Modeling and Automatic Formal Verification of the Fairisle ATM Switch Fabric using MDGs

Sofiène Tahar, Xiaoyu Song[§], Eduard Cerny[§], Zijian Zhou[‡] and Michel Langevin[‡]

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

[§] University of Montreal, IRO Dept., Montréal, Québec, H3C 3J7 Canada
E-mail: {song, cerny}@iro.umontreal.ca

[‡] Nortel Technology, Ottawa, Ontario, K1Y 4H7 Canada
E-mail: {mlangevin, zzhou}@nortel.ca

Abstract. *In this paper we present several techniques for modeling and formal verification of the Fairisle Asynchronous Transfer Mode (ATM) switch fabric using Multiway Decision Graphs (MDGs). MDGs represent a new class of decision graphs which subsumes ROBDDs while accommodating abstract sorts and uninterpreted function symbols. The ATM device we investigated is in use for real applications in the Cambridge University Fairisle network. We modeled and verified the switch fabric at three levels of abstraction: behavior, RT and gate levels. In a first stage, we validated the high-level specification by checking specific safety properties that reflect the behavior of the fabric in its real operating environment. Using the intermediate abstract RTL model, we hierarchically completed the verification of the original gate-level implementation of the switch fabric against the behavioral specification given as an abstract state machine (ASM). Since MDGs avoid model explosion induced by data values, this work demonstrates the effectiveness of MDG-based verification as an extension of ROBDD-based approaches. All the verifications were carried out fully automatically in a reasonable amount of CPU time.*

1 Introduction

The consequence of errors in the design or implementation of communication networks and components is increasingly critical. This is especially so if networks are used in safety-critical applications where communications problems could cause loss of life. Simulation and testing have traditionally been used for checking the correctness of those systems. However, it is practically impossible to run an exhaustive test or simulation for such large and complex systems. The use of formal verification for determining the correctness of digital systems is thus gaining interest, as the correctness of a formally verified design implicitly involves all cases regardless of the input values. One obstacle of formal verification is, however, the fact that existing techniques either require a deep understanding of mathematical logic and formal proofs or are insufficient for handling large systems [14].

ATM (*Asynchronous Transfer Mode*) is considered as the network technology for addressing the variety of needs for new high-speed, high-bandwidth applications. ATM was adopted by the CCITT as the target mode for the B-ISDN (the Broadband Integrated Services Digital Network) [12]. Although ATM is being hailed as the most important communication mechanism in the foreseeable future, there is currently little experience on the application of formal verification to ATM network hardware.

In this paper, we present several techniques for modeling and formally verifying an ATM network component using a new class of decision graphs, called Multiway Decision Graphs (MDGs) [7]. These decision graphs subsume the class of Bryant's Reduced Ordered Binary Decision Diagrams (ROBDD) [2] while accommodating abstract sorts and uninterpreted function symbols. The device we investigated is the Fairisle 4 by 4 switch fabric which is part of the Fairisle ATM network [17] designed and in use at the Computer Laboratory of the University of Cambridge. This switch fabric, which forms the heart of the ATM Fairisle network, was fabricated without consideration for formal verification and is used for real data transmission, e.g., in multi-media applications. The Fairisle switch fabric thus provides a realistic vehicle for the investigation of the formal verification of ATM networks.

The main contributions of this work are the development of a specification of the expected behavior of the ATM switch fabric, the modeling of its implementation at the gate and a more abstract RT level, and the successful equivalence checking between the different levels. Using invariant checking, we have been able to validate our specification by verifying properties which reflect the behavior of the fabric in the Fairisle ATM environment. In addition, we checked the correctness of several implementations with introduced errors which were successfully identified by the counterexample facility in the MDG tools. The behavioral specification and thus the higher level verification have no restrictions on the frame size, cell length or word width. The verification is based on reachability analysis of the product machine of the implementation and the specification, modeled as networks of Abstract State Machines (ASM) [6]. MDGs are used to encode the output and transition relations of ASMs and the set of reachable abstract states, allowing implicit abstract state enumeration. Using the applications provided by the MDG software package, all verification tasks were achieved fully automatically in a reasonable amount of CPU time. This experiment confirms the effectiveness of the MDG-based verification methodology as an extension of ROBDD-based approaches [2], since model explosion induced by data values expanded to their binary representation is largely avoided. Our results demonstrate the feasibility of applying the MDG-based verification method to a realistic hardware design.

The organization of this paper is as follows: in Section 2, we review related work on formal verification of ATM hardware. In Section 3, we give a brief introduction to multiway decision graphs and the related MDG verification techniques. In Section 4, we overview the Fairisle ATM switch. In Section 5, we describe the behavioral specification of the switch fabric and show its modeling as abstract state machine (ASM) represented by MDGs. The descriptions of the hardware implementation of the switch fabric and the related MDG modeling at the gate and RT levels are sketched out in Section 6. In Section 7, we describe the property checking approach adopted to validate the behavioral specification. In Section 8, we explore the equivalence verifications between the different hierarchy levels, providing a complete verification from the high-level behavior down to the gate-level implementation. We also compare our results to related work using HOL and VIS. In Section 9 we draw our conclusions.

2 Related Work

There exist few results in the open literature that are directly related to the *formal verification* of ATM network hardware components.

Chen et. al at Fujitsu Digital Technology Ltd. [5] exploited symbolic model checking to detect a design error in an ATM circuit. The ATM circuit consists of about 111K gates and supports high-speed switching operations at 156 MHz. When the circuit was manufactured it showed an abnormal behavior under certain circumstances. Using SMV (Symbolic Model Verifier) [16], they identified the error by checking some properties described in CTL (Computational Tree Logic) [16]. Given the Boolean representation in SMV, to avoid state space explosion, they abstracted the width of addresses from 8 bits to 1 bit, and the number of addresses in an FIFO (Write Address FIFO) from 168 to 5. However, in some cases a property could not be verified because of this reduction and a detailed gate-level model was needed for certain blocks to pinpoint the source of the error. In general, however, the adopted data abstraction method was not quite adequate.

Curzon [8] verified the 4 by 4 fabric of the Fairisle switch fabric using the HOL theorem prover [13]. (More details about the Fairisle ATM switch will be presented in Section 4 since the verification of the same fabric is investigated in the current paper). He hierarchically verified each of the modules used in the design of the switching element, by describing the behavioral and structural specifications down to the gate level, and then proving the related correctness theorems in HOL. The separate proofs were then combined to prove the correctness of the whole switch fabric against a high-level specification of its timing behavior. From this verification, the author found no error in the fabricated implementation. However, several errors in the formal specifications were found, which highlights the fact that a correct specification could be just as hard to develop as an implementation.

Other researchers have also used the Fairisle fabric as a case study. Schneider et. al [19] formally verified it using the verification system MEPHISTO which is based on the HOL theorem prover. They described the behavior and structure of the lower-level hardware modules and automated their verification. Jakubiec et. al [15] used it in their work based on the Coq proof system. Garcez et. al [11] has also verified some properties of the fabric using the HSIS model checking tool.

More recently, Lu et. al [18] used the VIS tool [1] to verify relevant liveness and safety properties (described in CTL) on various abstracted models of the Fairisle fabric. In addition, they conducted equivalence checking between the behavioral and structural specifications of the sub-modules written in Verilog. The authors also re-implemented the fabric using the Synopsys synthesis tool. While succeeding in the equivalence checking at a certain level of the design hierarchy and full model checking, the VIS tool failed in verifying the whole switch fabric using equivalence checking due to the state space explosion of even the most reduced model with a 4-bit datapath.

Although Curzon [8] showed the effectiveness of HOL theorem proving for verifying an ATM switch, the use of HOL is interactive and requires much expertise to guide the verification process [14]. The work at Fujitsu Ltd. [5] used the ROBDD-based SMV model checker which is automatic and the authors succeed in checking some important properties related to the circuit implementation. Yet, the adopted data abstraction (e.g., using 1 bit to represent 8-bit data width) for avoiding state explosion is not always applicable. The ATM verification performed by Lu et. al [18] using an automated ROBDD-based tool, VIS, failed to complete the verification of even a very reduced model of the switch fabric through equivalence checking.

To overcome these drawbacks, we attempt to raise the level of abstraction of automated verification methods to that of interactive methods, without sacrificing automation. *Multiway Decision Graphs* (MDGs) have been recently proposed to represent circuits at a more abstract level [6]. MDGs are based on a subset of a many-sorted first-order logic with abstract sorts and uninterpreted function symbols. They subsume the class of Reduced Ordered Binary Decision Diagrams (ROBDD) [2]. In the next section, we will briefly describe MDGs and their verification methods. For more details about MDGs, see [6][23].

3 Multiway Decision Graphs

The formal system underlying MDGs is a subset of many-sorted first-order logic augmented with a distinction between *abstract sorts* and *concrete sorts*. Concrete sorts have finite *enumerations*, while abstract sorts do not. The enumeration of a concrete sort α is a set of distinct constants of sort α. The constants occurring in enumerations are referred to as *individual constants,* and the other constants as *generic constants* and could be viewed as 0-ary function symbols. The distinction between abstract and concrete sorts leads to a distinction between three kinds of function symbols. Let *f* be a function symbol of type $\alpha_1 \times \alpha_2 \times ... \times \alpha_n \rightarrow \alpha_{n+1}$. If α_{n+1} is an abstract sort, then *f* is an *abstract function symbol.* If all the α_1 ... α_{n+1} are concrete, then *f* is a *concrete function symbol*. If α_{n+1} is concrete while at least one of the α_1 ... α_n is abstract, then *f* is referred to as a *cross-operator;* Concrete function symbols must have explicit definition; they can be eliminated and do not appear in MDGs. Abstract function symbols and cross-operators are *uninterpreted.* A *multiway decision graph* (MDG) is a finite, directed acyclic graph (DAG). An internal node of an MDG can be a variable of a concrete sort with its edge labels being the *individual constants* in the enumeration of the sort; or it can be a variable of abstract sort and its edges are labeled by abstract terms of the same sort; or it can be a *cross-term* (whose function symbol is a cross-operator). An MDG may only have one leaf node denoted by **T**, which means all paths in the MDG are true formulas. Thus, MDGs essentially represent relations rather than functions.

