The Impact of Design Changes on Verification Using MDGs

M. Hasan Zobair, Sofiène Tahar and Paul Curzon§

Electrical & Computer Engineering Department, Concordia University, Montreal, Canada
Email: {mh_zobai, tahar}@ece.concordia.ca

§School of Computing Science, Middlesex University, London, U.K.
Email: p.curzon@mdx.ac.uk

Technical Report

October 1999

Abstract. *In this report, we investigate the impact of design changes on formal verification using MDG (Multiway Decision Graphs) tools. In particular, we would like to determine whether the design changes that make verification by interactive theorem proving simpler, also make verification by automated decision diagram approach simpler as well. The design we consider is the Fairisle 4 by 4 switch fabric which is in use for real applications in the Cambridge ATM Fairisle network. A major consideration during the design change decisions should not compromise other design goals such as performance and functionality. The specification and verification obtained in MDG demonstrated the expected positive impact of these design changes.*

1. Introduction

As communication networks become all pervasive, the consequences of errors in the design or implementation of network components become an increasingly important area. The validation of network components is at best difficult. Simulation cannot uncover all errors in an implementation because only a small fraction of all possible cases can be considered. Formal verification is a different technique that can alleviate this problem. Because the correctness of a formally verified design implicitly involves all cases regardless of the input values [8].

Asynchronous Transfer Mode (ATM) [7] is being hailed as the solution to many communication problems. In essence, it consists of sending data over a packet-switched network using virtual circuits and short fixed size packets known as cells. It is a flexible technology and is being adopted by both the computer and telecommunication industries in local and wide area networks in response to changing communication demands. It has been adopted as the most important transfer mode of the foreseeable future. However, it represents a large paradigm shift in communications and there is currently a little experience from which to derive confidence of correct behavior. An ATM network design is thus a timely application for verification research. There have been several projects on the verification of ATM switches. For instance, the formal verification of the Cambridge Fairisle switch fabric had been done by Curzon [3] using the HOL theorem prover. Tahar *et al.* [13] verified the same switch in an automatic fashion using the MDG (Multiway Decision Graphs) tools by property checking and equivalence checking. Lu *et al.* [10] also formally verified this same ATM switch fabric using VIS. Chen *et al.* [2] at Fujitsu Digital Technology, formally verified an ATM circuit using SMV. By using a combination of theorem proving and model checking Rajan *et al.* [11] discovered bugs in a high-level ATM model that was presumed correct during simulation.

In this report, we investigate whether the formal verification task of an ATM design can be simplified by making necessary design changes: that is whether a notion of "Design for Verifiability", similar to that for testability, is of practical interest. Curzon [4][5] introduced this idea in the context of interactive proof. By using the HOL theorem prover [6], he suggested that the cost of verification in terms of time can be reduced by making appropriate design changes. In this work, we investigate whether the same design changes also reduce the verification cost while using the MDG (Multiway Decision Graphs) tools [1].

Our investigation involved the verification of an existing hardware design which was designed 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 [9]. The MDG tools, which were developed at the University of Montreal, are based on a new class of decision graphs. These decision graphs subsume Reduced Ordered Binary Decision Diagrams (ROBDD) while accommodating abstract sorts and uninterpreted function symbols. While verifying the original description of the switch fabric using the HOL theorem prover, which we refer to here as the *Original* Switch Fabric, Curzon *et al.* [5] noted the factors that were increasing the verification cost in terms of time. It became obvious that particular aspects of the behavioral specification were lengthening the verification time by significant amounts. Moreover, by changing the behavior of the switch fabric, which is controlled by the environment of the switch fabric, i.e., port controllers, the problems would have been removed. While changing the actual design, Curzon *et al.* were concerned that such changes should not affect the performance or functionality of the device. We will refer to the modified design which includes the suggested design changes during the interactive proof, as the "*Cleaned*" version of the Original Fairisle 4 by 4 switch fabric. The new design incorporated the following changes without any significant loss of functionality:

- The header arrive at least 5 cycles after the frame start signal.
- The header and frame start must not occur together.
- Internal delays were added to the datapaths so that no extra cell byte is lost.
- Minor changes to the internal timing of the data switch so it reads two grant signals at a more sensible time.

In this new work, we repeat Curzon's reverification of the *Cleaned* Fabric using MDG tools. In addition to the suggested modification, we changed the original environment state machine which was used in *Original* Fairisle 4 by 4 switch fabric verification using MDG tools [13]. The new verification of the *Cleaned* version of the design has been performed by property checking and equivalence checking provided by the MDG tools. We made this change to keep the external constraints on modules simple which in turn improves the design for verification.

The outline of this paper is as follows: In Section 2, we describe the *Original* version of the Fairisle switch fabric in terms of behavioral and structural description. In Section 3, we describe the changes to the fabric that were suggested by the verification attempt in theorem prover. In Section 4, we describe the verification of the *Cleaned* version in MDG. In Section 5, we are comparing and contrasting different aspects of the *Cleaned* and *Original* versions of the switch fabric and Section 6 concludes the paper.

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*, as shown in Figure 1. It switches ATM cells from the input ports to the output ports. A cell consists of a *header* (one-byte tag containing routing information as shown in Figure 2) and a fixed number of data bytes. The port controllers synchronize incoming data cells, append headers in the front of the cells, and send them to the fabric. The fabric waits for cells to arrive, strips off the tags, arbitrates between cells destined to the same output port, sends successful cells to the appropriate output port controllers, and passes acknowledgments from the output port controllers to the input port controllers. If different port controllers inject cells destined for the same output port controller into the fabric at the same time then only one will succeed and the others must retry later. The header also includes priority information (*priority* bit) that is used by the fabric for arbitration which takes place in two stages.

