On the Verification and Reimplementation of an ATM Switch Fabric Using VIS

Jianping Lu and Sofiène Tahar

Technical Report No. 401

September 1997

Electrical & Computer Engineering Department, Concordia University
1455 de Maisonneuve Blvd. W., Montreal, Quebec, H3H 1M8 Canada
Email: {jianping, tahar}@ece.concordia.ca

Abstract. *In this paper we present our results and methods on formally verifying the implementation of an Asynchronous Transfer Mode (ATM) network switching fabric using the Verification Interacting with Synthesis (VIS) tool. We also present our experience of using VIS in the reimplementation of this fabric. The design we considered is in use for real applications in the Cambridge Fairisle network. We produced the behavioral and structural descriptions[1] of the switch fabric at different levels of abstraction. We then performed equivalence checking between the structural and behavioral descriptions of each submodule of the implementation hierarchy. Using several 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. In addition, we succeeded in discovering several injected design errors using both property checking and equivalence checking. The experience on the verification and reimplementation of the fabric demonstrates that VIS can interact with Synopsys and Verilog-XL simulator easily and hence can be used in the digital design flow to make the design more reliable.*

Key Words: ATM, VIS, Verilog-XL, Simulation, Synthesis, Property Checking, Sequential Equivalence Checking, Cascade Property Division, Parallel Property Division, Latch Reduction.

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.

Because ATM (Asynchronous Transfer Mode) is a network technology for addressing the variety of needs for new high-speed, high-bandwidth applications, ATM 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 [5] formally verified the 4 by 4 fabric of the Cambridge Fairisle switch using the HOL theorem prover [7] and Tahar et al. [13][8] verified the same switch fabric in an automatic fashion using the MDG (Multiway Decision Graphs) tools [4] by property checking and equivalence checking. Another case of formal verification of an ATM circuit was made by Chen et al. [3] at

1. In this project,we use the notion of *behavioral* and *structural* descriptions to mean *RTL* and *netlist* descriptions, respectively.

Fujitsu Digital Technology Ltd, where the authors identified a design error in an ATM circuit using the tool SMV (Symbolic Model Verifier) [10] 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 [14]. We also report our experience of applying VIS in the reimplementation of this ATM network component.

The device we investigated is a part of a network which carries real user data: the Fairisle ATM network [9], designed and in use at the Computer Laboratory of the University of Cambridge. The component we considered is the Fairisle 4 by 4 switching fabric which performs the actual switching of data cells and forms the heart of the ATM Fairisle communication network. VIS, which is developed at the University of California, Berkeley, is a tool integrating the verification, simulation and synthesis of finite-state hardware systems. It uses a Verilog front-end and supports fair CTL model checking, language emptiness checking, combinational and sequential equivalence checking, cycle-based simulation, and hierarchical synthesis.

We wrote the Verilog HDL behavioral description of the switch fabric at different levels of abstraction, and translated the original Qudos HDL [6] structural description of the switch fabric to Verilog. We performed equivalence checking between the behavioral and structural descriptions of submodules and achieved property checking on abstracted models of the switch using several approaches. In addition, we succeeded in detecting several injected design errors using both property checking and equivalence checking. Furthermore, we point out to the potential application of VIS in the digital design flow while performing the reimplementation of this ATM switch fabric. Although VIS supports cycle-based simulation and interacts with SIS (Sequential Interactive Synthesis) [11] to do the logic synthesis, the VIS simulator and SIS are not as reliable and popular as Cadence's Verilog-XL and Synopsys's Design Compiler respectively. Therefore, we used Verilog-XL and Synopsys in the reimplementation of this switch fabric, and applied VIS within the design flow. The experience of the reimplementation of this ATM switch fabric demonstrates that VIS can be easily used for on-the-fly verification and hence can assist some commercial designs efficiently.

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 show the methodology and results of the property checking and errors detection using VIS. In Section 4, we introduce the results of equivalence checking and error detection. The reimplementation of this ATM switch fabric is presented in Section 5 and Section 6 concludes the paper.

2 The ATM Switch Fabric

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

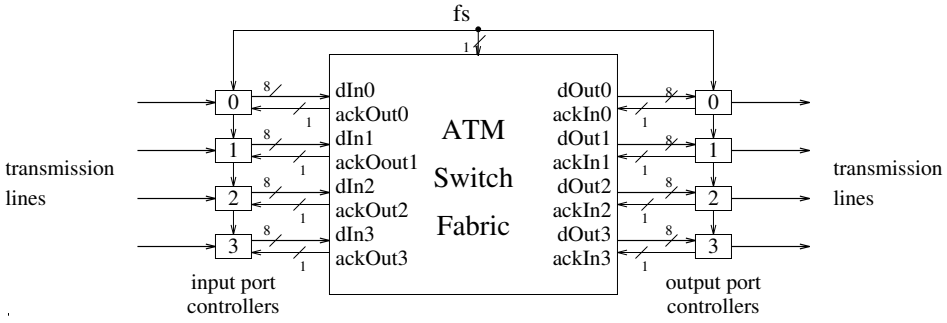

Figure 1: The structure of the Fairisle ATM switch

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

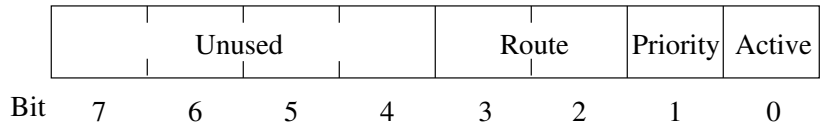

Figure 2: The routing tag of a Fairisle ATM cell

This tag is stripped off before the cell reaches the output stage of the fabric. The fabric switches cells, consisting of a fixed number of data bytes, from the input ports to the output ports according to the routing tag. 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 routing tag 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. 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 routing tags arrive at the same time. In this paper, we are concerned with the verification of the switch fabric only.