Using MDGs, a data value can hence be represented by a single variable of abstract sort, rather than by concrete Boolean variables. Variables of concrete sorts are used for representing control signals, and variables of abstract sorts are used for representing datapath signals. With MDGs, data operations can be represented by uninterpreted function symbols. As a special case of uninterpreted functions, cross-operators are useful for modeling feedback from the datapath to the control circuitry. For circuits with large datapaths, such as the ATM Fairisle switch fabric, the representation of MDGs are much more compact than that of ROBDDs, thus greatly increasing the range of circuits that can be verified since the verification is independent of the width of the datapath.

A state machine is described using finite sets of input, state and output variables, which are pairwise disjoint. The behavior of a state machine is defined by its transition/output relations, together with a set of initial states. An abstract description of the state machine, called *abstract state machine* (ASM) [7], is obtained by letting some data input, state or output variables be of an abstract sort, and the datapath operations be uninterpreted function symbols. As ROBDDs are used to represent sets of states, and transition/output relations for finite state machines, MDGs are used to compactly encode sets of (abstract) states and transition/output relations for ASM. We thus lift the implicit enumeration technique [3] from the Boolean level to the abstract level, and refer to it as *implicit abstract enumeration* [6]. Starting from the initial set of states, the set of states reached in one transition is computed by the relational product operation. The frontier-set of states is obtained by *pruning* (removing) the already visited states from the set of newly reached states using pruning-by-subsumption. If the frontier-set of states is empty, then a least fixed point is reached and the reachability analysis procedure terminates. Otherwise, the newly reached states are merged (using disjunction) with the already visited states and the procedure continues the next iteration with the states in the frontier-set as the initial set of states.

Like ROBDDs, MDGs must be *reduced* and *ordered*. They obey a set of well-formedness conditions given in [6] which turns MDGs into a canonical representations. Algorithms for computing *disjunction*, *relational product* (*conjunction* followed by *existential quantification* [3]), *pruning-by-subsumption* (for test of set inclusion) and *reachability analysis* (using abstract implicit enumeration) have been developed in a prototype MDG software package implemented in Quintus Prolog. Besides extending this application scope, the rewriting facility can be used to simplify MDGs. A detailed description of the algorithms can be found in [6]. Based on these algorithms, the following verification applications are currently provided within the MDG software package:

Combinational verification: taking advantage of the canonicity of MDGs [6], the equivalence checking of input-output relations for two combinational circuits is computed. The MDGs representing the input-output relation of each circuit are computed by the relational product algorithm from the MDGs of the components of the circuit. Then it is checked whether the two MDGs are isomorphic.

Safety properties checking: using reachability analysis, the state space of a given sequential circuit (abstract state machine) is explored and checked if a certain invariant holds in all the reachable states. The transition relation of the abstract state machine is represented by an MDG computed by the relational product algorithm from the MDGs of the components.

Sequential verification: the behavioral equivalence of two sequential circuits (abstract state machines) is verified by checking that the circuits produce the same sequence of outputs for every sequence of inputs. The same inputs are fed into the two circuits and reachability analysis is performed on the combined circuit using an invariant asserting the equality of the corresponding outputs in all reachable states.

Counterexample generation: when an invariant is not satisfied during safety property checking or sequential verification, the MDG tools generate a counterexample to help with identifying the source of the error. A counterexample consists of a list of assumptions, inputs and states values at each clock cycle, and provides a trace leading from the initial state to the faulty behavior.

These MDG-based verification techniques have been used for the verification of a set of known (combinational and sequential) benchmark circuits including the verification of two simple non-pipelined microprocessors against their instruction set architectures [4]. In this paper, we investigate the verification of the Fairisle ATM switch fabric which is by an order of magnitude larger than any other circuit verified so far using MDGs.

4 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-octet tag containing routing information as shown in Figure 2) and a fixed number of data octets.

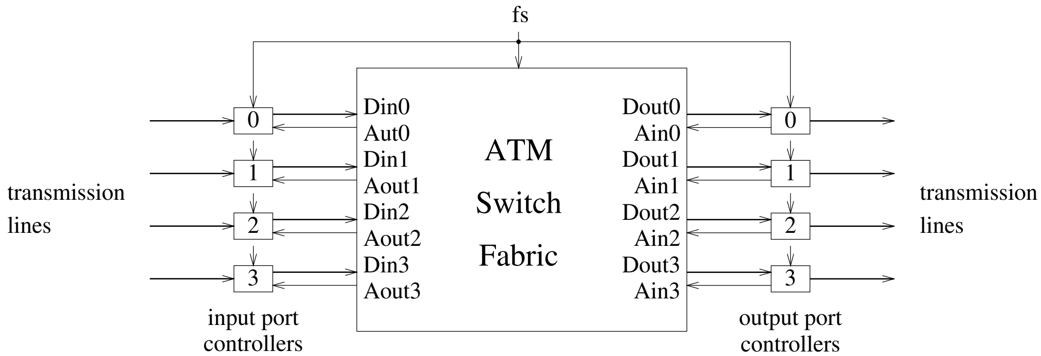


Figure 1: The Fairisle ATM switch

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. 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. 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 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* (*fs*) 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* only.

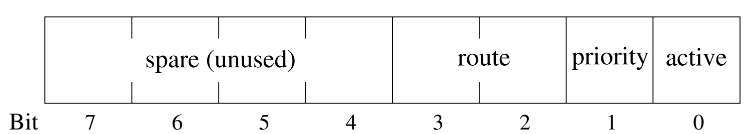


Figure 2: Header of a Fairisle ATM cell

The behavior of the *switch fabric* is cyclic. In each cycle or *frame*, it waits for cells to arrive, processes them, sends successful ones to the appropriate output ports, and sends acknowledgments. It then waits for the arrival of the next round of cells. The cells from all the input ports start when a particular bit (the *active* bit, Figure 2) of any one of them goes high. The fabric does not know when this happens. 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 throughout the frame then the frame is inactive—no cells are processed. Otherwise it is active.

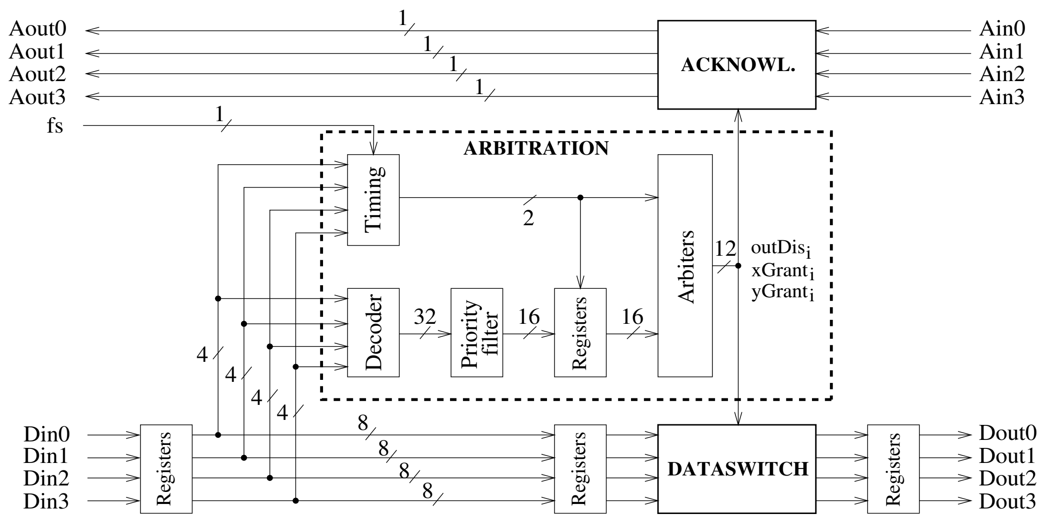


Figure 3: Block diagram of the Fairisle switch fabric

As shown in Figure 3, the inputs to the fabric consist of the cell data lines, the acknowledgments that pass in the reverse direction, and the frame start signal *fs* which is the only external control signal. The outputs consist of the switched data, and the switched and modified acknowledgment signals. The switch fabric is composed of an *arbitration unit* (*timing*, *decoder*, *priority filter* and *arbiters*), an *acknowledgment unit* and a *dataswitch unit*. The timing block controls the timing of decisions with respect to the frame start signal and the arrival time of the headers. Arbitration is implemented in two stages. The decoder reads the headers of the cells and decodes the port requests and priorities. The priority filter removes the requests with low priority and those from inactive inputs, and passes the actual request situation for each output port to the arbiters. The arbiters (in total four—one for each port) make arbitration decisions (when two or more cells have the same destination) for each output port i, pass the results to the other modules using grant signals $xGrant_i$

and $yGrant_i$ (Figure 3) and control the timing of the other units using output disable signals $outDis_i$ (Figure 3). The dataswitch unit performs the actual switching of data from input ports to output ports according to the most recent arbitration decision (a set of registers is used in the dataswitch; they are reset when their corresponding $outDis_i$ signal is high). The acknowledgment unit passes appropriate acknowledgment signals to the input ports. Negative acknowledgments are sent until arbitration is completed.