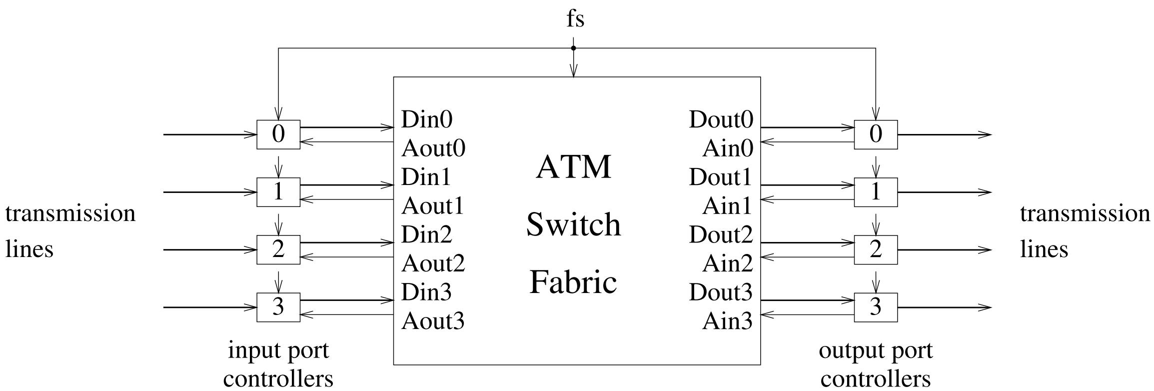

Figure 1: The structure of the Fairisle ATM switch

High priority cells are given precedence before the other cells. The choice within both priorities is made on a round-robin basis. The input controllers are informed of whether their cells were successful using acknowledgment signals. The fabric sends a negative acknowledgment to the unsuccessful input ports, but passes the acknowledgment from the requested output port controllers to the successful input port. The port controllers and the switch fabric all use the same clock, hence bytes are received synchronously on all links. They also use a higher-level cell frame clock—the *frame start* (f_s) signal (see Figure 1). It ensures that the port controllers inject data cells into the fabric synchronously so that the headers arrive at the same time. In this paper, we are concerned with the verification of the *switch fabric*.

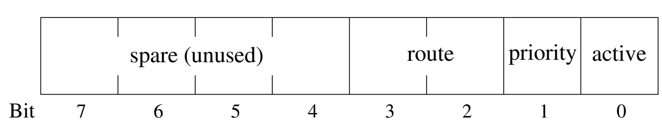

Figure 2: The routing tag of a Fairisle ATM cell

3. MDG Modeling of the *Cleaned* Fabric

Inspired by [4][5] and the verification of the *Original* design of the Fairisle switch fabric using the MDG tools [13], we derived an MDG description of the *Cleaned* version of the switch fabric. The behavioral specification of the switch fabric is represented in the form of an Abstract State Machine (ASM). It reflects the complete behavior of the fabric under the assumptions that the environment maintains some constraints on the arrival time of the frame start signal and the cell headers. We investigated the modified behavior of the switch fabric under the control of the environment and describe our *Cleaned* version of the switch fabric in the following three sub-sections. In Section 3.1, we describe the assumptions about the environment of the fabric, i.e., port controller in the form of a finite state machine. In Section 3.2, we describe the behavioral specification of the switch fabric under the assumptions described in Section 3.1. In Section 3.3 we describe the implementation of the switch fabric.

3.1 Environment for the port controllers

The set of timing-diagrams in Figure 3 represents the expected behavior of the *Cleaned* version of the switch fabric during an active frame. Based on this and similar sets of timing-diagrams we derived our environment state machine which controls the changed input-output behavior of the switch fabric. After the *frame start* (at time t_s), the switch waits for the headers to appear on the input lines *Din*. After the arrival of the headers (at time t_h), an arbitration between the inputs clashing for the same output is done in at most 2 cycles. The successful cells (bytes that follow the headers on *Din*) are transferred to the corresponding output port (*Dout*) with a delay of 5 cycles while acknowledgment (*Ain*) starting at time t_h+3 traverse in the opposite direction without any synchronous delay. Note that the last 5 cycles t_e-1 to t_e-5 of a frame do not accept any data.

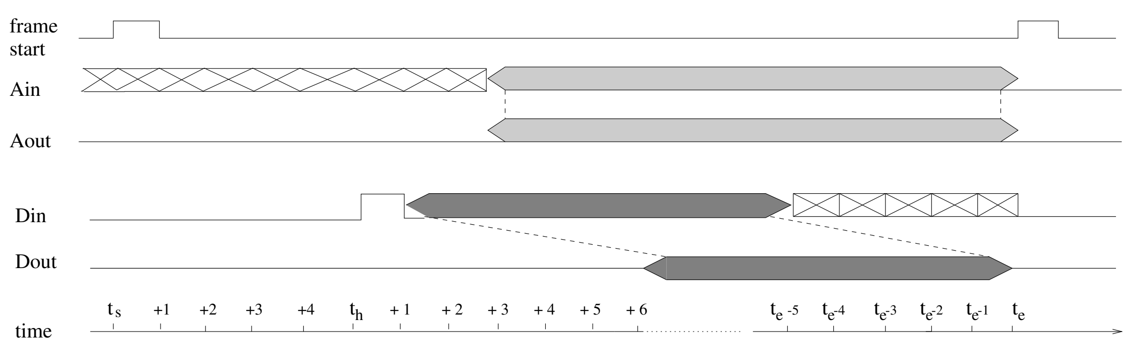

Figure 3: Timing diagram behavior during an active frame

The finite state machine having 64 states in Figure 4 was inspired by these timing-diagrams. We modified the original environment state machine as given in the verification design documentation of the *Original* version of the Fairisle switch [13] to comply with the modifications suggested by Curzon *et al*. [5]. In the rest of the section, we describe the four modified assumptions about the environment of the switch fabric and the reasons of the modifications from its antecedent [13].

1. At start up (t_0) the frame starts without any delay, i.e., $t_s = t_0$. In the current usages of the fabric, the *frame start* is delayed by at least 2 cycles before being asserted. We brought this modification because it was lengthening the verification time by a significant amount.

2. The header arrive (t_h) at least 5 cycles after the *frame start*, i.e., $t_h \geq t_s + 5$. In the current usages of the fabric, the header arrives at least 3 cycles after the *frame start* (t_s), i.e., $t_h \geq t_s + 3$. We

made this change, so that this 4 by 4 switch fabric can be used as a module for the front elements of a 16 by 16 switch fabric (a 16 by 16 switch fabric consists of eight 4 by 4 switch fabric — 4 front elements and 4 back elements) without any modification. In the case of the *Original* version the header arrives at 8 clock cycles after the *frame start* [5]. Under this assumption the required clock cycles to process a cell having length 52 bytes were 68 clock cycles (refer to Figure 5). The fabric was designed before the port controllers, so it was not clear what the necessary delays required by the port controllers would be. In our *Cleaned* version, we need only 64 clock cycles to process a cell having the same length which is similar to the frame size. Because of this assumption, the arrival and leaving times of the cells will be less.

