

# MODEL CHECKING OF THE FAIRISLE ATM SWITCH FABRIC USING FORMALCHECK

*Leila Barakatain and Sofène Tahar*

ECE Dept., Concordia University, Montreal, Quebec, H3G 1M8 Canada  
E-mail: {l\_baraka, tahar}@ece.concordia.ca

## ABSTRACT

In this paper we describe the 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. For the current verification in FormalCheck, we used the same Verilog HDL code as in [9] with some modifications. We specified and verified in FormalCheck a set of liveness and safety properties against several model sizes of the switch fabric. First, we verified an abstracted (1-bit) model of the switch fabric, which was already verified using VIS [9]. Then, we accomplished the verification of a 4-bit model, and the full 8-bit model. We furthermore provide a comparative study between the verification results in VIS and FormalCheck.

## 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 [7]. Although ATM (Asynchronous Transfer Mode) is hailed as the most important communication mechanism in the foreseeable future [6], there is currently little experience on the application of formal verification to ATM network hardware.

In this paper, we present our results on the model checking of an ATM switch fabric using FormalCheck [3]. The switch we investigated is a part of a network which carries real user data: the Fairisle ATM network [8], 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.

The Fairisle ATM switch fabric has been the subject of a number of related formal verification work using a variety of tools. Notably the work of Curzon [4] using the HOL theorem prover, Garcez [5] using the HSIS model checker, Tahar *et. al* [10] using the MDG equivalence and invariant

checkers, and more recently Lu *et. al* [9] using the VIS model checker.

In the work presented here, we used the same Verilog HDL code as in [9] (with some modifications) of a 1-bit abstracted model of the Fairisle ATM switch fabric. Then we wrote the Verilog RTL descriptions for a 4-bit and the full 8-bit switch fabric. We verified these models, and no design errors were found. For the verification in FormalCheck, we defined a set of typical liveness and safety properties which we checked on the Verilog models of the switch fabric. Furthermore, we conducted a comparison between the experimental results obtained on the model checking of the same switch fabric using the FormalCheck [3] and VIS [1] tools.

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 provide the properties descriptions and verification results in FormalCheck. In Section 4 we discuss a comparison between the experimental results obtained with VIS and FormalCheck. We conclude the paper in Section 5.

## 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.

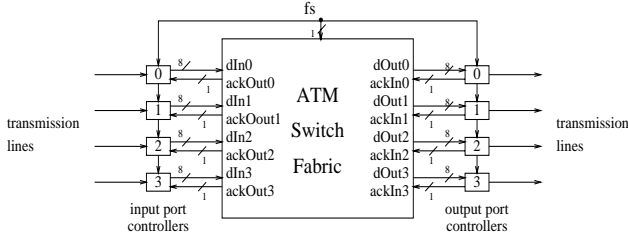


Figure 1: The Fairisle ATM switch

The port controllers synchronize incoming and outgoing data cells, appending control information in the front of the cells in a header (Figure 2). This latter 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.

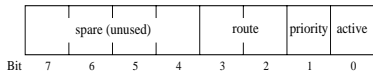


Figure 2: Header of a Fairisle ATM cell

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 send-

ing 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 two cycles prior to the arrival of the *frameStart* signal. Because the decision is completed three clock cycles after the header time (arrival of header), the fabric begins to send acknowledgment at least three clock cycles after that.

## 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.

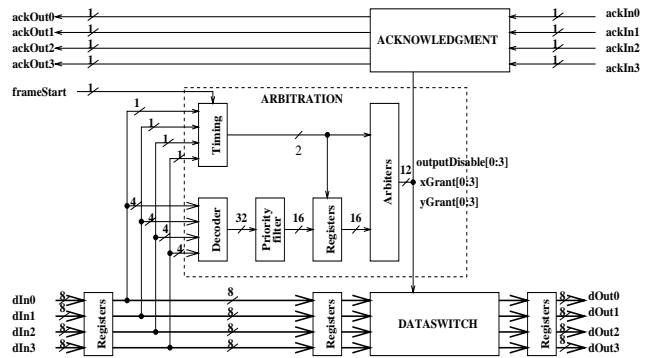


Figure 3: Fairisle switch fabric structure

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

We specified and verified in FormalCheck a set of liveness and safety properties against several model sizes of the switch fabric. First, we modeled in Verilog and verified in FormalCheck an abstracted (1-bit) model of the switch fabric, which was already verified using VIS [9]. Afterwards, we modeled and verified a 4-bit and an 8-bit models 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 [10]. 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 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 [10].