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 [8], 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. The decoder reads the routing tags 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 the input ports according to these same signals.

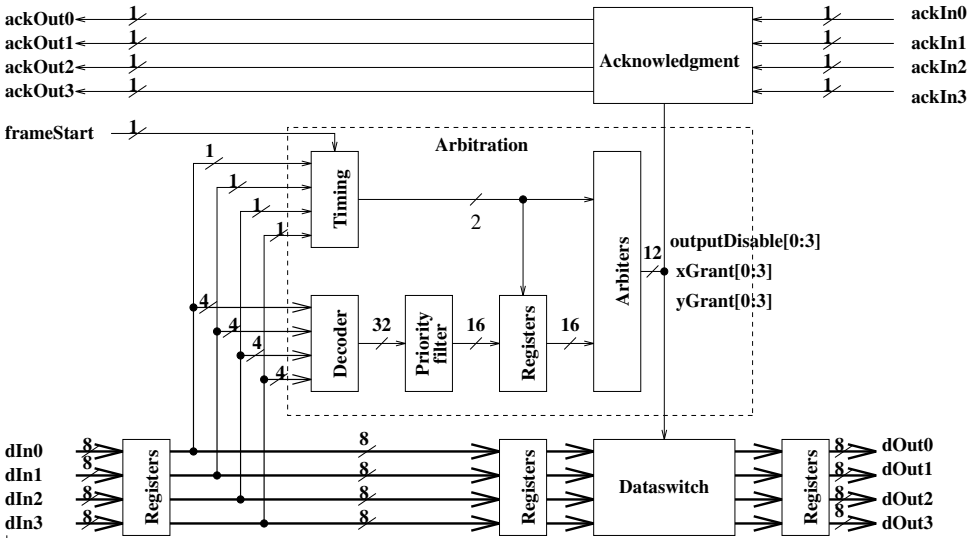

Figure 3: Fairisle switch fabric implementation

3 Property Checking

State space explosion is a well-known problem in FSM-based verification approaches. In hardware designs, the state space increases exponentially with the number of latches of a design. Because model checking algorithms are based on state space exploration, their efficacy is also limited by this phenomenon. Although the number of states that can be handled has been increased dramatically since the introduction of BDDs [2] 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.

While using VIS for property checking, properties have to be expressed in CTL. However, in order to give an explicit value for an input signal in a CTL expression, we have to declare the signal (variable) 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 of 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 approach, we modeled the port controllers as a finite state machine. Since the frameStart signal is cyclic every 64 clock cycles, we could express the port controllers as a 68-state environment state machine (Figure 4). This 68-state environment state machine is inspired from the work described in [13]. 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 [13].

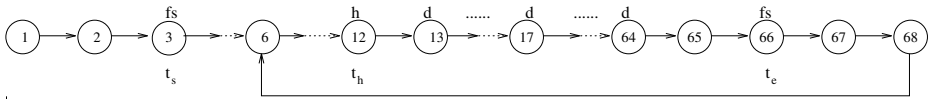

Figure 4: 68-state environment state machine

While trying property checking using this 68-state environment state machine, we noticed that it makes the properties hard to express in CTL and also increases the CPU time of property checking. Thus we combined the states with the same behavior to one state; for instance, because states 13 to 64 of Figure 4 have the same behavior, which is that data are input to the fabric, we combined them to one state that is state S3 in Figure 5.

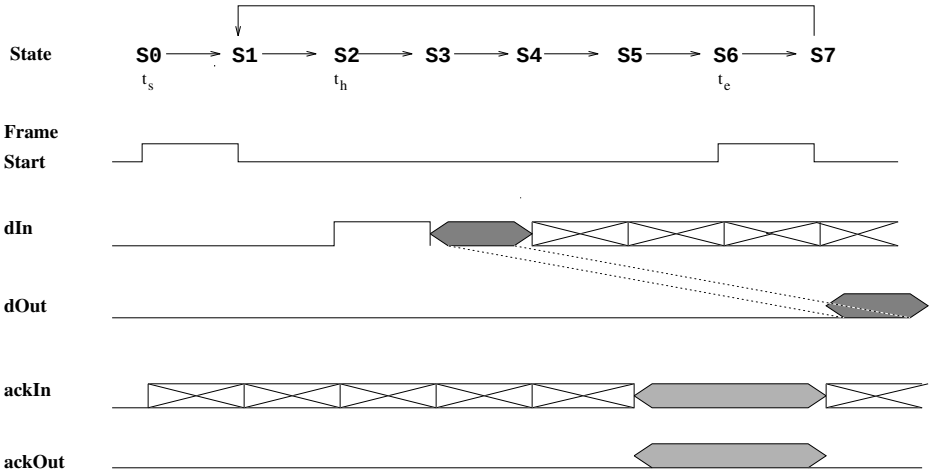

Figure 5: Abstracted environment state machine with related timing diagrams

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 two main features of the port controllers, namely:

1. *To initialize the fabric, the active bits must be low for 2 clock cycles prior to the frameStart signal arriving.*
2. *The acknowledgment signals from output port controllers are forwarded by the fabric at least 3 clock cycles after the header arrival time.*

Since we fixed the state S2 as header arrival time (t_h), we give the following additional constraint which makes the environment state machine feasible.