3. The current design of the fabric allows cells, and thus headers, to arrive at any time within a frame provided that they all arrive together. This leads to a complex assumption about a frame structure which will change form for different modules. This aspect of the design needs to ensure that the cells do not arrive close to the frame start. This feature of the design caused much confusion in the implementation of both the port controllers and the larger fabric. To overcome this complexity, there should be a fixed time spacing between the arrival of the header and the frame start signal. If the header arrives at a known time after the frame start, the timing circuitry would be simpler and make it easier to verify. By considering the above reasons, the header may arrive at least 5 cycles before the next frame start, i.e., $t_e > t_h + 5$.

4. After the active bit goes high a f*rame start* signal cannot arrive until the data is processed. Hence, the next *frame start* cannot arrive before at least 11 cycles from the current *frame start* (t_s). This is to maintain the consistency of the timing interval between the arrival of the header and the *frame start* signal.

Based on the above four assumptions our environment state machine will be a finite state machine having 64 states as shown in Figure 4.

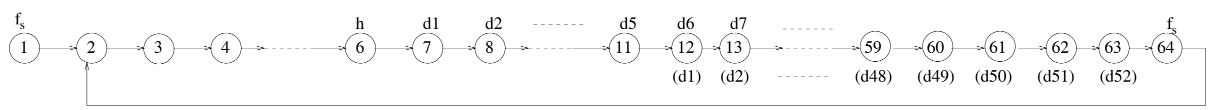

Figure 4: The new environment state machine

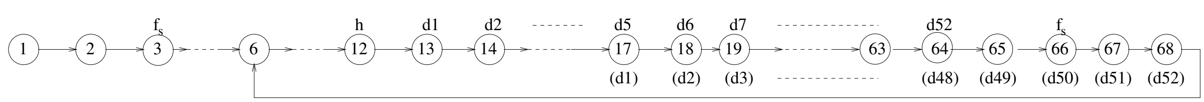

Figure 5: The original environment state machine

In Figure 4, there are 64 states enumerated by integers (using MDG-HDL, state variables can be described as a concrete variables of sort [1, 2, .., 64]). State transitions are denoted by arrows. In analogy to the environment states machine [13], we used f_s, h and d_i to denote the *frame start* signal, the header of an active cell and the data processing in that state, respectively. The notion of *(d_i)* in Figure 4 and Figure 5 indicating that data are switched to the output port in that state. The *frame start* signal may arrive in state 1 or 64. Header and next frame start signal may arrive in states 6 or 64, respectively. If the cell length is 52 bytes, the *frame start* signal will be cyclic in every 64 clock cycles, i.e., the frame size is 64 cycles. States 2 to 64 represent the cyclic behavior of the fabric. Between states 12 to 63, the remaining 52 bytes of the cell following the header arrive at the output port.

3.2 Behavioral Specification of the Switch Fabric

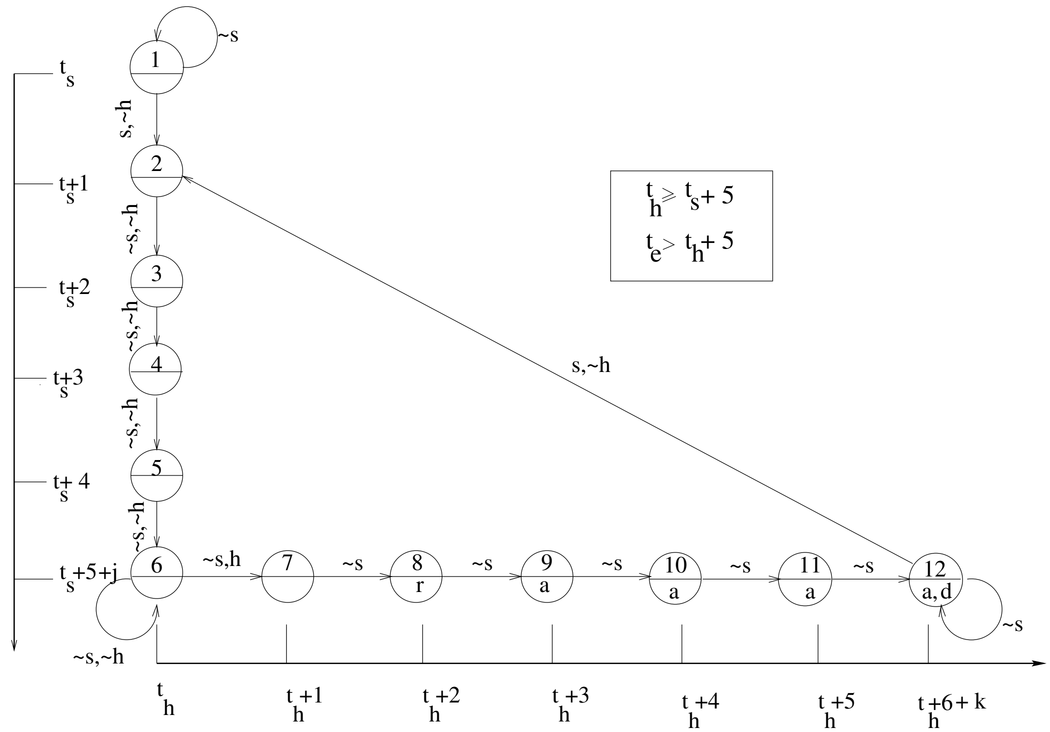

Figure 6: ASM of the *Cleaned* switch fabric

Inspired by, the constraints from the above environment state machine which represents the port controllers behavior, we describe in the following the overall behavior of the switch fabric. It can be expressed in the form of a finite state machine (ASM) having 12 states (Figure 6). To simplify the presentation, the symbols *s* and *h* denote a *frame start* (f_s=1) and the arrival of headers (active bit set in at least one *Din*), respectively; "~" denotes negation, and the symbols *a*, *d* or *r* inside a state represent the processing of the acknowledgment output (*Aout*), the data output (*Dout*) or round-robin arbitration, respectively. Note that the absence of an acknowledgment or data symbol in a state means that the default value 0 is produced.

