties “Clock” and “Reset” signals were defined as default constraints in the FormalCheck project. “Reset” is used to initialize the registers (it starts with high for duration of 2, and then goes to low forever).

Property1: At state S2 (t_h), if input port 0 chooses output port 0 in the routing tag, potentially the data in input port 0 will be transferred to output port 0.

In CTL, this liveness property is expressed as following:

$$\text{AG } (dIn0[0] = 1 \wedge dIn0[2] = 0 \wedge dIn0[3] = 0 \wedge \text{state} = 2 \Rightarrow \text{EF } (dIn0^{S3} == dOut0));$$

where $dIn0^{S3}$ stores the value of $din0$ in state S3. Similarly, we could give other 15 liveness properties (one for each remaining 4x4 combination) to demonstrate that any input port that chooses any output port in the routing tag will potentially transfer data to that output port, but we do not express all these here.

In FormalCheck this liveness property is expressed as following:

```
Property: property1
Type: Eventually
After: environment.reset == 0 && environment.state == 2 &&
      environment.clock == rising && environment.d0[0] == 1 &&
      environment.d0[2] == 0 && environment.d0[3] == 0
Eventually: environment.dOut0 == environment.dIn0S3
Options:(None)
```

Next, we consider several safety properties. Safety properties checking is similar to a simulation of many cases at the same time. In following we present six example safety properties (Property 2 - Property 7) of the fabric along with their CTL expressions. Note that Properties 4, 5, 6 and 7 are similar to those described in [6].

Property 2: The arbitration component cannot make output port 0 and output port 1 connect to the same data input port at any time.

In CTL, this property is expressed as following:

```
AG ( ¬ (xGrant[0] == xGrant[1] ∧ yGrant[0] == yGrant[1] ∧ outputDisable[0] = 0 ∧
      outputDisable[1] = 0));
```

In FormalCheck this property is expressed as following:

```
Property2
Type: Never
Never: environment.reset == 0 &&
      environment.xGrant[0] == environment.xGrant[1] &&
      environment.yGrant[0] == environment.yGrant[1] &&
      environment.outputDisable[0] == 0 &&
      environment.outputDisable[1] == 0
Options:(None)
```

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

In CTL, this property is expressed as following:

```
AG ((state = 3 ∨ state = 4 ∨ state = 5 ∨ state = 6)
    ⇒ dOut0 = 0 ∧ dOut1 = 0 ∧ dOut2 = 0 ∧ dOut3 = 0);
```

In FormalCheck this property is expressed as following:

```
Property3
Type: Always
After: environment.state == 3 || environment.state == 4 ||
      environment.state == 5 || environment.state == 6
Always: environment.dOut0 == 0 && environment.dOut1 == 0 &&
      environment.dOut2 == 0 && environment.dOut3 == 0
```

Options: Fulfill Delay: 0 Duration: 1 count of
environment.clock == rising

Property 4: Except states S6 (i.e. except the time interval t_h+4 to t_e), the default value is put on the acknowledgment output ports.

In CTL, this property is expressed as following:

```
AG ((state = 1 ∨ state = 2 ∨ state = 3 ∨ state = 4 ∨ state = 5 ∨ state = 7)
    ⇒ ackOut0 = 0 ∧ ackOut1 = 0 ∧ ackOut2 = 0 ∧ ackOut3 = 0);
```

In FormalCheck this property is expressed as following:

Property: property4

Type: Always

```
After: environment.reset == 0 && (environment.state == 1 ||
    environment.state == 2 || environment.state == 3 ||
    environment.state == 4 || environment.state == 5 || environment.state == 7 ||)
```

```
Always: environment.ackOut0 == 0 && environment.ackOut1 == 0 &&
    environment.ackOut2 == 0 && environment.ackOut3 == 0
```

Options: Fulfill Delay: 0 Duration: 1 count of
environment.clock == rising

Property 5: In state S7 (i.e. from t_h+5 to t_e+1), if the input port 0 chooses output port 0 with the priority bit set in the header and no other input port has its priority bit set. The value on $dOut0$ will be $dIn0^{S3}$ which is the data input that is 4 clock cycles earlier than the data output $dOut0$.