All the design units were repeatedly subdivided until the logic gate level was eventually reached, providing a hierarchy of components. The hardware design has a total of 441 basic components (a logic gate with two or more inputs, or a 1-bit flip-flop). The switch fabric was built on a 4200 gate equivalent Xilinx FPGA. The switching element can be clocked at 20 MHz and frame start pulses occur every 64 clock cycles. The design was described in Qudos HDL [10] hardware description language which was used for generating the Xilinx netlist. The Qudos simulator [10] was used to perform the original (non-formal) validation.

5 Specification of the Fabric Behavior

Inspired by the design documentation of the Fairisle switch fabric, we derived a high-level behavioral specification in the form of an abstract state machine. This specification was developed independently of the actual hardware design and includes no restrictions on the frame and cell lengths, and the word width. It reflects the complete behavior of the fabric under the assumption that the environment maintains certain timing constraints on the arrival of the frame start signal and the cell headers. In the following, we give a description of the expected behavior of the switch fabric in the form of a state machine. We then depict the ASM model of this behavior.

5.1 Expected Behavior of the Fabric

A timing-diagram of the expected input-output behavior of the switch fabric during an active frame is in Figure 4. After the frame starts (at time t_s), the switch waits for the headers to appear on the data input lines *Din*. After the arrival of the headers (at time t_h), an arbitration is done in at most 2 cycles. The successful cells (bytes that follow the headers on *Din*) are transferred to the corresponding output ports (*Dout*) with a delay of 4 cycles, while acknowledgments traverse in the opposite direction, without synchronous delay, starting at time t_h+3. Notice that the last cycle of a frame (at time t_e-1) does not transfer data. When there is no data or acknowledgment to transfer, the switch forces *zero* values on the output data lines (thus, the value of *Ain* are *don't care*).

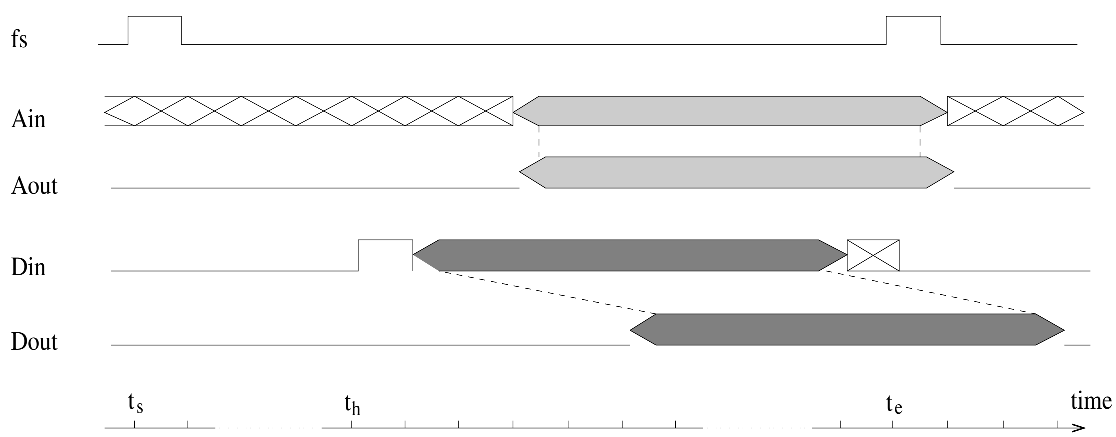


Figure 4: Expected behavior in an active frame

Based on a set of timing-diagrams (similar to the above example) which describe the expected input-output behavior of the switch fabric, we derived a high-level specification in the form of a finite state machine[1] (Figure 5). The state machine describes the fabric behavior under the following four assumptions about its environment:

1. At start up (t_0) the frame start (t_s) is delayed by at least 2 cycles before being asserted, i.e., $t_s \geq t_0+2$
2. The next frame start arrives ($t'_s=t_e$) at least 3 cycles after frame start (t_s), i.e., $t_e > t_s+2$
3. The headers arrive (t_h) at least 3 cycles after frame start (t_s), i.e., $t_h > t_s+2$
4. The headers arrive (t_h) at least 3 cycles before next frame start (t_e), i.e., $t_e > t_h+2$

There are 14 conceptual states in the machine. To simplify the presentation, the symbols *s* and *h* denote a frame start (*fs* = 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 conceptual state represent the computation 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 conceptual state means that the default value 0 is output.

1. Although an implementation of the ATM switch fabric already existed before we started this work, this specification was done by one of the co-authors without consulting it, i.e., the specification would be the same if it were developed before any hardware design of the switch fabric has been carried out.

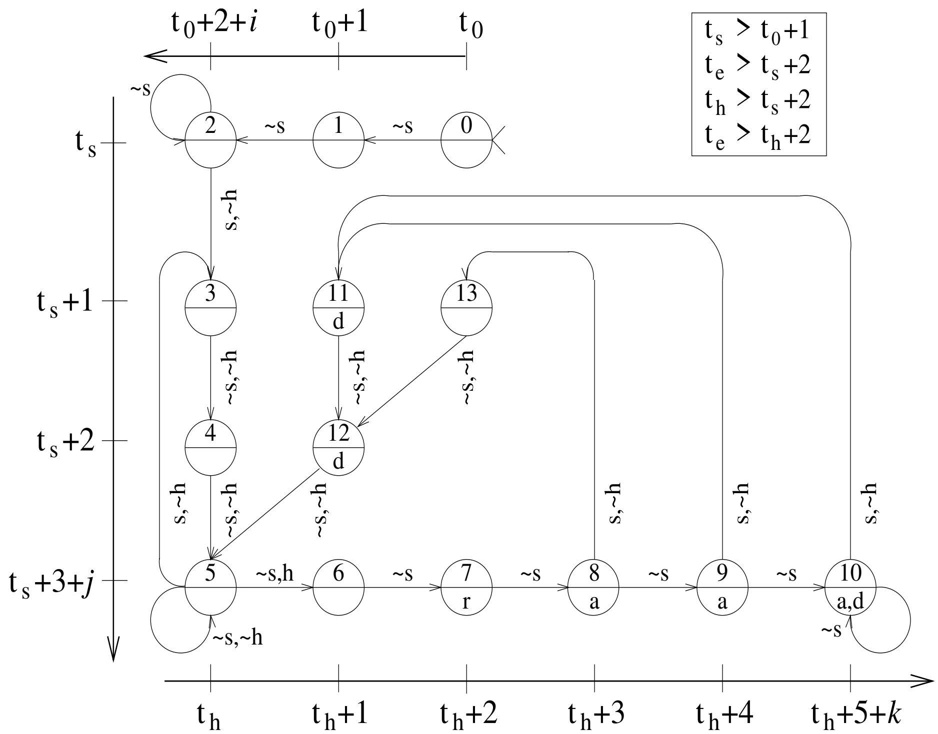


Figure 5: The expected switch fabric behavior

Three time axes illustrate the time units of a frame to which the transitions correspond. The symbols t_0, t_s and t_h represent the initial time, 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 0 is the initial state from which there must be two transitions without the arrival of the frame start (states 1 and 2). This complies with the first constraint on the environment of the switch. The states 0, 1 and 2 are related to the time axis t_0. The waiting loop for the first frame start in state 2 is shown by a natural number *i*.

States 3, 4 and 5 describe the behavior of the fabric after the arrival of a frame start, with at least a three-cycle delay before the arrival of either the headers or the next frame start. The delay represents the second and third constraints on the environment. These states are related to the time axis t_s. The waiting loop for the arrival of either the headers or the next frame start in state 5 is shown by a natural number *j*. Note that the arrival of the next frame start corresponds to the end of an empty frame.

States 6 to 13 describe the behavior of the fabric after the arrival of headers. When the headers arrive, the frame start signal must not arrive before at least 3 cycles to comply with the last constraint on the environment. States 6 to 10 are related to the time axis t_h. After arbitration (state 8), the switch transfers the acknowledgments in each cycle of a frame and switches data delayed by two cycles. This delay is represented using the sequence of transitions from state 8 to state 10. The self-loop in state 10 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 states 8, 9 or 10 marks the beginning of another frame. Here, a new sequence of state transitions along the t_s axis progresses similarly as in states 3, 4 and 5 described above, but considering possibly different scenarios for completing the transmission of the preceding cells.

To compute the acknowledgments, the data outputs and the round-robin arbitration, we use the following state variables:

- co_i (i= {0,…,3}) of enumeration {0,1}: co_i is 1 iff the output port *i* is connected. The co_i variables correspond to the $outDis_i$ signals of the implementation (Figure 3).
- ip_i (i={0,…,3}) of enumeration {0,…,3}: ip_i is the input port connected to the output port *i* (during arbitration, it is the last input port connected to the output port *i*). The ip_i variables correspond to the pair of $xGrant_i$ and $yGrant_i$ signals of the implementation (Figure 3).
- $sr_{i,j}$ (i={0,…,3}; j={1,…,4}) of enumeration {0,…,255}: $sr_{i,j}$ is the value of Din_i delayed by *j* clock cycles. That is, during each transition of the state machine, the data input Din_i is shifted in.

In the states annotated by *a* (8, 9 and 10) the values of $Aout_i$, i={0,…,3}, are computed as follows (**ef** stands for **else if**):

if ((co_0=1) **and** (ip_0=i)) **then** ($Aout_i$=Ain_0)
ef ((co_1=1) **and** (ip_1=i)) **then** ($Aout_i$=Ain_1)
ef ((co_2=1) **and** (ip_2=i)) **then** ($Aout_i$=Ain_2)
ef ((co_3=1) **and** (ip_3=i)) **then** ($Aout_i$=Ain_3)
else ($Aout_i$=0)

That is, if input port *i* is connected, $Aout_i$ takes the value of the corresponding Ain_j; otherwise, $Aout_i$ is the default 0.