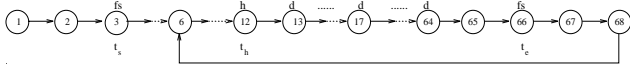


Figure 4: 68-state environment state machine

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. In this figure,  $t_s$ ,  $t_h$  and  $t_e$  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 [9].

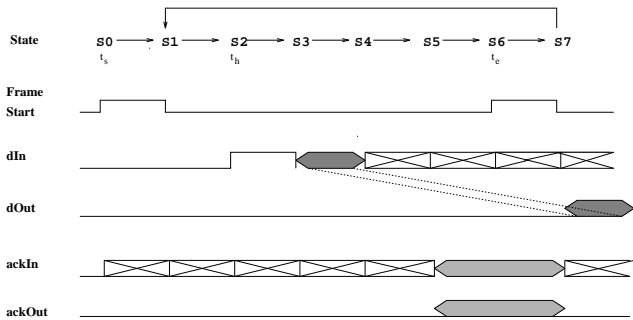


Figure 5: Abstracted environment state machine with related timing diagrams

### 3.2 Properties Description

We considered a set of seven properties of the fabric including liveness and safety properties, which we describe within FormalCheck. These properties are similar to the CTL properties proposed in [9]. In the following FormalCheck properties, “&&”, “||”, and “==” denote logical “and”, “or”, and “equal”, respectively.

For properties description in FormalCheck, we make use of the “Fulfill Delay” and “Duration” options provided by the tool [3]. 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 an 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 two cycles, and then goes to low forever).

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

In FormalCheck this liveness property is expressed as follows:

```
Property: property1
Type: Eventually
After: reset == 0 && state == 2 &&
      clock == rising && dIn0[0] == 1 &&
      dIn0[2] == 0 && dIn0[3] == 0
Eventually: dOut0 == dIn0_S3
Options: (None)
```

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 header will potentially transfer data to that output port, but we do not express all these here.

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. Note that Properties 4, 5, 6 and 7 are similar to those described in [10].

**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 FormalCheck this property is expressed as following:

```
Property: property2
Type: Never
Never: reset == 0 &&
      xGrant[0] == xGrant[1] &&
      yGrant[0] == yGrant[1] &&
      outputDisable[0] == 0 &&
      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 FormalCheck this property is expressed as following:

```
Property: property3
Type: Always
After: state == 3 || state == 4 ||
      state == 5 || state == 6
Always: dOut0 == 0 && dOut1 == 0 &&
      dOut2 == 0 && dOut3 == 0
Options: Fulfill Delay: 0
      Duration: 1 count of 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 FormalCheck this property is expressed as following:

```
Property: property4
Type: Always
After: reset == 0 && (state == 1 ||
      state == 2 || state == 3 ||
      state == 4 || state == 5 ||
      state == 7)
Always: ackOut0 == 0 && ackOut1 == 0 &&
      ackOut2 == 0 && ackOut3 == 0
Options: Fulfill Delay: 0
      Duration: 1 count of 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 FormalCheck this property is expressed as following:

```
Property: property5
Type: Always
After: dIn0 == 3 && dIn1[1] == 0 &&
      dIn2[1] == 0 && dIn3[1] == 0 &&
      state == 2 && clock == rising
Always: dOut0 == dIn0_S3
Options: Fulfill Delay: 4
      Duration: 1 count of 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 header, and no other input port has its priority bit set, the value on  $ackOut0$  will be the input of  $ackIn0$ .