3. *If the active bit of a input port in state S2 is zero, the data input of the same input port in state S3 is also zero.*

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 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 expression 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. 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.

Next, we consider several safety properties. Safety properties checking is similar to a bisimulation 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 [13].

Property 2: The data bytes in a cell are transferred (from t_h+5 to t_e+1, i.e. state S7) from input port 0 to output port 0 sequentially with 4 clock cycle delay.

```
EG (state = S7 -> dOut0 = dIn0^S3);
```

Property 3: The arbitration component cannot make output port 0 and output port 1 connect to the same data input port at any time.

```
AG ( ! (xGrant[0] == xGrant[1] * yGrant[0] == yGrant[1] * outputDisable[0] = 0
* outputDisable[1] = 0));
```

Property 4: From state S3 (t_h+1) to S6 (t_h+4), 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 5: Except states S5 and S6 (i.e. except the time interval t_h+3 to t_e), the default value is put on the acknowledgment output ports.

```
AG ((state = S1 + state = S7 + state = S2 + state = S3 + state = S4) -> ackOut0
= 0 * ackOut1 = 0 * ackOut2 = 0 * ackOut3 = 0);
```

Property 6: 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*.

```
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 7: In state S5 (i.e. from t_h+3 to t_e-1), 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*.

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

Property 8: 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*.

```
EG (state = S2 * dIn0[3:0] = 0001 * dIn1[0] = 0 * dIn2[0] = 0 * dIn3[0] = 0 ->
dOut0 = dIn0^S3);
```

3.3 Abstracted fabric

We did not succeed in using the original fabric to check the properties due to state space explosion, and we also failed checking a 4-bit version of the whole fabric. To cope with the state space explosion problem, we often simplify the design to some abstracted models. Fortunately, most industrial designs are done in a modular and regular fashion, easing a representation of the design by an abstracted model. The fabric is also modular, as described in Section 2, it consists of arbitration, acknowledgment and dataswitch units. At first, we reduced the datapath of the dataswitch unit from 8 bits to 4 bits, the property checking still consume too much CPU time. We therefore reduced the datapath further 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 6.

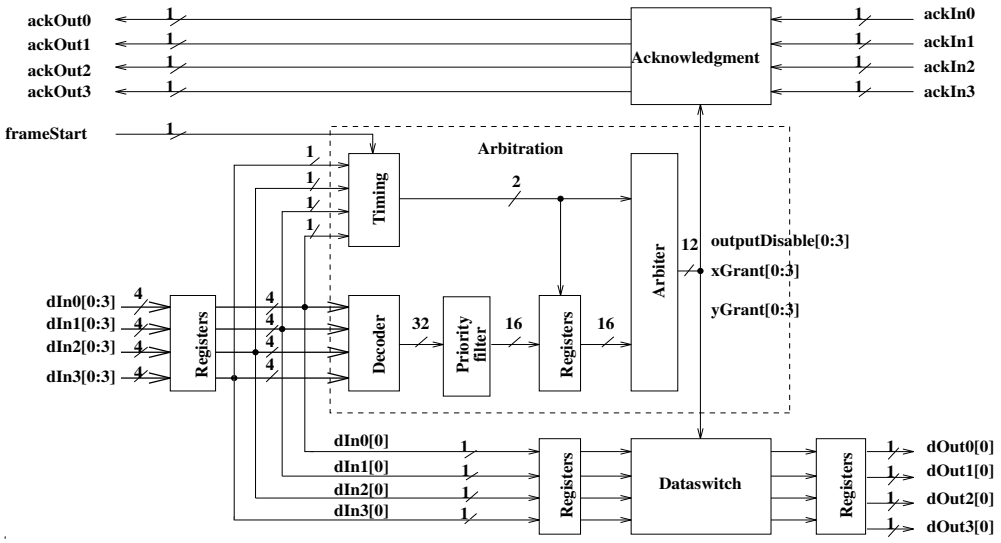

Figure 6: The abstracted switch fabric

Based on this abstracted model, we checked the above seven properties which were done on SUN Sparc 20 workstation (55MHz/256MB). (All the results in this paper are done in this workstation). The CPU time (elapse time), memory usage and nodes allocated of every property checking are shown in Table 1.

Table 1: Property checking on the abstracted fabric

Properties	CPU time (seconds)	Memory usage (MB)	Nodes allocated
Property 1	3933.9	40.3	84,199,139
Property 2	4550.7	4.3	90,371,031
Property 3	14.8	2.8	368,749
Property 4	3593.4	32.4	93,073,140
Property 5	833.0	4.5	28,560,871
Property 6	3679.7	40.9	79,687,784
Property 7	414.8	5.3	4,180,124
Property 8	1037.9	11.6	29,755,252

All the properties in Table 1 were checked using the sift algorithm [14] which is a dynamic ordering algorithm. Dynamic ordering provides an optimized order that will drastically decrease the memory usage, nodes allocated, and hence decrease CPU time. We choose property 2 as an example to see the difference between the property checking with and without dynamic ordering (see Table 2).

Table 2: Dynamic ordering in property checking

Property 2	CPU time (seconds)	Memory usage (MB)	Nodes allocated
with dynamic ordering	4550.7	4.3	90,371,031
without dynamic ordering	55125.7	51.2	302,307,020

3.4 Discussion on 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. Here, we discuss several approaches we adopted to speed-up checking these properties.

Some properties are impossible to be checked since they take too much time and are memory consuming even if we use the abstracted fabric and use dynamic ordering. So we tried to divide a property to several sub-properties, and checked each sub-property separately. If all the sub-properties pass the model checking, then the property passes the model checking too. We call this *Property Division*. We propose two approaches to divide a property. One is *Cascade Property Division*, and the other is *Parallel Property Division*.

3.4.1 Cascade property division

Cascade property division is to divide a property into several sequentially related sub-properties. If each sub-property passes model checking, then the property passes model checking. We use property 7 to explain this approach. The original CTL expression of property 7 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);
```