In CTL, this property is expressed as following:

```
AG (dIn0[3:0] = 0011 ∧ dIn1[1] = 0 ∧ dIn2[1] = 0 ∧ dIn3[1] = 0 ∧ state = 2
    ⇒ AX AX AX AX AX ( dOut0 == dIn0S3 ));
```

In FormalCheck this property is expressed as following:

Property: property5

Type: Always

```
After: environment.d0 == 3 && environment.d1[1] == 0 &&
    environment.d2[1] == 0 && environment.d3[1] == 0 &&
    environment.state == 2 && environment.clock == rising
```

```
Always: environment.dOut0 == environment.dIn0S3
```

Options: Fulfill Delay: 4 Duration: 1 count of
environment.clock == rising

Property 6: In state S6 (i.e. from t_h+4 to t_e), if input port 0 chooses output port 0 with priority bit set in the routing tag, and no other input port has its priority bit set, the value on $ackOut0$ will be the input of $ackIn0$.

In CTL, this property is expressed as following:

```
AG (dIn0[3:0] = 0011 ∧ dIn1[1] = 0 ∧ dIn2[1] = 0 ∧ dIn3[3] = 0 ∧ state = 2
    ⇒ AX AX AX AX ( ackOut0 == ackIn0 ));
```

In FormalCheck this property is expressed as following:

```
Property: property6
Type: Always
After: environment.d0 == 3 && environment.d1[1] == 0 &&
      environment.d2[1] == 0 && environment.d3[1] == 0 &&
      environment.state == 2 && environment.clock == rising
Always: environment.ackOut0 == environment.ackIn0
Options: Fulfill Delay: 3   Duration: 1 count of
        environment.clock == rising
```

Property 7: In state $S7$ (i.e. from t_h+5 to t_e+1), if the input port 0 chooses output port 0 without the priority bit set in the header and no other input port has active bit set. The value on $dOut0$ will be $dIn0^{S3}$ which is the data input that is 4 clock cycles earlier than the data output $dOut0$.

In CTL, this property is expressed as following:

```
EG (state = 2  $\wedge$  dIn0[3:0] = 0001  $\wedge$  dIn1[0] = 0  $\wedge$  dIn2[0] = 0  $\wedge$  dIn3[0] = 0
     $\Rightarrow$  dOut0 = dIn0S3);
```

In FormalCheck this property is expressed as following:

```
Property: property7
Type: Eventually
After: environment.clock == rising && environment.d0 == 1 &&
      environment.d1[0] == 0 && environment.d2[0] == 0 &&
      environment.d3[0] == 0 && environment.state == 2
Eventually: environment.dOut0 == environment.dIn0S3
Options: (None)
```

3.3 Experimental Results

All verifications in this paper were executed on a SUN, Ultra 2 Model 2296, Sparc CPU (296MHz/1.1 GB). We first verified the abstracted model of the ATM switch fabric (1-bit model), and then designed a 4-bit model of the fabric and verified it. Finally an 8-bit model of the fabric was designed and verified. The experimental results are shown in Table 1, Table 2 and Table 3, respectively. These tables include the number of reached states, the number of states in the model, the average state coverage, the CPU time (real time) in seconds, and memory usage in megabytes.

As we can see from these tables by increasing the number of bits in the data path, the memory usage increases. In Properties 2, 4, and 6 we can see the number of state variables, and the number of reached states are the same. The reason for that is these properties do not depend on the datapath and therefore the number of state variables and reached states should be independent of the width of input/output data. Properties 1, 3, and 5 are dependent on datapath, as a result, by increasing the of input/output data width, the number of state variables and the number of reached states will increase as well.

Table 1: Verification results of property checking on 1-bit fabric using FormalCheck

Properties	States reached	State variables	State Var. Avg. coverage	Real time (Seconds)	Memory usage (MB)
Property 1	1.05e+06	63	100.00%	11	6.34
Property 2	4.20e+06	55	99.09%	10	6.34
Property 3	6.78e+07	84	98.81%	12	6.34
Property 4	7.23e+07	76	98.68%	14	6.34
Property 5	1.05e+06	64	98.44%	11	6.34
Property 6	7.29e+07	76	98.68%	14	6.34
Property 7	1.05e+06	63	100.00%	11	6.34

Table 2: Verification results of property checking on 4-bit fabric using FormalCheck

Properties	States reached	State variables	State Var. Avg. coverage	Real time (Seconds)	Memory usage (MB)
Property 1	1.94e+06	102	99.02%	20	6.77
Property 2	4.20e+06	56	99.11%	12	6.77
Property 3	1.08e+08	120	97.50%	17	6.78
Property 4	7.23e+07	76	98.68%	16	6.77
Property 5	1.94e+06	103	98.06%	14	6.78
Property 6	7.29e+07	76	98.68%	13	6.78
Property 7	1.94e+06	102	99.02%	14	6.77

Table 3: Verification results of property checking on 8-bit fabric using FormalCheck