In states annotated by d (10, 11 and 12), the values of $Dout_i$, i={0,…,3} are computed as follows:

if (co_i=0) **then** ($Dout_i$=0)

ef (ip_i=0) **then** ($Dout_i$=$sr_{0,4}$)

ef (ip_i=1) **then** ($Dout_i$=$sr_{1,4}$)

ef (ip_i=2) **then** ($Dout_i$=$sr_{2,4}$)

else ($Dout_i$=$sr_{3,4}$)

That is, if the output port i is connected, $Dout_i$ takes the value of the corresponding Din_j delayed by 4 cycles; otherwise, $Dout_i$ is 0. The values of co_i and ip_i are modified only during arbitration, i.e., during the transition from state 7 to state 8; each (co_i, ip_i) value-pair is computed from the values of all $sr_{j,2}$ (the cell headers), considering the active, priority and route fields, and the current value of ip_i for the round-robin arbitration. This can be easily described using **if-then-else** constructs, but it is too long to be shown here.

5.2 ASM Model for MDG-based Verification

The conventional method to model a state machine specification is to use finite state machines and represent them using Reduced Ordered Binary Decision Diagrams (ROBDD) [1]. However, the presence of 16 8-bit wide state variables in the specification and implementation makes a state space exploration procedure very difficult. To alleviate the problem, we use *Multiway Decision Graphs* (MDG) as introduced in Section 3. MDG-based modeling allows us to consider the data input, state and output variables as values of an abstract (i.e., non-specified) sort. For instance, the 8-bit-wide data in both ATM specification and implementation models can be described as values of an abstract sort *wordn*. MDG-oriented modeling using *Abstract State Machine* (ASM) [3] can represent an unbounded class of FSMs, depending on the interpretation of the abstract sorts and operators. In the following, we show how to model the specification of the ATM switch fabric as ASMs. We use a prolog-style HDL—MDG-HDL which is supported by the MDG software package. MDG-HDL allows the description of behavioral specifications by using high-level constructs such as ITE (If-Then-Else) formulas and CASE formulas, or tabular representations.

The main sorts and operators for the fabric ASM modeling are as follows:

- concrete sort *bool* = {0,1}
- concrete sort *port* = {0,…,3}
- concrete sort *Ctl* = {0,…,13}
- abstract sort *wordn* (representing data bytes)
- generic constant *zero* of sort *wordn*
- cross-operator *act* of type [*wordn* ⟶ *bool*] (representing the active field of header)
- cross-operator *pri* of type [*wordn* ⟶ *bool*] (representing the priority field of header)
- cross-operator *rou* of type [*wordn* ⟶ *port*] (representing the route field of header)

The switch fabric ASM is composed of

1) Input variables *fs*, Ain_i (i = {0,…,3}) of sort *bool*, and Din_i (i = {0,…,3}) of sort *wordn*;

2) Output variables $Aout_i$ (i = {0,…,3}) of sort *bool* and $Dout_i$ (i = {0,…,3}) of sort *wordn*;

3) State variables *c* of sort *Ctl*, co_i (i= {0,…,3}) of sort *bool*, ip_i (i={0,…,3}) of sort *port*, and $sr_{i,j}$ (i={0,…,3}; j={1,…,4}) of sort *wordn*.

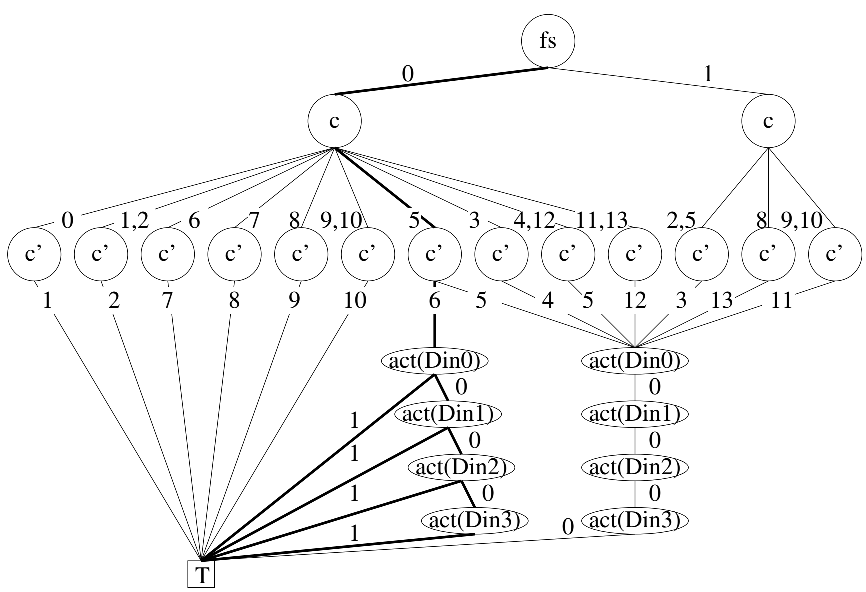


Figure 6: The MDG associated with next-state *c'*

The description of the ASM is completed by giving its output and next-state relations. An MDG is associated with each output and next-state variable, encoding its value in relation with the input and state variables. For instance, the MDG of the next-state variable *c'* is shown in Figure 6: The transition from state 5 to state 6 under the meta-symbols ~*s* and *h* of Figure 5 is encoded by the set of highlighted paths (*fs* = 0) and (*c* = 5) and (*c'* = 6), and at least one ($act(Din_i)$ = 1) representing the arrival of a header. The formula represented by the set of MDG paths is similar to the one represented by the set of ROBDD paths leading to the true leaf, except that first-order terms can appear along the paths. The terms $act(Din_i)$ are the *cross-terms;* they encode data-dependent decisions. The MDG of the output $Dout_0$ is in Figure 7. $Dout_0$ is equal to the corresponding $sr_{i,4}$ value, depending on ip_0 if the output port 0 is connected (co_0 = 1) and if the conceptual state *c* is 10, 11 or 12; otherwise, $Dout_0$ = *zero*. The MDGs of the other output and next-state variables can be derived in a similar way.

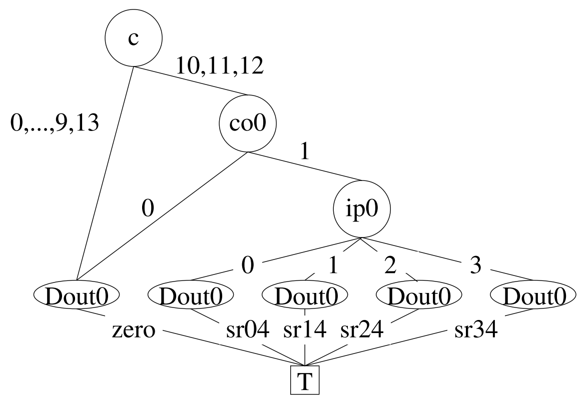


Figure 7: The MDG associated with $Dout_0$

6 Description of the Fabric Implementation

In this section, we describe the implementation of the fabric at the gate and the RT levels. We translated the original Qudos HDL models into very similar models using the Prolog-style MDG-HDL which allows the description of hierarchical hardware structures using module constructs and comes with a large library of predefined basic components (logic gates, multiplexors, registers, bus drivers, ROMs, etc.). Based on the gate-level description, we also produced an RTL implementation by modeling the data signals $Din_i/Dout_i$ (*i*=0,1,2,3) as *n*-bit words, and by describing the dataswitch using abstract multiplexors instead of logic gates. This lead to a simpler implementation of the dataswitch, reflecting the switching behavior in a more understandable way and using a smaller number of components.

6.1 Gate-Level Netlist

Many of the modules in the original Qudos description were large and logically unrelated while preserving the mapping to a Xilinx gate array. Similarly to the description done by Curzon in HOL [8], we organized the model in several levels of hierarchy, making use of modularity within MDG-HDL that is lacking in Qudos HDL, thus facilitating both the specification and the verification.

The switch fabric is composed of the acknowledgment, arbitration and dataswitch units (Figure 3). Each unit is defined as a module which is further subdivided until the same gate-level implementation is reached as in the original Qudos HDL design. To illustrate the similarities and differences of the two descriptions, we consider the Qudos HDL and MDG-HDL structural descriptions of a mutiplexing component of the dataswitch—DMUX4T2. We first give the Qudos HDL definition:

```
DEF DMUX4T2(d[0..3],x:IN;dout[0..1]:IO);
xBar:IO;
BEGIN
Clb:=XiCLBMAP5i20(d[0..1],x,d[2..3],dout[0..1]);
InvX:= XiINV(x,xBar);
B[0]:= AO(d[0],xBar,d[1],x,dout[0]);
B[1]:= AO(d[2],xBar,d[3],x,dout[1]);
END;
```

where the first statement is a dummy declaration providing information about the way in which the component should be mapped into a Xilinx gate array, and the AO components are AND-OR logic gates. Using MDG-HDL this same module is described as follows:

```
module(DMUX4T2
   port(inputs((d0,bool),(d1,bool),(d2,bool),(d3,bool)),(x,bool)),
        outputs((dout0,bool),(dout1,bool))),
   structure(
      signals(xBar,bool),
      component(InvX,NOT(input(x),output(xBar))),
      component(AO_0,AO(input(d0,xBar,d1,x),output(dout0))),
      component(AO_1,AO(input(d2,xBar,d3,x),output(dout1))))).
```

Here, the components NOT and AO are atomic modules provided by the MDG software package. Note also that the data sorts in the interface and the internal signals must always be specified. In the above example all signals are of the concrete Boolean sort *bool*.