Two time axes illustrate the time units of a frame to which the transitions correspond. The symbols t_s and t_h represent the arrival time of a *frame start* signal and the arrival time of a header, respectively. The end time (t_e) of a frame is not given, since it is the same as t_s of the next frame. State 1 is the initial state from which a frame may begin without any delay. This complies with the first constraint on the environment of the switch. After a waiting loop for the first *frame start* in state 1, states 2 to 6 describe the behavior of the fabric after the arrival of a *frame start*, with at least a five-cycle delay before the arrival of the headers. This delay represents the second constraint on the environment. The waiting loop for the arrival of the headers in state 6 is shown by a natural number *j*. States 7 to 12 describe the behavior of the fabric after the arrival of the headers. When the headers arrive, the *frame start* signal must not arrive before at least 6 cycles to comply with the third constraint on the environment. The arrival of a *frame start* in state 12 complies with the last constraint of the environment which describes that the next frame will not arrive before at least 11 cycles from the current *frame start.* After arbitration (state 9), the switch fabric transfers the acknowledgments in each cycle of a frame and switches data delayed by three cycles. This delay is represented using the sequence of transitions from state 9 to 12. The self-loop in state 12 represents the transmission of data and acknowledgments in the remaining cycles of the cell (indicated by a natural number *k*). The arrival of a *frame start* in state 12 marks the beginning of another frame. Here, a new sequence of state transitions along the t_s axis progresses similarly as in states 2, 3, 4, 5 and 6 described above.

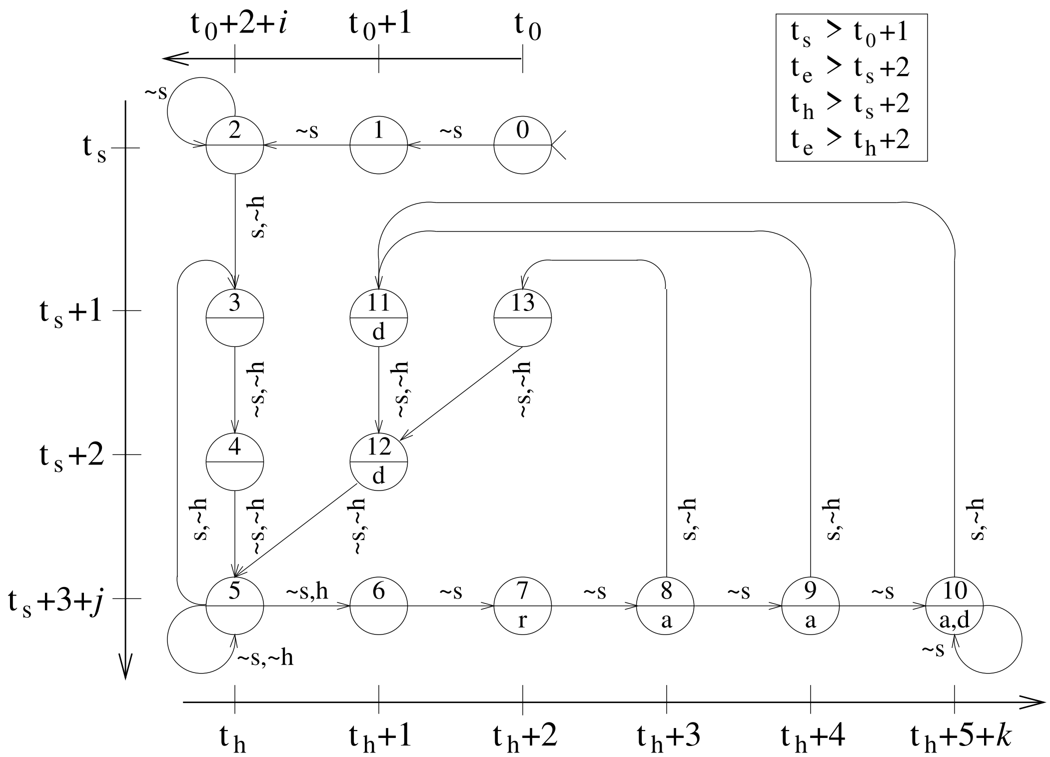

Figure 7: ASM of the *Original* switch fabric

Figure 7 represents the *Original* ASM of the switch fabric that used in the Original MDG verification of the Fabric [13]. To model the computation in MDG of the acknowledgments, the data outputs and the round-robin arbitration, we use the same techniques described in [13].

3.3 Implementation of the Switch Fabric

Figure 8 shows a block diagram of the *Original* 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. We do not describe the functionality of each module in this section. For more details about the implementation refer to [3].

To reflect the modifications suggested in [5], minor changes were made to the Timing unit, Arbiters, control path between the Arbitration and Dataswitch units and datapath to the Dataswitch unit of the original implementation. The modified Timing module ensures that the header and frame start signals must not occur together. The *frame start* signal just gets there 5 cycles later as required to make it trigger 5 cycles later. All it does is delay the triggering of the switching to give the cell bytes time to arrive. The block diagrams of the *Original* and the *Cleaned* version of the Timing module are given in Figure 10 and Figure 11, respectively. The shaded boxes in Figure 9 represent the modified modules in the *Cleaned* implementation.