Properties	States reached	State variables	State Var. Avg. coverage	Real time (Seconds)	Memory usage (MB)
Property 1	1.27e+11	187	99.47%	24	6.78
Property 2	4.20e+06	55	99.09%	15	6.79
Property 3	7.05e+12	200	98.50%	19	6.81
Property 4	7.23e+07	76	98.68%	18	6.81
Property 5	1.27e+11	188	98.94%	19	6.81
Property 6	7.34e+07	76	98.68%	16	6.81
Property 7	1.27e+11	187	99.47%	19	6.78

4 Comparing the verification results of VIS and FormalCheck tools

To have a fair comparison between FormalCheck and VIS tools, we tried to verify the properties defined in Section 3.2 for the 1-bit fabric model using both tools VIS and FormalCheck. The experimental results of this comparison is shown in Table 4.

FormalCheck, by default, uses 1-step reduction algorithm, which first performs a single reduction, and then verifies the property. As we can see in Table 4 the memory usage of the properties in FormalCheck has been less than VIS for all of the properties being checked. The CPU time usually depends on the amount of work being loaded on the CPU at the time of the verification. So by considering the CPU time we cannot have a fair comparison between the tools, but we can say in general, FormalCheck is faster than VIS. Note also that VIS was unable to check the 4-bit and 8-bit version of the Fairisle switch fabric [5].

Table 4: Results of verification of the 1-bit fabric using FormalCheck and VIS

Properties	Elapsed time using VIS (Sec.)	Real time using FormalCheck (Sec.)	Memory usage using VIS (MB)	Memory usage using FormalCheck(MB)
Property 1	3.2	11	8.2	6.34
Property 2	0.6	12	6.1	6.34
Property 3	67.9	12	9.6	6.34
Property 4	41.2	14	9.1	6.34
Property 5	2	11	8.2	6.34
Property 6	41.9	14	9.1	6.34
Property 7	1.7	11	8.2	6.34

5 Conclusions

In this study, we explored model checking for a real ATM switch and detected several design errors in it. The main contributions of this work are (1) the establishment of the abstracted model of the ATM switch fabric, (2) the definition of a set of typical properties to the fabric in FormalCheck, (3) the verification of original model of the fabric (8-bit), and (4) the comparison between two hardware verification tools, FormalCheck and VIS

The current application of FormalCheck (version 2.1.156) works very well with circuits of moderate size. FormalCheck provides various algorithms to perform verification, such as “Symbolic State Enumeration” (using ordered Binary Decision Diagrams or BDDs [3]), “Explicit State Enumeration”, and “Auto-Restrict” options [1]. To verify large circuits and to avoid the state space explosion there are several techniques to use: (1) choosing the suitable run option can reduce the run time when verifying large circuits, (2) using a suitable reduction method and reduction seed in FormalCheck, and (3) if these techniques fail, an abstracted model of the circuit can always be used.

One of the motivations of this work was to compare the verification of this switch fabric using FormalCheck with the verification of the same switch using VIS. Verification in FormalCheck has the advantage of having built-in reduction methods which makes the job faster and much easier. Another advantage of FormalCheck is that expressing the properties in it is very easy and we do not have to express the properties in CTL. Furthermore, one major advantage of FormalCheck is the built in simulator inside it which makes detecting errors inside the design much easier.

To verify a large size design, the verifier cannot always view the design as a black box, and so he/she must have a thorough knowledge of the design to check properties in model checking. Also formal verification should be used in the design flow as much as possible.

References

- [1] Bell Labs Design Automation, Lucent Technologies: *FormalCheck User's Guide*. V2.1, July 1998.
- [2] R. Brayton et. al. VIS: A System for Verification and Synthesis. Technical Report UCB/ERL M95, Electronics Research Laboratory, University of California, Berkeley, December 1995.
- [3] R. Bryant: Graph-Based Algorithms for Boolean Function Manipulation; *IEEE Transactions on Computers*, Vol.C-35, No. 8, August 1986, pp.677-691.
- [4] I. Leslie and D. McAuley: Fairisle: An ATM Network for the Local Area; *ACM Communication Review*, Vol. 19, No. 4, pp. 327-336, September, 1991.
- [5] J. Lu and S. Tahar: On the Verification and Reimplimentation of an ATM Switch Fabric Using VIS, Technical Report, Dept. of ECE, Concordia University, September 1997.
- [6] S. Tahar, X. Song, E. Cerny, Z. Zhou, M. Langevin, and O. Ait-Mohamed: Modeling and Verification of the Fairisle ATM Switch Fabric using MDGs; *IEEE Transactions on CAD of Integrated Circuits and Systems*, Vol. 18, No. 7, July 1999, pp. 956-972.