Like property checking in the abstracted fabric, we need build an environment state machine for each sub-property. For sub-property1, every input signal (variable) in its CTL expression is already in the environment state machine of the abstracted fabric, so we can use that environment state machine directly. But 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 (variables) *xGrant[i], yGrant[i]* and *outputDisable[i]* are given.

Table 3 gives the comparison between the property checking with cascade property division and the property checking without it. The CPU time for checking property 7 is enhanced by 41 times by cascade property division.

Table 3: Cascade property division in property checking

Property 7		CPU time (seconds)	Memory usage (MB)	Nodes allocated
without cascade property division		414.8	5.3	4,180,124
with cascade property division	sub-property1	5.1	1.9	109,438
	sub-property2	4.9	1.7	78,743
	Total	10	-	-

3.4.2 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, and checks every sub-property by an abstracted model that is stripped from a design regularly. We use property 4 to illustrate this approach.

The original CTL expression of property 4 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 a correspondent abstracted fabric unit from the abstracted fabric. Figure 6 describes the structure of this model. To verify sub-property 1, 2, 3, and 4, we used fabric unit 0, 1, 2 and 3, respectively. In Figure 6, $ackOuti^j$ denotes the acknowledgment signal on input port *i* of fabric unit *j*, so that the acknowledgment signal on input port *i (ackOuti)* represents the disjunction of the signals $ackOuti^0$, $ackOuti^1$, $ackOuti^2$ and $ackOuti^3$. Table 4 gives a

comparison between property checking with parallel division and property checking without it for property 4. From this table, the CPU time for property checking has been enhanced by 73 times.