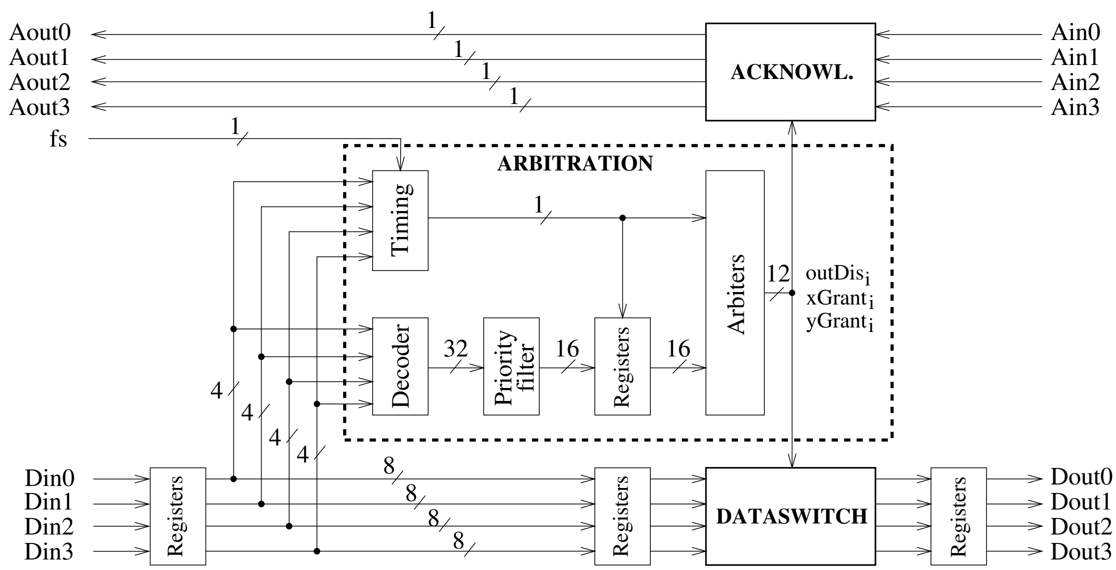

Figure 8: Fairisle switch fabric *Original* implementation

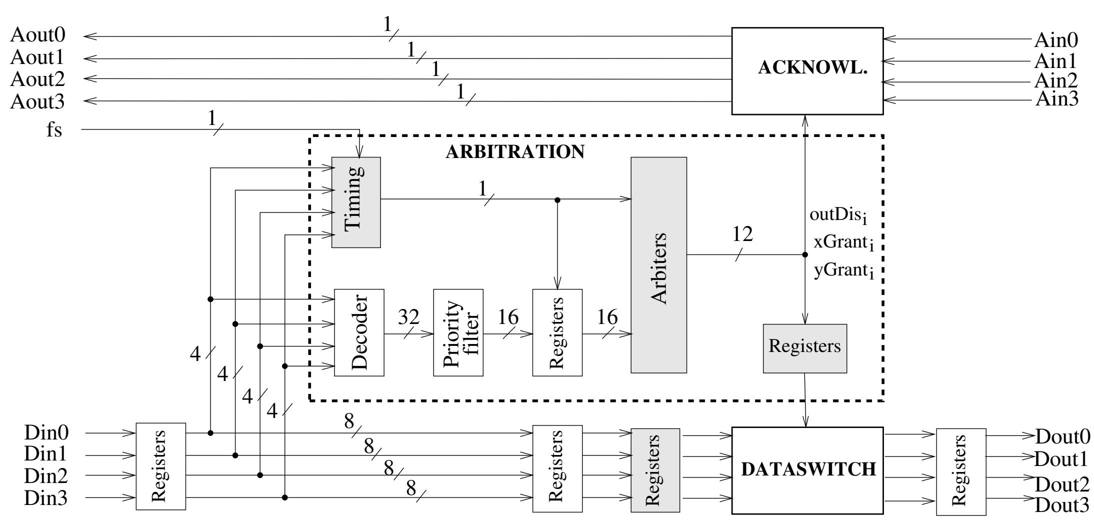

Figure 9: Fairisle switch fabric *Cleaned* implementation

The Arbiters generate the *output disable* and *grant* signals. They work consistently only when a *frame start* arrives at the same time as a *routeEnable*. The *Original* Arbiters disable the outputs one cycle earlier than is desirable. This is essential because of the way the dataswitch part is implemented. On each cycle, to determine which output the current byte should be sent to, the Dataswitch consults the two bits of the grant control signal produced by the Arbiter. One of those bits is sampled on the cycle before it is used, but the other is sampled on the same cycle. This ultimately means that the grant signal for the last cycle cannot be used as its value changes between the bits being sampled. The problem is removed by adding extra delays across the path to the

Dataswitch. The *Cleaned* Arbiters disable the outputs one cycle later than the *Original* one, so that the last bytes of a cell are not ignored.