6.2 RTL Model

The data inputs and outputs of the switch fabric consist of 4 byte-wide lines. In Qudos HDL there is no facility for describing high-level words. Thus the data-in and data-out lines are modeled as 32 individual lines. This could be easily modeled in MDG-HDL using concrete sorts, say an enumeration sort *word8* for 8-bit words. However, they are better described as words of size *n* using abstract sorts, e.g., an abstract sort *wordn*, similarly as the behavioral specification (Section 5.2). Such high-level words are of arbitrary size, thus making descriptions generic with unspecified word sizes. The verification is thus directly valid for switch elements of different word size.

Beside abstracting the data lines from a bundle of bits to a compact word of abstract sort, we have abstracted the behavior of the dataswitch unit by modeling it using simple data multiplexors instead of collections of logic gates. For example, a set of 8 DMUX4T2 modules (as those described in Section 6.1) is modeled using a single mutiplexor component (which is part of a larger module) as follows:

```
signal(dw0,wordn).
signal(dw1,wordn).
signal(dwout,wordn).
signal(x,bool).
component(DMUX4T2_w,MUX(sel(x),inputs([(0,dw0),(1,dw1)]),output(dwout))).
```

The data signals are of abstract sort *wordn*, and the control signal *x* of concrete sort *bool* is the select input of the multiplexer.

We now obtain a much simpler abstract implementation of the dataswitch which reflects the switching behavior in a more understandable way and is implemented with a smaller number of components and signals. Instead of 5 hierarchical levels of modules, we have only 2 levels for describing the data switch which is subdivided into 4 sub-modules that represent 4 independent switching slices. Each slice chooses a word to be output to one of the output ports and consists of three multiplexors and two registers, controlled by using separate grant and disable signals.

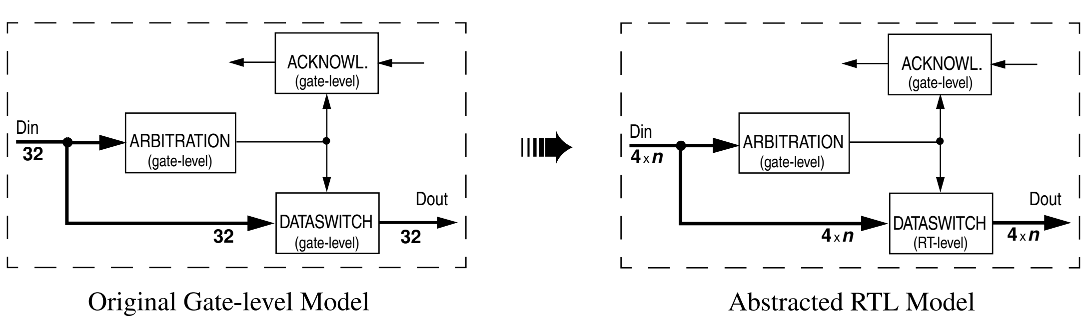


Figure 8: Model abstraction for the switch fabric

Figure 8 shows the abstraction of the switch fabric. The arbitration block now accepts n-bit data of sort *wordn*. It is no longer fed with single bits of the header. Therefore, we must introduce a set of uninterpreted functions (cross-operators) that extract (decode) the fields *active*, *priority* and *route* from the abstract headers (as illustrated in Figure 9). For example, the active bit is obtained using the function *act* of type [*wordn* $\longrightarrow$ *bool*].

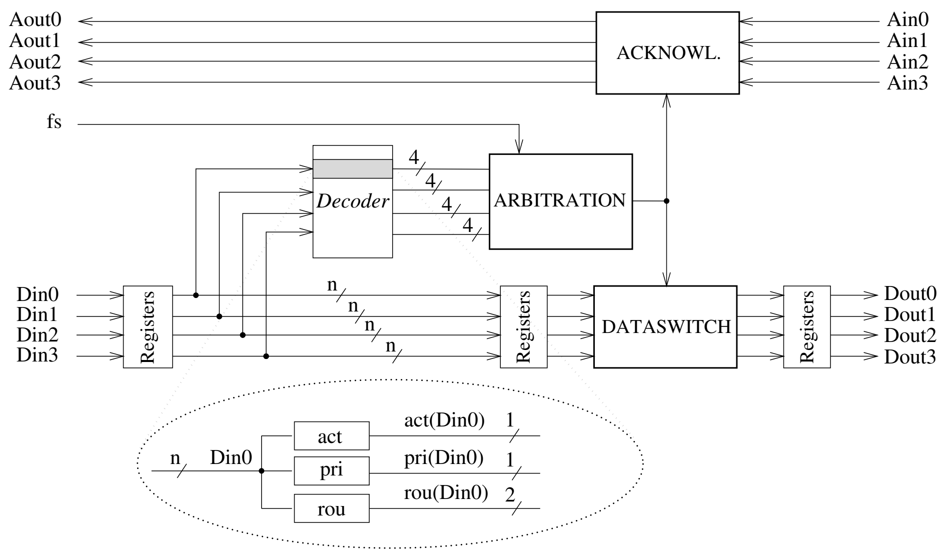
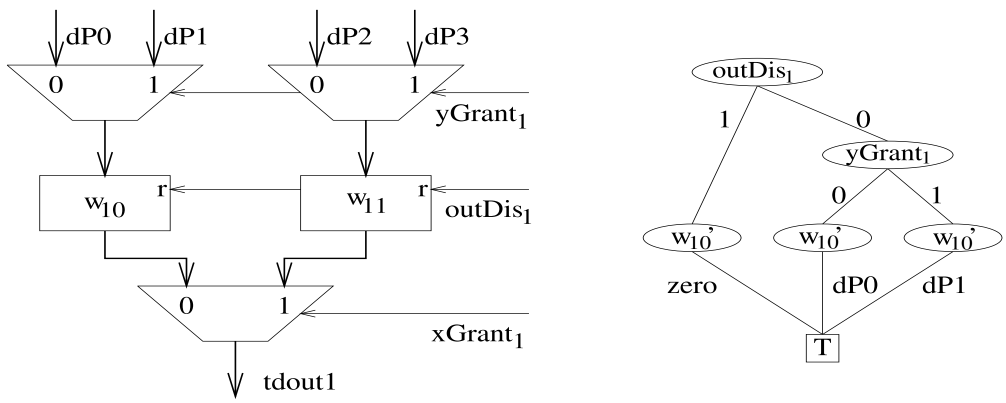


Figure 9: Extraction of control fields of the header

The original MDG-HDL gate-level description of the fabric consists of a network of 504 components connected by 519 signals, while the abstract one uses only 298 components and 307 signals. The state space of this abstract machine (20 abstract plus 30 Boolean state variables) is by far smaller than the gate-level one (162 Boolean state variables). However, if we wish to use this abstract implementation model of the fabric for further experimentation, we should ensure that the two implementations are equivalent. This is discussed in Section 8.

Given the RTL netlist, an ASM model of its behavior is extracted by composing, for each output or register, the MDGs of the components in its cone of influence, and abstracting the internal signals [23] like in the ROBDD case. For instance, Figure 10 (a) shows the RTL netlist of a word-slice of the switch output port 1. The dP_i signals come from the registers at the input of the switch, thus delaying the data inputs by two cycles (Figure 3). The output of the dataswitch is fed into registers before reaching the output of the fabric. The registers inside the dataswitch, e.g., w_{10} and w_{11} in Figure 10 (a), are used to partially compute the output, given the value of $yGrant_i$ (selection between odd or even input ports). The selection is then completed when the value of $xGrant_i$ is known. If *$outDis_i$* is 1, the intermediate registers are forced to *zero*. Given the MDGs of multiplexors and registers with synchronous reset, the MDG of w_{10}' (next value of w_{10}) is shown in Figure 10 (b). The MDGs associated with the other registers and the output signals are obtained in a similar way.



(a) RTL netlist

(b) MDG of signal w_{10}'

Figure 10: RTL netlist and related MDG model for a portion of the dataswitch

7 Validation of the Specification by Property Checking

To formally verify the switch fabric, we investigate the correctness proof of the implementation against the behavioral specification. Before doing so, we must ensure that the specification itself is correct with respect to the Fairisle network environment. Therefore, we first applied property checking to validate that the specification satisfies some specific requirements while working under the control of its environment, i.e., the port controllers. Sample properties are correct priority computation, correct circuit reset and correct data routing. In this section, we describe our techniques for the validation using safety property verification with MDGs.

The environment of the switch fabric generates the frame start signal *fs*, at every 64^{th} rising clock cycle, and resets it in the next clock cycle. Initially, it should wait at least 2 clock cycles to let the fabric reset before it can generate the first *fs* signal. The header must be generated at the 8^{th} rising clock edge after the *fs is* reset to low, i.e., 9 clock cycles after *fs* is set. When the active bit in this header is set, the cell is called active. Otherwise it is inactive (an empty cell). To simplify our presentation, we consider the case for active cells only.