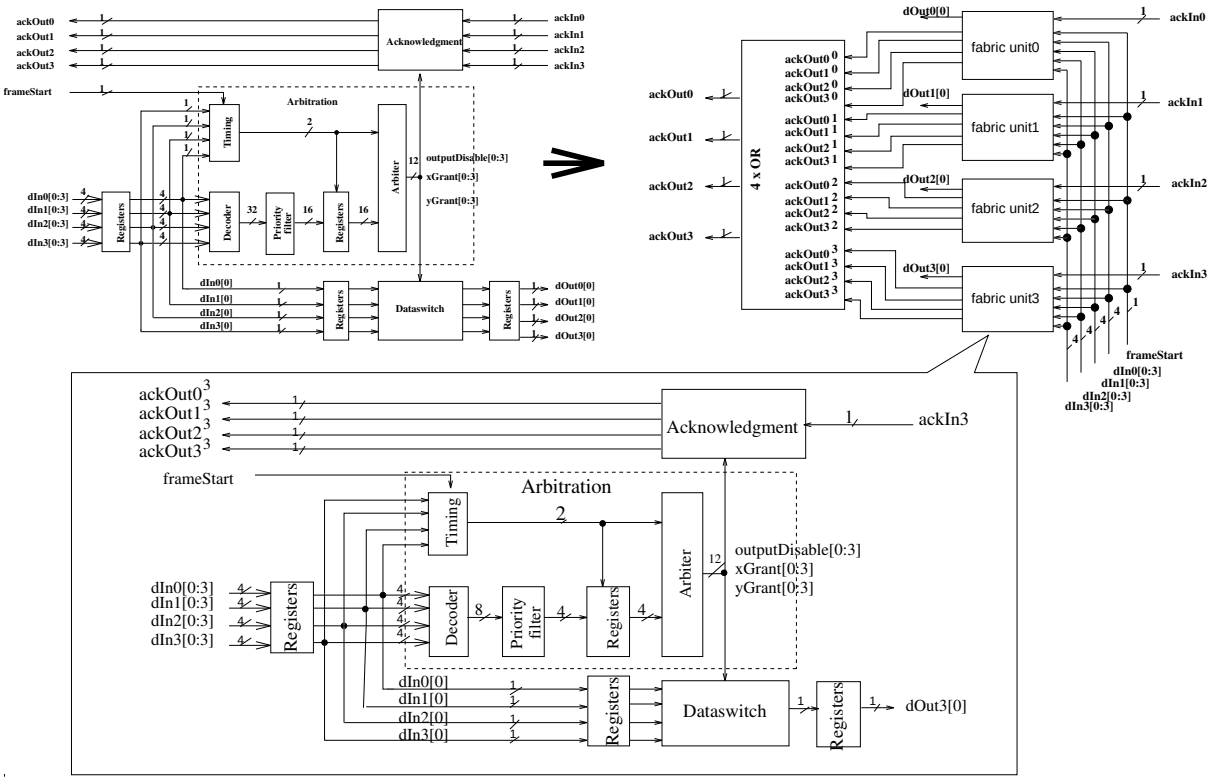

Figure 7: The abstracted fabric units

Table 4: Cascade property division in property checking

Property 4		CPU time (seconds)	Memory usage (MB)	Nodes allocated
without cascade property division		3593.4	32.4	93,073,140
with cascade 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	-	-

3.4.3 Latches Reduction

In a digital design, latches introduce states, so latches have a very obvious influence on the speed of property checking. Reducing the number of latches in a model will greatly speed-up a property checking. Table 5 collects the number of latches among the original fabric, the abstracted fabric and one abstracted fabric unit (all including some latches used for the environment state machine).

Table 5: The number of latches among different models

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

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 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 to do a property checking within an acceptable time in VIS. This result will guide us to build abstracted models.

Many designs use latches to pause data for 1 clock cycle in its primary inputs and outputs. Since these latches are directly connected to input and output ports, 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 6 shows that the CPU time of checking property 2 has been enhanced by nearly 100 times by using latch reduction. In this example, the data output latches that are used to delay output data for 1 clock cycle were ignored in the case of "abstracted fabric with latches reduction". When we ignore latches used to pause input or output data for some clock cycles, the timing behavior of a design will be changed, so we must do some corresponding update of a property to match the model with latch reduction. We use property 2 as an example to illustrate this. The original property 2 is *"The data bytes in a cell are transferred from input port 0 to output port 0 sequentially with **4 clock cycles** delay"*. Since the data output latches are ignored, the updated property 2 should be *"The data bytes in a cell are transferred from input port 0 to output port 0 sequentially with **3 clock cycles** delay"*.

Table 6: Latches reduction in property checking

Property 2	CPU time (seconds)	Memory usage (MB)	Nodes allocated (millions)
Abstracted fabric without latch reduction	4550.7	4.32	90,371,031
Abstracted fabric with latch reduction	23.1	3.15	235,004

3.5 Concluding results on property checking

All property checking results in Table 1 were obtained by using the abstracted fabric and dynamic ordering, and some results are not satisfactory since they take a long CPU time. By using cascade property division, serial property division and latch reduction, we got satisfactory results (see Table 7). For the properties listed in this paper, one of these enhancement approaches was enough for the property checking; for more complicated properties, we may combine several approaches at the same time.

Table 7: Summary of enhanced property checking of the fabric

Properties	Property 1	Property 2	Property 3	Property 4	Property 5	Property 6	Property 7	Property 8
CPU time (seconds)	27.8	46.1	14.8	48.9	72.5	34.1	10.0	46.1
Enhancement approach	latch reduction	latch reduction	-	parallel division	parallel division	latch reduction	serial division	latch reduction

3.6 Error detection in property checking

No errors were discovered in the above property checking. 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. (4)

We used erroneously the same select signal to control output port 0 and 1. (5) We reduced a set of delay registers for all the data input ports. These five errors were detected by property checking and VIS generated counterexamples that exhibit the incorrect behavior of the corresponding signals. Experimental results are reported in Table 10, where the CPU time includes the time for property checking and counterexample generation.

Table 8: Error detection in property checking

Experiments	Properties used for error detection	CPU time (seconds)	Memory usage (MB)	Nodes allocated
Error 1	Property 4	82.5	3.5	1,408,477
Error 2	Property 6	49.3	2.4	250,666
Error 3	Property 7	15.4	1.1	85,238
Error 4	Property 3	33.9	3.5	199,347
Error 5	Property 8	16.9	1.1	53,825

4 Equivalence Checking

Besides property checking, VIS supports combinational and sequential equivalence checking of two circuits. Since we did post-design verification of the switch fabric, 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 behavior specification. If both descriptions are equivalent, the correction of the fabric is proved. We first provided a complete 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 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. This advantage will be illustrated in more details in next section.