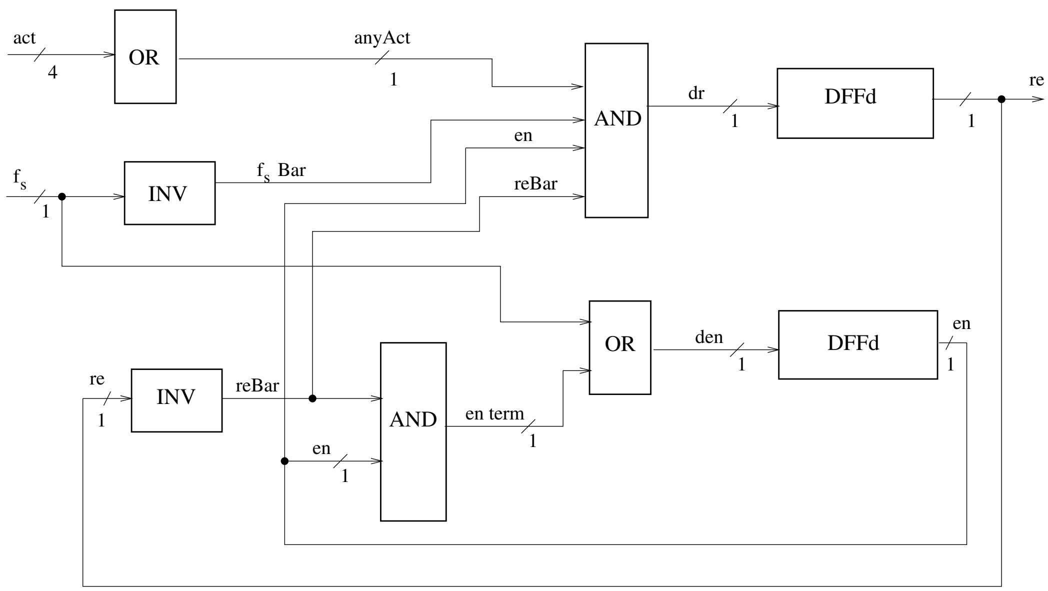

Figure 10: The implementation of the *Original* timing module

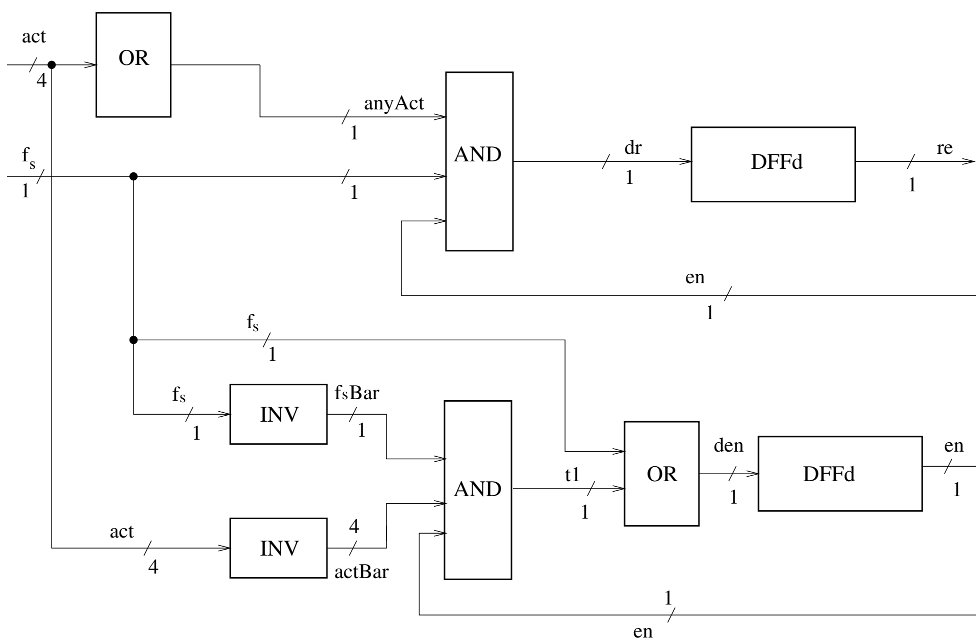

Figure 11: The implementation of the *Cleaned* timing module

The Dataswitch module chooses a word to be output to each of the output ports. It delays the data long enough for an arbitration decision to be made. To comply with the extra delay in the arbitration unit, minor changes had been made to its internal timing so it can read the two grant lines at a more sensible time. To do so, an extra register is added across the datapath to Dataswitch unit.

4. MDG Verification of the *Cleaned* Fabric

The *Cleaned* version was designed and formally verified, based on the modifications described in the previous section. In the following two sub-sections we describe property checking and sequential equivalence checking of the switch fabric. Before checking the equivalence of the specification of the switch fabric against the implementation, we have to make sure that these two models themselves are correct with respect to the new environment.

4.1 Property Checking

We applied property checking to ascertain that both implementation and specification of the switch fabric satisfy some specific requirements while working under the control of the environment, i.e., port controllers. Sample properties are correct circuit-reset and correct data-routing. Using the time points t_s, t_h and t_e, as introduced in Section 3, we described several properties which reflect the modified behavior of the switch fabric. The verification of the *Cleaned* version of the switch fabric was done using the following four properties [13]:

- *Property 1*: From t_s+5 to t_h+5, the default value (*zero*) appears on the data output port $Dout_i$, where *zero* is a generic constant and $i = 0,...,3$.
- *Property 2*: From t_s+1 to t_h+2, the default value (0) appears on the acknowledgment output port $Aout_i$, $i = 0,...,3$.
- *Property 3*: From t_h+6 to t_e-1 (i.e., 1 cycle before the next t_s), if input port i, $i = \{0,..,3\}$, chooses output port j, $j = \{0,..,3\}$, with the priority bit set in the header, and no other input ports have their priority bits set, the value on $Dout_j$ will be equal to that of Din_i 5 clock cycles earlier.
- *Property 4*: From t_h+3 to t_e-1 (i.e., 1 cycles before the next t_s), if input port *i* chooses output port *j* with the priority bit set in the header, and no other input ports have the priority bit set, the value on $Aout_j$ will be that of Ain_i.

Properties 1 and *2* deal with the reset behavior of the circuit, while *Property 3* and *4* state specific behaviors of the switching of cells. Although the (informal) description of the above properties explicitly involves the notion of time, we can verify them using only safety property checking based on the environment state machine model described earlier. This is elaborated in the following sub-sections.

4.1.1 Properties Description

The 64 state environment state machine (Figure 4) represents the cyclic behavior of the port controller of the ATM switch. This state machine periodically generates a frame of 64 clock cycles: starting from state 2 and back to it corresponds to one frame. The data inserted into the fabric has a length of 52 bytes (48 bytes data + 4 bytes header). Using the following ITE (If-Then-Else) formulas of the MDG-HDL, we can restate the previous four properties in terms of a state variable ***s*** of the environment state machine as follows.

- *Property 1*: **If** ($s \in [6, ..., 11]$) **then** $Dout_i = zero$ **else** don't care
- *Property 2*: **If** ($s \in [2, ..., 8]$) **then** $Aout_i = 0$ **else** don't care
- *Property 3*: **If** ($s \in [12, ..., 63]$) $\wedge$ *priority*[0..3] = [1,0,0,0] $\wedge$ *route*[0] = 0 **then** $Dout_0 = Din_0$' **else** don't care

- *Property 4*: **If** ($s \in [9, ..., 63]$) $\wedge$ *priority*[0..3] = [1,0,0,0] $\wedge$ *route*[0] = 0 **then** $Aout_0 = Ain_0$ **else** don't care

where Din_0' is the input of Din_0 5 clock cycles earlier, *priority*[0..3] are the priority bits of the four input ports and *route*[0] represents the routing bits for input port 0 (refer to Figure 2).