In FormalCheck this property is expressed as following:

```
Property: property6
Type: Always
After: d0 == 3 && d1[1] == 0 &&
      d2[1] == 0 && d3[1] == 0 &&
      state == 2 && clock == rising
Always: ackOut0 == ackIn0
Options: Fulfill Delay: 3
      Duration: 1 count of 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 FormalCheck this property is expressed as following:

```
Property: property7
Type: Eventually
After: clock == rising && d0 == 1 &&
      dIn1[0] == 0 && dIn2[0] == 0 &&
      dIn3[0] == 0 && state == 2
Eventually: dOut0 == dIn0_S3
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 Tables 1, 2 and 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, increasing the number of bits in the datapath, increases the memory usage. In Properties 2, 4, and 6 we can also see the number of state variables, and the number of reached states are the same. The reason for this is that these properties do not depend on the datapath and therefore the number of state variables and reached states is independent of the width of input/output data. Properties 1, 3, and 5 are dependent on datapath, as a result, by increasing the input/output data width, the number of state variables and the number of reached states will increase as well.

**Table 1: Model checking results on the 1-bit fabric model using FormalCheck**

Properties	States reached	State variables	State var. 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: Model checking results on the 4-bit fabric model using FormalCheck**

Properties	States reached	State variables	State var. 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: Model checking results on the 8-bit fabric model using FormalCheck**

Properties	States reached	State variables	State var. 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. COMPARISON WITH THE VERIFICATION IN VIS

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 ver-

ifies the property [3]. As we can see in Table 4, the memory usage of the properties in FormalCheck has been less than in VIS for all of the properties 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 models of the Fairisle switch fabric [9].

**Table 4: Comparative verification results on the 1-bit fabric model 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 investigated the model checking of a real ATM device, the Fairisle switch fabric, using the commercial tool, FormalCheck. The main contributions of this work are: (1) the establishment of different Verilog models of the ATM switch fabric; (2) the definition in FormalCheck of a set of typical properties on the fabric; (3) the verification of the original (8-bit) model of the fabric; and (4) a comparison between two hardware verification tools, FormalCheck and VIS.

The current application of FormalCheck 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 [2]), “Explicit State Enumeration”, and “Auto-Restrict” options [3]. To verify large circuits and to avoid 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 needs to be developed.

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 formulas. 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.

## 6. REFERENCES

- [1] 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.
- [2] R. Bryant, “Graph-Based Algorithms for Boolean Function Manipulation,” *IEEE Transactions on Computers*, Vol.C-35, No. 8, pp. 677-691, August 1986.
- [3] Cadence, *Formal Verification Using Affirma FormalCheck*, Manual, Version 2.3, October 1999
- [4] P. Curzon, “The Formal Verification of the Fairisle ATM Switching Element,” Technical Reports 329, University of Cambridge, Computer Laboratory, March 1994.
- [5] E. Garcez, “The Verification of an ATM Switching Fabric using the HSIS Tool,” Technical Report, WSI-95-13, Tübingen University, Germany, 1995.
- [6] D. Ginsburg, *ATM Solutions for Enterprise Internetworking*, Addison Wesley, 1996
- [7] C. Kern and M. Greenstreet, “Formal Verification in Hardware Design: A Survey,” *ACM Transactions on Design Automation of E. Systems*, Vol. 4, pp. 123-193, April 1999.
- [8] I. Leslie and D. McAuley, “Fairisle: An ATM Network for the Local Area,” *ACM Communication Review*, Vol. 19, No. 4, pp. 327-336, September, 1999.
- [9] J. Lu and S. Tahar, “Practical Approaches to the Automatic Verification of an ATM Switch Fabric using VIS,” *Proc. IEEE 8th Great Lakes Symposium on VLSI*, Lafayette, Louisiana, USA, pp. 368-373, February 1998.
- [10] S. Tahar et. al, “Modeling and Verification of the Fairisle ATM Switch Fabric using MDGs,” *IEEE Transactions on CAD of Integrated Circuits and Systems*, Vol. 18, No. 7, pp. 956-972, July 1999.