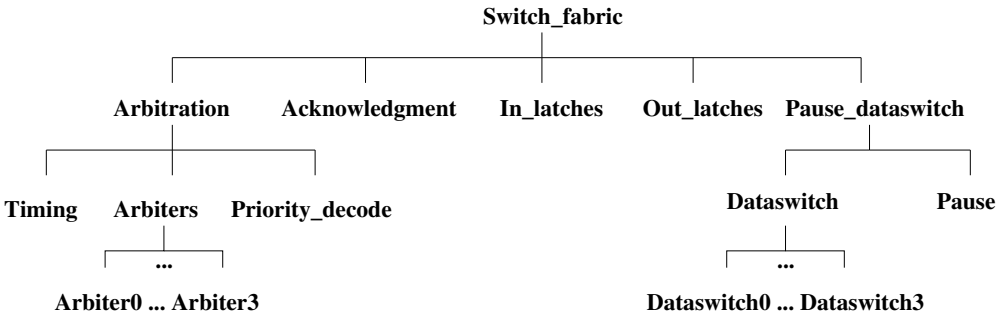

Figure 8: The modular structure of the switch fabric

Figure 8 represents the hierarchical structure of the fabric for which we provided behavioral descriptions. For illustration purpose, we use the timing module to demonstrate how the equivalence checking works.

4.1 The Timing module: An example

The timing module determines when the arbitration unit is triggered. This module can be described as a finite state machine shown in Figure 8. It specifies the following behavior: The *routeEnable* signal is normally low. After the *frameStart* signal it waits until any of the active bits (*anyActive* signal) goes high, indicating the start of a packet. On the next cycle *routeEnable* goes high for one cycle, returning to low until the next frame.

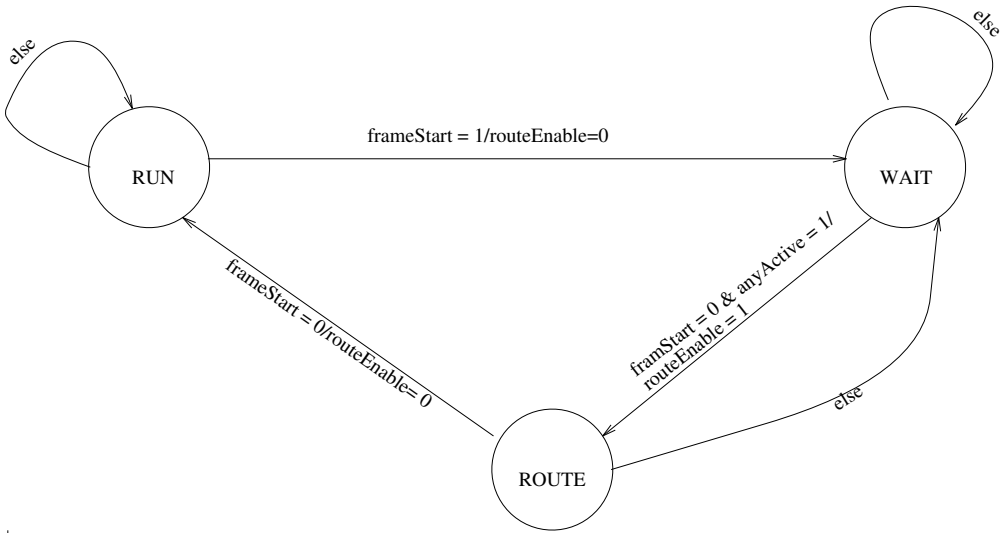

Figure 9: State transitions of the timing module

According to Figure 8, we wrote the following Verilog behavioral description for the timing module:

```
typedef enum {RUN, WAIT, ROUTE} timing_state;
module TIMING(frameStart,clock,active0,active1,active2,active3,routeEnable);
input frameStart, clock;
input active0, active1, active2, active3;
output routeEnable;
timing_state reg state;
wire anyActive;
assign anyActive = active0 ||  active1 || active2 || active3;
assign routeEnable = (state == ROUTE) ;
initial state = RUN;
always @(posedge clock)  begin case (state)
   RUN: if ( frameStart == 1)  state = WAIT;
   WAIT: if ( ( frameStart == 0) && (anyActive == 1)) state = ROUTE;
   ROUTE: begin if ( frameStart == 0) state = RUN;
                else state = WAIT; end
endcase end
endmodule
```

On the other hand, the Verilog translation of the Qudos structural description of the timing module looks as follows:

```
module TIMING(frameStart,clock,active0,active1,active2,active3,routeEnable);
input frameStart, clock;
input active0, active1, active2, active3;
output routeEnable;
wire xBar, dx, dy, yterm, anyActive, frameStartBar;
reg routeEnable, y;
initial begin routeEnable = 0; y = 0; end
not InvX (xBar, routeEnable);
or OrAnyActive (anyActive, active0, active1, active2, active3);
not InvFS (frameStartBar, frameStart);
and AnyTerm (yterm, xBar, y);
and AndDx (dx, xBar, y, anyActive, frameStartBar);
or OrDy (dy, yterm, frameStart);
always @(posedge clock)
begin routeEnable = dx; y = dy; end
endmodule
```

The equivalence checking between the above structural and behavioral descriptions was done automatically in VIS. We have been able to prove that the two networks (descriptions) are sequentially equivalent.

4.2 Summary of results on equivalence checking

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 checked 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 10). We failed to verify in VIS the modules *Dataswitch, Pause_dataswitch* and *Switch_fabric*. Table 10 gives the verification CPU time and number of latches for the modules verified using equivalence checking. Because of a software bug in the VIS tool, we were unable to obtain any statistics on memory usage and number of nodes allocated during the equivalence checking [12].