4.1.2 Properties Verification

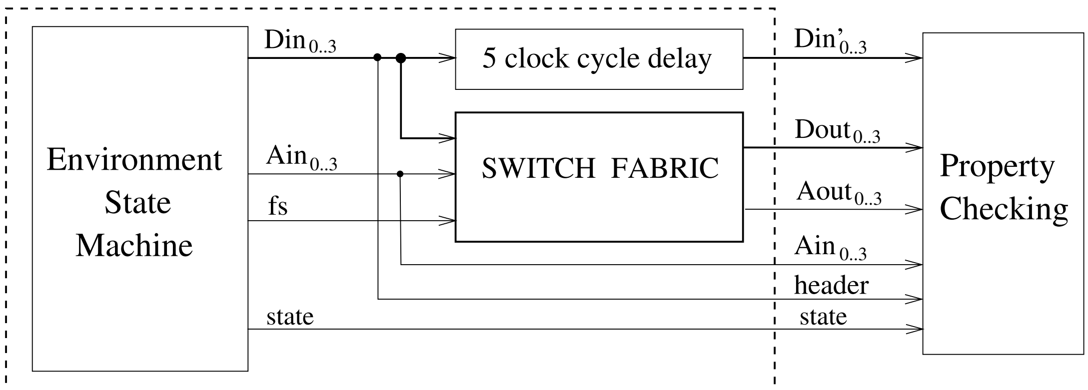

Figure 12: Composed state machine for property checking

To verify these safety properties, we composed the fabric (specification or implementation) with the environment state machine as shown in Figure 12. As there is a 5-clock-cycle delay for the cells to reach the output ports, a delay circuit (five-stage shift register) is used to memorize the input values that are to be compared with the outputs. Thus we can state the properties in terms of the equality between Din_i' and $Dout_j$ (e.g., *Property 3*). By combining these machines (the dashed frame in Figure 12) and the delay counter, we obtain the required platform for verification. This verification technique is indeed inspired by the technique described in [13]. By using the property checking facility of the MDG tools, we checked in each reachable state if the outputs satisfy the logic expression of the property which should be true over all reachable states. The experimental results from the verification of all the properties stated in Section 4.1.1 for both implementation and specification, are given in Table 1 and Table 2, respectively. All experimental results were obtained on a Sun Ultra SPARC 2 workstation (296MHz / 768 MB) and include CPU time in sec., memory usage in MB and the number of MDG nodes generated.

4.2 Equivalence Checking

The original design of the switch fabric was described in Qudos HDL. The switch fabric is composed of the acknowledgment, arbitration and dataswitch units. Each unit is further defined as a module which is further subdivided until the same gate-level implementation is reached as in the original Qudos HDL design. The authors in [13] translated the Qudos HDL description into MDG-HDL using the same collection of gates. The switch fabric has a 32 bit-wide data input and output lines. In Qudos HDL the data input and output lines are modeled as 32 individual lines. By using the data abstraction technique of the MDG tools, we could better describe these 32 individual lines as words of size n (e.g., an abstract sort *wordn*). This arbitrary word size makes the descriptions generic where we do not need to specify the exact word size. By abstracting the data lines from a bundle of bits to a compact word of abstract sort, we obtain an abstract RTL model of the switch fabric. This RTL model will be equivalent to the original gate-level description, if it produces the same output as the original gate-level for all input sequences. As the gate-level description is not generic, it is not possible to verify the equivalence of an abstract RTL model against an original gate-level implementation. To do that we should "instantiate" the data signals of the abstract RTL model to be 8-bits wide. We can decode the abstract data to Boolean data by using *uninterpreted* function symbols in the MDG-HDL description of the two models. Decoding can be realized by using 8 uninterpreted functions bit_i (i=0..7) of type $[wordn \rightarrow bool]$. bit_i extracts the i^{th} bit of an n-bit data. Based on this technique, one can verify that an abstract model is equivalent to its original gate-level implementation.

Table 1: Property checking on the implementation of the *Cleaned* fabric

Verification	CPU time (in sec.)	Memory (in MB)	MDG Nodes generated
Property 1	173.81	32	88085
Property 2	151.53	31	89738
Property 3	166.21	30	90554
Property 4	164.76	32	90933

Table 2: Property checking on the behavioral specification of the *Cleaned f*abric

Verification	CPU time (in sec.)	Memory (in MB)	MDG Nodes generated
Property 1	186.65	27	74948
Property 2	199.67	23	75287
Property 3	195.23	23	73020
Property 4	160.43	28	72843

Table 3: Equivalence checking between different levels of the *Cleaned* fabric

Verification	CPU time (sec.)	Memory Usage (MB)	Number of Nodes
RTL vs. Beh. Level	1934.56	148	230798
gate-level vs. RTL	30.85	13	13899

By using the sequential equivalence checking facility of the MDG tool, we verified that the abstract RTL implementation of the switch fabric complied with the specification of the behavioral model. To verify the RTL implementation against the behavioral specification, we made use of the fact that the corresponding input/output signals used in both descriptions have the same sort. We obtained a complete verification of the switch fabric from a behavioral specification down to the gate-level implementation using the above two verification steps. The experimental result of the verification is given in Table 3.

5. Comparison between *Cleaned* and *Original* version of the switch fabric

The motivation of this work was to compare the formal verification, in terms of time, of the *Cleaned* version of the Fairisle ATM switch fabric with the *Original* version using MDG tool. Timing aspect of formal verification is an important issue to the industrial community. We are comparing these two versions with respect to the *machine-time* and the *human-time*.

Human time spent on the verification of the *Original* version was longer than that of the *Cleaned* version. The amount of work in re-running a verification of a modified design is minimal compared to the initial effort since the latter includes all the modeling aspect. In the verification by MDG tools, manual interventions is needed for variables ordering which has an impact on the verification time. In the verification of the *Original* version, much of the time was spent on determining a suitable variable ordering. As there were no major changes to the *Original* version, we did not spend much time on redetermining a suitable variable ordering. The translation of the original Qudos HDL design description to the MDG-HDL gate-level structural model took about one person-week as described in the paper by Tahar *et al.* [12]. The time spent on the modification of the structural description of the design for the *Cleaned* version was four person-days. Because the verifier needs to understand the design thoroughly, the time spent for understanding and writing the behavioral specification of the *Original* version was about four person-weeks. On the other hand, for the *Cleaned* version it took two person-weeks. In the verification of the *Original* version, the time required to setup four properties, to build the environment state machine, to conduct the property checking both on the implementation and the specification and to interpret the results was about three person-weeks. For the *Cleaned* version, building a new environment state machine and conducting the property checking on both the implementation and the specification was taken about two person-weeks. The equivalence checking of the RTL implementation with its behavioral specification and the RTL model against the gate-level model of the *Original* version required about two person-weeks as the adoption of abstraction mechanisms and correction of description errors for RTL implementation were needed. On the other hand for the *Cleaned* version, it took about one person-week. The summary of the differences between the *Original Cleaned* version, in terms of *human-time* taken during the verification phase, is given in Table 4.