Using the time points t_s, t_h and t_e, as introduced in Section 5.1, we described several properties which reflect the behavior of the switch fabric. These properties are indeed inspired by the toplevel timing specification in [8] and the other design documentation of the Fairisle switch [17]. We shall illustrate our verification technique by the following representative properties:

- *Property 1*: From t_s+3 to t_h+4, the default value (*zero*) appears on the data output port $Dout_0$ where *zero* is a generic constant.
- *Property 2*: From t_s+1 to t_h+2, the default value (0) appears on the acknowledgment output port $Aout_0$.
- *Property 3*: From t_h+5 to t_e+2, if input port 0 chooses output port 0 with the priority bit set in the header, and no other input ports have their priority bits set, the value on $Dout_0$ will be equal to that of Din_0 4 clock cycles earlier.
- *Property 4*: From t_h+3 to t_e, if input port 0 chooses output port 0 with the priority bit set in the header, and no other input ports have the priority bit set, the value on $Aout_0$ will be that of Ain_0.

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 a state machine model inspired by [22]. This is elaborated in the next section to follow.

7.1 Representation of Properties using MDGs

The cyclic behavior of the port controller can be simulated by an *environment state machine* having 68 states as shown in Figure 11. The machine generates in specific states the frame start signal *fs*, the headers, denoted as *h*, and the data, denoted as *d*, in the states as indicated in the figure. Acknowledgments from the output port controllers are available at every state. Also the first *fs* signal is generated at the 3rd clock cycle after power on. States 1 to 5 are related to the initialization of the fabric.

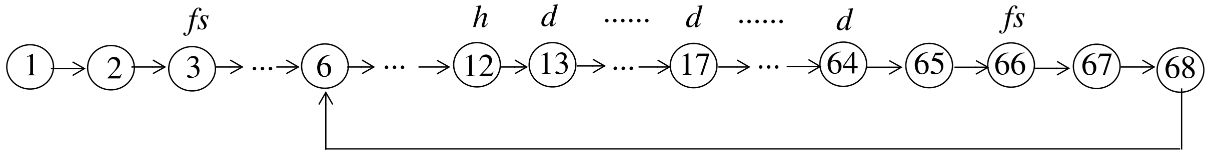


Figure 11: Environment state machine

States 6 to 68 represent the cyclic behavior of the fabric, where one cycle corresponds to one frame. With this diagram, we can map the time points t_s, t_h and t_e to states. In this case, t_s=3 or 66; t_h=12; and t_e=66. Then, e.g., t_h+5 to t_e+2 are the states between 17 and 68 when the remaining bytes of the cell following the header are switched to the output port. Consequently, we can express the properties in terms of states rather than time points. It can be checked that this state machine is an instance of the general timing state machine (Figure 5) with cell length of 53 and frame size of 64.

The environment state machine is defined using a state variable *s* of concrete sort having the enumeration [1..68]. We now restate the previous properties as invariants using ITE (If-Then-Else) formulas of the Prolog-style MDG-HDL.

- *Property 1*: **If** ($s \in$ [6,...,16]) **then** $Dout_0$ = *zero* **else** don't care,
- *Property 2*: **If** ($s \in$ [4,...,14, 67, 68]) **then** $Aout_0$ = 0 **else** don't care,
- *Property 3*: **If** ($s \in$ [17,...,68]) **and** *priority*[0..3] = [1,0,0,0] **and** *route*[0] = 0 **then** $Dout_0$ = Din_0' **else** don't care,
- *Property 4*: **If** ($s \in$ [15,...,66]) **and** *priority*[0..3] = [1,0,0,0] **and** *route*[0] = 0 **then** $Aout_0$ = Ain_0 **else** don't care,

where Din_0' is the input of Din_0 4 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). These invariants can be easily represented using MDGs. For example, Figure 12 (a) and (b) show the MDGs for *Property 1* and *Property 2*, respectively. The edge label *x* issued from $Dout_0$ is an abstract variable disjoint with all other variables and represents the *don't care* value (in the property it is implicitly existentially quantified as the expression defines a set of states).

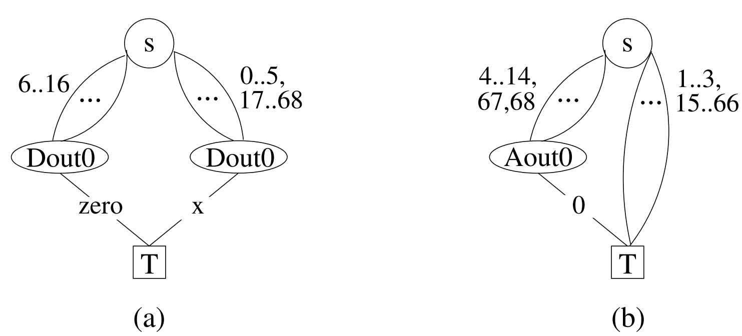


Figure 12: MDGs of *Property 1* and *Property 2*

For the verification of these invariants, we compose the fabric with the environment state machine as shown in Figure 13. As there is a 4-clock-cycle delay for the cells to reach the output ports, a delay circuit is used to remember the input values that are to be compared with the outputs. Hence, we can state the properties in terms of the equality between Din_0' and $Dout_0$ (e.g., *Property 3*). By combining these machines (the dashed frame in Figure 13), we obtain the required platform for checking the invariants.

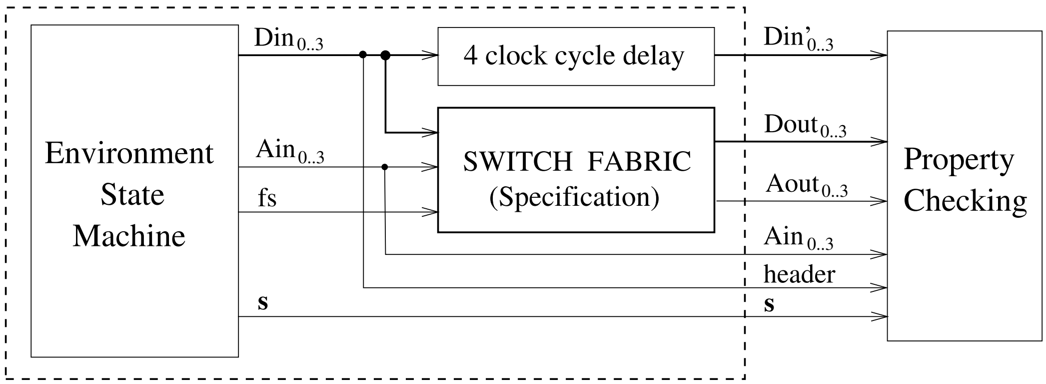


Figure 13: Composed state machine for property checking

By exploring the state space of the combined machine, at each reachable state, we check if the outputs satisfy the invariant. The experimental results are given in the next section. In another work [21], we applied the above property checking techniques to verify the Fairisle ATM switch fabric RTL implementation under the conditions of its operating environment. There, we also used the same set of properties to uncover several design errors that we intentionally injected in the RTL netlist model.

7.2 Experimental Results

We report the experimental results of checking the properties described in Section 7.1 (*Property 1* to *Property 4*) which reflect the essential behavior of the switch for one output port. The composed machine (Figure 13) for one port has 53 state variables (27 abstract variables and 26 concrete variables which are equivalent to 32 Boolean variables). The experimental results are shown in Table 1, including the CPU time in seconds, the memory usage in megabytes, the number of consecutive state transitions and the number of MDG nodes generated. The experiments were done on a SPARC station 20 with 128 MB of memory.

Table 1: Experimental results of property checking

Verification	CPU time (in sec)	Memory (in MB)	Transitions	Nodes generated
Property 1	486	40	68	90908
Property 2	890	57	68	113666
Property 3	344	37	68	83985
Property 4	977	57	68	123423

8 Verification by Equivalence Checking

Our primary verification goal is to show that the original gate-level implementation of the switch fabric complies with the specification of the expected behavior. Since the implementation at the gate level is too big to be handled (verified) at once, we use the introduced abstract RTL description to close the semantic gap between the implementation and the specification, and perform the verification hierarchically in two steps. (1) We verify that the RTL description and the behavior specification show an equivalent behavior for an arbitrary word width n and any frame size and cell length that respect the environment assumptions of the specification. (2) We verify that the RTL model description is equivalent to the original gate-level implementation for words of width n = 8, where the n-bit words of abstract sort are aligned with 8-bit words of concrete sort. In the following sections, we will elaborate on each of these steps. The verification was achieved automatically in a reasonable amount of CPU time, as will be shown in a separate section on experimental results. In addition, we also verified several faulty implementations where the introduced errors were successfully identified using the counterexample facility of the MDG tools.

8.1 RTL vs. Behavior Model Verification

To verify the RTL implementation against the ASM of the behavioral specification, we make use of the fact that the corresponding input/output signals are of the same sort and that the same function symbols which extract the control information (*active*, *priority* and *route* fields) from the header are used in both descriptions. The two machines are behaviorally equivalent if and only if they produce the same output values for all input sequences. Using MDG-based reachability analysis, verification is done for an arbitrary word width n and any frame size and cell length that respect the environment assumptions of the specification.

For the product of the ASMs from Sections 5.2 and 6.2, an MDG representing a set of total-states encodes a relation between 39 concrete and 36 abstract state variables. The relation may depend on data values, extracted using cross-terms. For instance, parts of the MDG encoding the set of total states reached at the 9th iteration of the reachability analysis (i.e., all the states reachable one transition after arbitration) are depicted in Figure 14.

In ROBDDs, 8 Boolean variables would be needed for each abstract variable of the MDGs (i.e., 288 Boolean variables for data). Since many non-disjoint combinations of data state variables have the same values in the set of reachable states, it is not possible to interleave these Boolean variables to avoid explosion caused by the binary encoding. In MDGs, the encoding is done using abstract data, yet isomorphic graph sharing is exploited as in ROBDDs (e.g., the left hand side of Figure 14, where the variables d_{ij} represent the values of Din_i at iteration j). Decisions on values of abstract data are represented by cross-terms which also contribute nodes in the MDGs (e.g., $act(d_{06})$ in Figure 14). Although cross-terms add complexity to the graph structure in general, the overhead is much smaller than the explosion induced from encoding data in binary form.