Table 9: Equivalence checking of submodules

Component	CPU time (seconds)	Memory usage (MB)	Nodes allocated	Number 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

Besides dynamic ordering, and in order to further enhance the equivalence checking, we tried to build an environment state machine to restrict the behavior of input variables. We also used different partition and image computation methods, but all these methods did not speed-up the equivalence checking dramatically.

4.3 Analysis on sequential equivalence checking

Since the modules *In_latches, Out_latches* and *Pause* are pure 32-bit latches, their sequential equivalence checking did not consume much CPU time although each of them has 32 latches. But generally the more latches a module has, the more expensive is the sequential equivalence checking. When the number of latches in one module reaches a certain value, the CPU time of the sequential equivalence checking rises a lot even if a circuit with few latches is added to this module. The following experiment shows this feature.

The module *Dataswitch_i* consists of four DMUX units. Table 10 compares the CPU time of equivalence checking among four similar modules with one, two, three and four *DMUX* units, respectively. In Table 10, the *Dataswitch_i* module (including four *DMUX* units) has only 4 latches more than the module with three *DMUX* units, but the CPU time of equivalence checking increased from 9.7 seconds to 1855.8 seconds. Therefore, it is almost impossible to check equivalence for the modules *Dataswitch*, *Pause_dataswitch and Switch_ fabric* even if each submodule is checked properly. In addition, in order to check the sequential equivalence of a module, we should divide this module into some submodules carefully to make each submodule easy to check. Through more experiments, we found that sequential equivalence checking could be done with acceptable time if the number of latches in a model is less than 20.

Table 10: Equivalence checking among modules with different DMUX units

Components	CPU time (seconds)	Memory usage (MB)	Nodes allocated	Number of latches
Module with one DMUX	0.7	?	?	4
Module with two DMUX	6.8	?	?	8
Module with three DMUX	9.7	?	?	12
Dataswitch_i (with four DMUX)	1855.8	?	?	16

4.4 Error detection in equivalence checking

No errors were discovered after equivalence checking on the implementation of each submodule. For experimental purposes, however, we detected the same five errors described in Section 3.6 by equivalence checking. VIS generated counterexamples that exhibit the incorrect behavior of the corresponding signals. Experimental results are reported in Table 10.

Table 11: Error detection in equivalence checking of submodules

Experiments	Affected submodules	CPU time (seconds)	Memory usage (MB)	Nodes allocated
Error 1	Arbiters	20.6	?	?
Error 2	Priority_decode	24.0	?	?
Error 3	Acknowledgment	1.7	?	?
Error 4	Priority_decode	54.5	?	?
Error 5	In_latches	0.7	?	?

5 Formal Verification in Digital Design

Although formal verification is gaining more and more interest by industry than before, its application, however, is not yet well established. In this section, we introduce our experience of using VIS in the reimplementation of the Fairisle switch fabric.

We began our reimplementation from the behavioral description of each submodule as described in Section 4. Because of the limitation of the current formal verification, we also used Verilog-XL for simulation. Since the synthesis tool SIS that can interact with VIS is not as popular as Synopsys, we used Synopsys Design Compiler for synthesis instead of SIS.

After synthesizing the behavioral description in Synopsys, we got the Verilog structural description (referred to as Synopsys-Verilog in Figure 10). But the format of the structural description could not be used directly in VIS or XL-simulator, and also the primitive modules of Synopsys are different from those of VIS and XL-simulator. We wrote a program in AWK [1] to automatically translate the Synopsys-Verilog format to a Verilog format that is accepted by VIS and XL-simulator (referred to as VIS/XL-Verilog in Figure 10). On the other hand, we wrote a file including the primitive modules of Synopsys so that we could interact VIS with Synopsys and XL simulator directly.