Table 4: Summary of human-time taken for the Verification

Verification Phase	Cleaned version	Original version
Behavioral specification description	Two person-weeks	Four person-weeks
Implementation description of the Cleaned ver.	Four person-days	one person-week
New Env. state machine and Property checking	Two person-weeks	Three person-weeks
Equivalence checking: RTL vs. Beh. Spec. and RTL vs. Gate-level	One person-week	Two person-weeks

To demonstrate the reduced verification time we compare the *machine-time* taken to complete the *Cleaned* version verification with that for the *Original* version. The machine-time taken by the *Cleaned* version for both the Property checking and the Equivalence checking has been reduced by a significant amount of CPU time than that of the *Original* version. The differences between the verification of the *Cleaned* and the *Original* versions are illustrated with respect to CPU time taken, memory usages and MDG nodes generated (see Table 5).

Table 5: Experimental Results for the Verifications of the *Cleaned* and *Original* Fabric

Verification	***Cleaned* version**			***Original* version**		
	CPU time (s)	Memory (MB)	Nodes Generated	CPU time (s)	Memory (MB)	Nodes Generated
Reachability Analysis						
Specification	180.40	32	73157	188.59	36	74130
Implementation	219.22	34	90208	232.74	35	90319
Property checking						
Specification						
Property 1	186.65	27	74948	251.53	25	74554
Property 2	199.67	23	75287	279.02	26	76265
Property 3	195.23	23	73020	257.52	25	74636
Property 4	160.43	28	72843	236.42	25	74441
Implementation						
Property 1	173.81	32	88085	235.34	34	92229
Property 2	151.53	31	89738	233.49	35	93882
Property 3	166.21	30	90554	205.04	33	92052
Property 4	164.76	32	90933	268.75	84	225486
Equivalence checking						
RTL vs. Beh. Spec.	1934.56	148	230798	2210.22	162	245707
RTL vs. Gate-level	30.85	13	13899	30.85	13	13899

6. Conclusions

In this report, we have demonstrated that design for verifiability can have a significant effect on the speed of verification using automated decision diagram based technique. The same result was obtained by using interactive proof with the HOL theorem prover for the same design verification. The difference in nature of these two verification methodologies suggests design for verifiability can be widely applicable as design for testability. One of the motivations of this work was to show that designers can ease the verification task without compromising other design considerations. Our investigation suggests that one way this can be done is by ensuring that the operating assumptions of modules are as few and as simple as possible. Design for verifiability in mind makes any design simple to verify. The designer may have to work a little harder to ease the verifier's task. However, the result is a much cleaner design. It thus can be done early in the design cycle. The development of design constraints for formal verification would be useful. This is vital for safety-critical systems where formal verification techniques are most likely to be used.

The implementation we considered for this investigation was the Fairisle 4 by 4 switch fabric which performs the actual switching of data cells and forms the heart of the ATM Fairisle communication network. We made some changes to the timing constraints of the fabric which is controlled by the environment of the fabric, i.e., port controllers. By changing these timing constraints we made the operating assumptions of the fabric simpler and cleaner. We also changed the design of the Arbiters, the Timing unit, the control path between the Arbitration and Dataswitch unit and datapath to the Dataswitch unit without loss of any significant functionality. The verification time taken by both *human* and *machine* for the modified design (*Cleaned* version) was much less than that of the original design (*Original* version) as demonstrated in the previous section. Based on the above statistics we can conclude that the verification time can be saved if the "design for verifiability" is integrated into the design process itself.

References

[1] F. Corella, Z. Zhou, X. Song, M. Langevin and E. Cerny, "Multiway Decision Graphs for Automated Hardware Verification", *Formal Methods in System Design,* Vol. 10, pp. 7-46, February 1997.

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

[3] P. Curzon, "The Formal Verification of the Fairisle ATM Switching Element", *Technical Reports 328 & 329*, University of Cambridge, Computer Laboratory, March 1994.

[4] P. Curzon, "Tracking Design Changes with Formal Machine-checked Proof", *The Computer Journal*, Vol. 38, No. 2, pp. 91-100, July 1995.

[5] P. Curzon and I. Leslie, "Improving Hardware Designs whilst Simplifying their Proof", *Designing Correct Circuits,* Workshops in Computing, Springer-Verlag, 1996.

[6] M. Gordon and T. Melham, *Introduction to HOL: A theorem Proving Environment for Higher Order Logic.* Cambridge Univ. Press, Cambridge, U.K., 1993.

[7] D. Ginsburg, "ATM Solutions for Enterprise Internet working", Addison Wesley, 1996

[8] 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.

[9] I. Leslie and D. McAuley, "Fairisle: An ATM Network for Local Area", *ACM Communication Review*, Vol. 19, pp. 237-336, September 1991

[10] J.Lu, S. Tahar, D. Voicu and X. Song, "Model Checking of a Real ATM Switch", *Proc. IEEE International Conference on Computer Design* (ICCD,98), Austin, Texas, USA, IEEE Computer Society Press, pp. 195-198, October 1998;

[11] S. Rajan, M. Fujita, K. Yuan, and M. Lee, "High-Level Design and Validation of ATM Switch", *Proc. IEEE International High Level Design Validation and Test Workshop* (HDLVT'97), Oakland, California, USA, November 1997.

[12] S. Tahar and P. Curzon, "Comparing HOL and MDG: A case study on the Verification of an ATM Switch Fabric", *Nordic Journal of Computing,* Vol. 6, pp. 372-402, 1999.

[13] 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, pp. 956-972, July 1999.