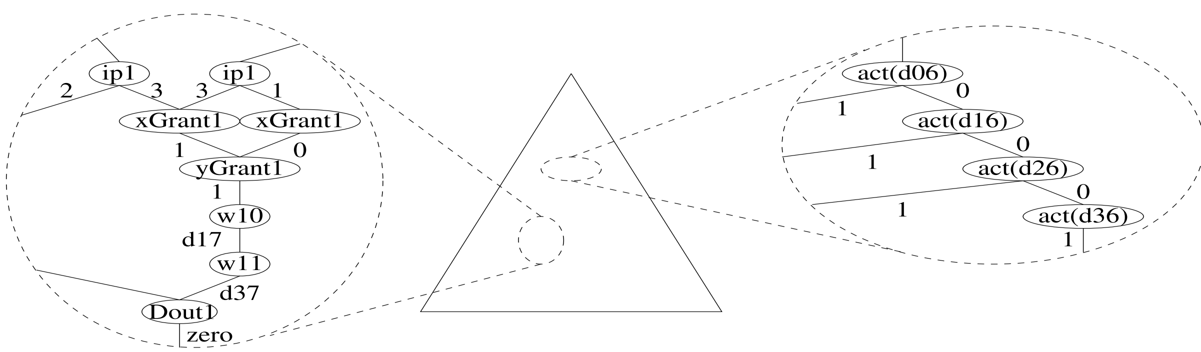


Figure 14: MDG encoding a set of reachable abstract states

8.2 Gate-level vs. RTL Model Verification

The correctness of equivalent behavior between the gate-level implementation and the abstract RTL model cannot be established for an arbitrary word size n since the gate-level description is not generic. As mentioned above, we should somehow "instantiate" the data abstract signals of the RTL model to 8 bits. This can be realized within the MDG environment using *uninterpreted functions* which encode and decode abstract data to Boolean data and vice-versa. For instance, *decoding* can be realized using 8 uninterpreted functions (cross-operators) bit^i (i: 0..7) of type [$wordn \rightarrow bool$], which extract the i^{th} bit of an n-bit data and hence encode each n-bit data line to an 8-bit bundle. Reverse *encoding* is done using one uninterpreted (abstract) function *concat8* of type [(*bool* x *bool* x *bool* x *bool* x *bool* x *bool* x *bool* x *bool*) $\rightarrow$ *wordn*] which concatenates any 8 Boolean signals to a single word and thus encodes each 8-bit bundle to a signal of sort *wordn*.

Based on this technique, four symmetric configurations are possible for performing this verification step. This is illustrated in Figure 15 where only the data inputs and outputs of the fabric are considered, since the abstraction only affects the dataswitch block (Figure 8). In all of these cases, we ensure that we feed the two machines with inputs of the same sort and check the equivalence of outputs of the same sort. The encoding and decoding blocks (as represented in Figure 15) are not functional blocks that we add to the implementation description but illustrate the uninterpreted functions that we insert into the invariant during equivalence verification. Any one of these configurations could be used in the verification, however, we noticed that case (d) in Figure 15 is the least expensive. This is because it avoids the use of the uninterpreted function concat8 which requires a full encoding of the 8-bit Boolean vector at the abstract level, i.e., for each Boolean vector which could be represented by the 8 bits, we have to provide a fixed correspondence to a specific constant of abstract sort, e.g., (0,0,0,0,0,0,0,0) $\leftrightarrow$ *zero*, (0,0,0,0,0,0,1,0) $\leftrightarrow$ *two*, where *zero* and *two* are generic constants of abstract sort *wordn*.

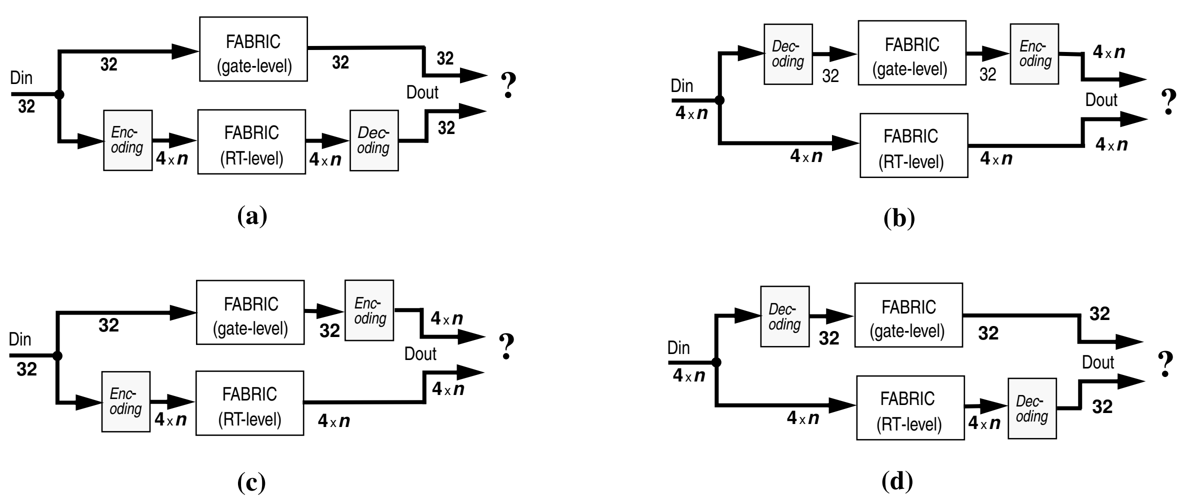


Figure 15: Different configurations for the RTL model verification

Since the data abstraction affects only the dataswitch block (see Figure 8), the verification mainly dealt with the equivalence of the dataswitch blocks at the two levels. The verification run time for the (best) variant (d) in Figure 15 is given in Table 3.

8.3 Experimental Results and Discussion

First, we report with some detail on the experimental results of comparing the ASM specification of the switch fabric against the RTL model using MDGs. The experiments were done on a SPARC station 20 with 128 MB of memory. Table 2 shows the size of some typical MDGs generated during the reachability analysis, the number of MDG paths and the CPU times for constructing them.

Table 2: Size and generation time of some MDGs

MDG	Number of Nodes	Number of Paths	CPU time (sec.)
q_0	71	256	1
q_7	8321	1675264	80
v_7	8572	1677824	5
q_8	11665	128128	150
v_8	20225	1805952	10
q_9	25985	256256	300
v_9	46160	2062208	10
q_{10}	4995	60360	1100
v_{10}	51106	2122568	60
q_{11}	1	1	60

The MDG q_j is the frontier set of states at the end of iteration j, while v_j is the set of all reached states. q_0 is the MDG encoding the initial total states. There are 256 paths in the MDGs because the initial values of each ip_i can vary. Note that, due to subgraph sharing, all these paths are encoded using only 71 nodes. The MDG q_7 is the frontier set of states after the first arbitration phase in the implementation (i.e., at $t_h + 2$), while q_8 encodes those after the arbitration is completed in both machines. q_9 and q_{10} encode states where the frame may be terminated. q_{11} is the final false MDG of the frontier set (representing the empty set) meaning that all reachable states have been visited. The output of the two machines were compared at each iteration, which took up to 100 sec. in some cases. No difference between the two ASMs were detected.

Parallel to the verification of the RTL against the behavioral specification, we verified the gate-level implementation against the abstract RTL model, where the n-bit words of the RTL model are aligned with 8 bits. Note that the state variables of the two machines had to be initialized with the corresponding values, i.e., constants *zero* of sort *wordn* and 0's of sort Boolean for the abstract and gate-level models, respectively. In this case, we used several rewriting rules stating that all bits of *zero* are equal to the Boolean 0, i.e., $bit^i(zero) = 0$, i=0..7. The experimental results are summarized in Table 3.

By combining the above two verification steps, we hierarchically obtained a complete verification of the switch fabric from high-level behavior down to gate-level implementation. The experimental results on a SPARC station 20 are recapitulated in Table 3, including the CPU time, the memory usage and the number of MDG nodes generated. It is apparent that in the first verification step a large set of states are encoded. Because of MDG's abstract representation and graph sharing, the encoding resulted in an acceptable number of nodes, however. This supports the conclusion that the verification would be impossible if ROBDD representation in the reachability analysis were used (cf. Section 8.4). The verification of the RTL model against the gate level was less CPU time and memory consuming, as it was reduced to comparing only the dataswitch block outputs of both machines.

Table 3: Experimental results of equivalence checking

Verification step	CPU time (sec.)	Memory (MB)	Number of Nodes
RTL vs. Beh. Spec	2920	150	320556
Gate-level vs. RTL	183	22	183300

Our verification confirms the results obtained by Curzon using HOL [8], that no errors were discovered in the implementation of the switching element. In fact, the fabric was extensively simulated (debugged) and has been in service for some time before it was verified by formal methods. To test the effectiveness of our approach, we experimented with three erroneous implementations: (1) We exchanged the inputs to the JK flip-flop that produces the $outDis_3$ signal. This prevented the circuit from resetting. (2) We used the priority information of input port 0 instead of input port 2. (3) We used an AND gate instead of an OR gate within the acknowledgment unit producing a faulty $Aout_0$ signal. These three errors were detected by verifying the RTL implementation model against the behavioral specification. In each case, a counterexample was generated to help with diagnosing the errors. Table 4 shows the results of these three experiments, including CPU time for performing the reachability analysis and for generating counterexamples, the memory usage and the number of MDG nodes generated.