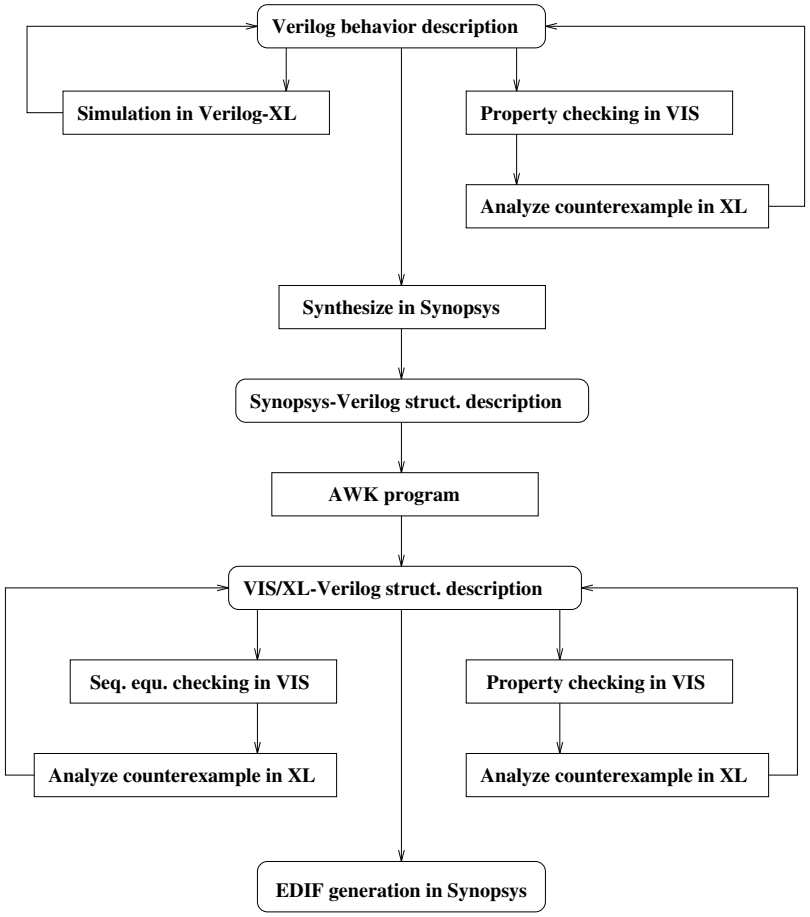

Figure 10: Digital design flow using VIS

Figure 10 represents the digital design flow we used in the reimplementation of the fabric. In addition to simulation and synthesis which are often used in industry, we did equivalence checking

between the behavioral description and the synthesized structural description of each submodule. We also verified some properties that are difficult to be checked by simulation. 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 have been able to convert the counterexamples into Verilog-XL and analyze them graphically.

VIS-Verilog has two special features which are nondeterminism and symbolic variables. The nondeterminism is normally used to build abstracted model, so we do not need to use it in a specific design. Because the symbolic variables are convenient for the definition of state variables, we also can substitute them by integer variables or other variable types. There are some trivial differences between the syntaxes of VIS-Verilog and that of XL-simulator, but we can interact VIS and XL simulator without any change if we only use the common expressions (syntax) that are accepted by both tools.

6 Conclusions

In this study, we have explored the formal verification for a real ATM switch fabric, and also reported our experience on reimplementing the fabric using VIS with Verilog-XL and Synopsys. The main contribution of our work is the development of several methods to avoid the state space explosion in the verification of the switch fabric. We also presented practical ways on how VIS can interact with XL-simulator and Synopsys in the reimplementation of the fabric.

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 methods such as property division and latch reduction which made VIS succeeding in checking some properties on large circuits.

One of the motivations of this work was to compare the verification of this switch fabric using VIS with the verification of the same switch using other tools. As we mentioned in Section 1, the formal verification of this fabric has been done in HOL by Curzon [5] and in MDG by Tahar et. al [13] [8]. Compared with the formal verification in HOL, the verification in VIS has the advantage that the verification jobs are done automatically, but has the disadvantage that one needs to build the abstracted models to verify the properties due to the problem of state space explosion. With the MDG tools, however, the automatic sequential equivalence of the whole fabric was possible thanks to the use of abstract sorts and uninterpreted functions. On the other hand, the MDG tools currently lack the ability to check liveness properties. Furthermore, one major practical advantage of VIS is its Verilog front-end which enables the interaction with commercial tools such as Synopsys, Cadence, etc.

Human time is a very important factor in formal verification. In this study, we spent a lot of time developing the techniques to accelerate both equivalence and property checking and also did a lot of experiments learning the limitations of VIS. Assuming a verifier who uses all the techniques described in this paper, he/she will spend about 20 days to verify this switch fabric as shown in Table 10. The structural descriptions are adapted directly from the Qudos HDL, and this will take 3 person days. The behavioral descriptions are more difficult because the verifier must understand the design thoroughly, this will take around 10 person days. Three person days will be spent on the simulation of the behavioral and structural descriptions including writing testbench files. Equivalence checking of submodules will take one day because it is easier to check errors in small

circuits. Three days will be taken to check properties, where most of the time will be spent abstracting the structures. Assuming a designer has the behavioral and structural description and simulation completed during the design phase, he/she will spend only four days for the formal verification as shown in Table 10.

Table 12: Summary of human time taken

Verification phases	Time taken
Structural description	3
Behavioral description	10
Simulation	3
Equivalence checking	1
Property checking	3
Total	20

Top-down design has become more and more popular in industry. It is possible for circuits of moderate size (less than 20 latches) to check equivalence between behavioral and synthesized structural description. In addition, property checking can handle large-sized circuits by modularized circuits and abstracted models. The most important aspect of formal verification is that it can generate counterexamples that are very helpful to detect errors in complicated sequential circuits. Our experience on reimplementing the fabric demonstrates that formal verification is practical and convenient to be used in a real design and will make the designer feel more confident of his/her design.

As we stated earlier, in order to verify a large-sized design by 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 to check 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.

Acknowledgments

We would like to thank the members of the VIS group Yuji Kukimoto, Rajeev Kumar Ranjan and Fabio Somenzi from University of California, Berkeley, and University of Colorado for their support in using the VIS tool.

References

[1] Aho, A. V. *The AWK programming language,* Addison-Wesley Pub. Co., 1988.

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

[3] 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.

[4] 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.

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

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

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

[8] 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; IEEE Computer Society Press, pp. 20 - 26.

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

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

[11] SIS: http: *//www-cad.eecs.berkeley.edu/Respep/Research/sis/abstract.html.*

[12] Somenzi, F.: personal communication, September 1997.

[13] 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.

[14] VIS: *http://www-cad.eecs.berkeley.edu/Respep/Research/vis*