Table 4: Verification of faulty implementations

Case	Reach. Anal. (sec)	Counterexample (sec)	Memory (MB)	Number of Nodes
Error 1	11	9	1	2462
Error 2	850	450	120	150904
Error 3	600	400	105	147339

8.4 Comparison with Verification in HOL and in VIS

Similarly to Curzon's work [8] using the HOL theorem prover, our MDG-based verification is generic, since it holds for an arbitrary data word size. This was not possible in the verification done by Lu et. al [18] using an ROBDD-based verifier, such as VIS. Furthermore, while Curzon's behavioral description exploits the powerful expressiveness of HOL to describe the behavior at the frame level, our specification is based on a state machine model similar to the descriptions in VIS. This latter, however, was written in Verilog in contrast to our MDG-HDL, thus allowing the direct test of the specifications using commercial simulation facilities. Unlike Verilog descriptions, Curzon's higher-order logic specifications as well as our MDG-HDL descriptions are not directly executable. We impose two more assumptions than Curzon on the environment of the switch fabric (the second and the third one in Section 5). These assumptions are taken from the informal description of the switch fabric, and permit to reduce the complexity of our automatic verification by a factor of 2. Similar assumptions have been adopted in [18] for the same reason.

In Curzon's verification with HOL, much work was needed in proving a large number of lemmas and in setting-up the proof scripts interactively, e.g., the time spent for the verification of the dataswitch unit was about one week [8]. The related proof script was about 530 lines long (17 KB). Using our state machine models and MDGs, the verification was achieved automatically without the need of any proof script, except the careful management of node ordering (which so far has to be done manually). The VIS verification was also conducted in an automated fashion. In addition, VIS provides several options for dynamic variable ordering. Major effort, however, was spent developing abstract models of the switch fabric units to manage the state explosion of the Boolean representation. Notwithstanding, while the HOL and MDG verifications succeeded in verifying the whole switch fabric, VIS failed in verifying even the smallest 4-bit datapath version of the fabric using equivalence checking.

Curzon reported in [9] that the detection and correction of errors in one erroneous 4 by 4 switch fabric using HOL took 3 man-months and consumed a large portion of the verification time. This is generally the case when the verification of faulty implementations is done using a theorem prover. In both MDG and VIS errors are detected automatically and can be diagnosed with the help of the counterexample facility. In addition, due to its Verilog front-end, VIS generated counterexamples can be analyzed using commercial tools such as Cadence XL-Verilog. We are in the course of releasing a VHDL to MDG-HDL translator which makes a small subset of VHDL models acceptable to the MDG tools.

For property checking in both MDG and VIS, it was necessary to introduce an environment state machine in order to restrict the possible inputs to the switch fabric. In addition, to cope with the state-space explosion of ROBDDs, the VIS work adopted several techniques to manage the expression and verification of properties described in CTL. In MDG tools, no facility is provided for CTL properties specification and no liveness properties can be verified in the current version of the MDG software package. There is, however, on-going work on the development of an MDG model checking algorithm based on a restricted first-order temporal logic (Abstract CTL—ACTL).

More details on the comparison of HOL and MDG for hardware verification using the Fairisle switch fabric as a case study can be found in [20].

9 Conclusions

In this paper, we have shown the applicability of formal verification techniques based on a new class of decision graphs, Multiway Decision Graphs (MDGs), to a realistic circuit—the Fairisle ATM switch fabric. This is much larger than any other circuit so far verified using MDGs. We provided MDG models of the fabric at different levels of the design hierarchy: high-level behavior, abstract RTL and gate-level implementations. In order to gain confidence on the developed specification model, we validated it by checking a set of safety properties that reflect the behavior of the fabric when used in the Fairisle ATM switching network environment. Using a hierarchical approach, we first verified the equivalence checking of the RTL hardware implementation of the Fairisle ATM switch fabric against the high-level behavioral specification with no restrictions on the frame size, the cell length and the word width. We then investigated the equivalence of the original gate-level implementation of the switch fabric against the RTL description model in which the generic words of abstract sort were aligned with 8-bit words. The verifications were based on the reachability analysis of the product machine of the implementation and the specification both given as Abstract State Machines (ASM). We found no errors in the current implementation. However, we verified several faulty implementations with injected errors which were successfully identified. The results achieved in the present work illustrate the practicability of such a complete formal verification down to the gate level using tools exploiting MDGs, a methodology that would be impossible in this case using only ROBDD-based reachability analysis. We have demonstrated that formal verification of a real piece of communication hardware can be accomplished automatically using the MDG tools.

ROBDD-based symbolic model checking is widely used for the verification of safety as well as liveness properties. However, since it requires a Boolean representation of the circuit, it must cope with the state explosion problem when the number of variables is large. MDGs have the ability of representing a data value with a single variable of abstract sort and hence are generally more compact than ROBDDs. Moreover, invariant checking on an ASM model is powerful, since it can take advantage of information contained in the symbolic terms. In comparison with ROBDD based verification methods, we used only one abstract variable instead of 8 Boolean variables for representing data registers. In the Fairisle example, the header of a cell contains 4 bits of control information. It is not possible to reduce the datapath width to less than 4 bits using a data path reduction technique (as confirmed in [18]). In general, if the control needs n bits, then it is impossible to reduce the datapath width to less than n. Thus, for datapaths containing mixed control/data information, ROBDD based datapath reduction technique is not efficiently applicable. On the other hand, using our MDG approach, we naturally allow the abstract representation of data, while the control information is extracted from the datapath using uninterpreted functions (cross-operators).

In [8] it is reported that the time spent on simulation would have been in the order of several weeks. However, errors were discovered after the testing process was completed when the first version of the fabric was in use. Had formal verification been applied to the ancestors of the actual design, it could have possibly discovered the errors and then validated any corrections.

Acknowledgments

We are grateful to Paul Curzon at Middlesex University, UK, and Francisco Corella at Hewlett-Packard Company, USA, for advocating this study. This work was partially funded by NSERC-Nortel Cooperative Research Grant CRD No. 191958 and NSERC research grant No. OGP014302.

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, pp. 677-691, August 1986.

[3] J. Burch, E. Clarke, D. Long, K. McMillan, and D. Dill, "Symbolic Model Checking for Sequential circuit Verification," IEEE Transactions on Computer-Aided Design, vol. 13, pp. 401-424, April 1994.

[4] E. Cerny, F. Corella, M. Langevin, X. Song, S. Tahar, and Z. Zhou, "Automated Verification with Abstract State Machines Using Multiway Decision Graphs," In: T. Kropf (editor), Formal Hardware Verification: Methods and Systems in Comparison, Lecture Notes in Computer Science 1287, State-of-the-Art Survey, Springer Verlag, 1997, pp. 79-113.

[5] B. 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, UK, pp. 132-136, June 1994.

[6] 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.

[7] F. Corella, M. Langevin, E. Cerny, Z. Zhou, and X. Song, "State 'Enumeration with Abstract Descriptions of State Machines," In Proc. IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'95). Frankfurt, Germany, October 1995.

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

[9] P. Curzon, "Problems Encountered in the Machine-assisted Proof of Hardware," In: J. Joyce, and C. Seger (editors), Higher Order Logic Theorem Proving and Its Applications, Lecture Notes in Computer Science 780, Springer Verlag, 1994.

[10] K. Edgcombe, "The Qudos Quick Chip User Guide," Qudos Limited.

[11] E. Garcez and W. Rosenstiel, "The Verification of an ATM Switching Fabric using the HSIS Tool," In the IX Brazilian Symposium on the Design of Integrated Circuits, 1996.

[12] D. Ginsburg, "ATM Solutions for Enterprise Internetworking,"Addison Wesley, 1996

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

[14] A. Gupta, "Formal Hardware Verification Methods: A Survey," Formal Methods in System Design, vol. 1, pp. 151-238, 1992.

[15] L. Jakubiec, S. Coupet-Grimal, and P. Curzon, "A Comparison of the Coq and HOL Proof Systems for Specifying Hardware," In E. Gunter and A. Felty (editors), International Conference on Theorem Proving in Higher Order Logics: B-Track: Short Presentations, pp 63-78, August 1997.

[16] M. McMillan, "Symbolic Model Checking," Kluwer Academic Publishers, Boston, Massachusetts, 1993.

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

[18] 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 (GLS-VLSI'98), Lafayette, Louisiana, USA, February 1998.

[19] K. Schneider and T. Kropf. "Verifying Hardware Correctness by Combining Theorem Proving and Model Checking," In J. Alves-Foss (editor), International Workshop on Higher Order Logic Theorem Proving and Its Applications: B-Track: Short Presentations, pp. 89-104, August 1995.

[20] S. Tahar and P. Curzon, "A Comparison of MDG and HOL for Hardware Verification," In: J. von Wright, J. Grundy, and J. Harrison (editors), Theorem Proving in Higher Order Logics, Lecture Notes in Computer Science 1125, Springer Verlag, 1996, pp. 415-430.

[21] S. Tahar, Z. Zhou, X. Song, E. Cerny, and M. Langevin, "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, pp. 106-111, March 1996.

[22] G. Thuau and B. Berkane, "A Unified Framework for Describing and Verifying Hardware Synchronous Sequential Systems," Formal Methods in System Design, vol. 2, pp 259-276, 1993.

[23] Z. Zhou, X. Song, F. Corella, E. Cerny, and M. Langevin, "Description and Verification of RTL Designs using Multiway Decision Graphs," Proc. Conference on Computer Hardware Description Languages and their applications (CHDL'95), Chiba, Japan, August 1995.