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

Sofiène Tahar[1], Xiaoyu Song[2], Eduard Cerny[2],
Zijian Zhou[3], Michel Langevin[4] and Otmane Aït-Mohamed[5]

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

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

[3] Texas Instruments, Inc., Dallas, TX 75266-0199, USA
E-mail: zzhou@ti.com

[4] Nortel Technologies, Ottawa, Ontario, K1Y 4H7 Canada
E-mail: mlange@nortelnetworks.com

[5] Cistel Technology, Inc., Nepean, Ontario, K2E 7L5 Canada
E-mail: otmane@cistel.com

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. 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 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 [19].

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) [17]. 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) [11]. These decision graphs subsume the class of Bryant's Reduced Ordered Binary Decision Diagrams (ROBDD) [5] 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 [21] 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 behavior of the ATM switch fabric, the modeling of its implementation at the gate level and a more abstract RT level, and the successful equivalence checking between the different levels. The behavioral specification and thus the higher level verification have no restrictions on the frame size, cell length or word width. Furthermore, we were able to validate our specification by verifying safety properties which reflect the behavior of the fabric in the Fairisle ATM environment (being a synchronous design with known synchronous delays, we could convert bounded liveness properties to safety properties verification). In addition, we verified several implementations with introduced errors; they were successfully identified by the counterexample facility in the MDG tools.

The verification is based on the reachability analysis of the product machine of the implementation and the specification, each modeled as networks of Abstract State Machines (ASM) [10]. 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 automatically in a reasonable amount of CPU time. Manual intervention was needed only in the identification of state variables and input/output signals that were to be assigned abstract sorts, as well as for variable ordering. This experiment illustrates the effectiveness of the MDG-based verification methodology as an extension of ROBDD-based approaches [5], since model explosion induced by data values expanded to their binary representation is largely avoided. Our results show that the MDG-based verification method can be successfully applied 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 (MDGs) 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 an 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 approach adopted to validate

the behavioral specification using safety property checking. In Section 8, we explore equivalence verification between the different abstraction 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. Finally in Section 9, we draw some 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. [9] exploited symbolic model checking to detect a design error in an ATM circuit. The 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) [20], they identified the error by checking some properties described in CTL (Computational Tree Logic) [20]. 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.

Curzon [13] verified the 4 by 4 fabric of the Fairisle switch fabric using the HOL theorem prover [18]. (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, highlighting the fact that a correct specification could be just as hard to develop as an implementation.

More recently, Lu *et. al* [22] used the VIS tool [4] to verify relevant liveness and safety properties (described in CTL) on various abstracted models of the Fairisle 4 by 4 switch fabric. In order to cope with the state explosion problem, the authors used several compositional reasoning techniques for properties division and had to adopt a number of model reduction and abstraction approaches. In addition, they conducted equivalence checking between the behavioral and structural specifications of the sub-modules written in Verilog. However, due to state space explosion, they did not succeed the equivalence checking on the whole fabric. The authors also re-implemented the fabric using the Synopsys synthesis tool.

Other researchers have also used the Fairisle switch fabric as a case study. For instance, Schneider *et. al* [23] formally verified it using the verification system MEPHISTO which is based on the HOL theorem prover. They described the structure of each of the modules used in the design hierarchically and provided their behavioral specifications using hardware formulas [23]. Although they automated the verification of lower-level hardware submodules, they have not accomplished the complete verification of the implementation against the intended overall behavior of the switch fabric.

Garcez [16] has also verified some properties on the implementation of the fabric using the HSIS model checking tool [3]. The author described the netlist implementation of the ATM switch

fabric using a subset of Verilog, and checked properties on submodules of the fabric using model checking and/or language containment. No model checking on the whole switch fabric model, nor a verification against a high-level specification is reported, however. Moreover, in some cases a slightly different implementation of a module is described in order to ease the verification.

The work of Schneider and Garcez was limited to the verification of submodules of the gate-level implementation and did not provide a proof of correctness for the whole switch fabric. They will, therefore, not be investigated further in this paper. Rather, a comparison of our approach with the work done by Curzon, Lu and Chen will be elaborated in the following as these address the verification of the complete fabric.

Although Curzon [13] 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 [19]. In contrast, the ATM verification performed by Lu *et. al* [22] using VIS was automatic. While succeeding with model checking reduced models of the whole fabric and with equivalence checking of submodules of the design hierarchy, due to state space explosion, the VIS tool failed to complete the equivalence checking of even a very reduced model of the fabric.

The work at Fujitsu Ltd. [9] used the ROBDD-based SMV model checker and the authors succeeded in checking some important properties related to the circuit implementation. Yet, to avoid state explosion, the adopted data abstraction (e.g., using 1 bit to represent 8-bit data width) was not quite adequate, because it required judgment to select the right address width, etc. This in turn increased the chances that such a change could mask an error in the original design.

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 [10]. 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) [5]. Our method, through the use of abstract data, is a generalization of the design rather than a simplification, and as such it cannot mask errors. It can, however, produce false negative answers when uninterpreted function symbols are used and the design is dependent on a specific interpretation. In the next section, we briefly describe MDGs and the associated verification methods. For more details about MDGs see [10, 27].

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) [11], 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 [12] from the Boolean level to the abstract level, and refer to it as *implicit abstract enumeration* [10]. 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 removing the already visited states from the set of newly reached states using the pruning-by-subsumption (*PbyS*) operation. If the frontier-set of states is empty, then the reachability analysis procedure terminates, since there are no more unexplored states. Otherwise, the newly reached states are merged (using disjunction) with the already visited states and the procedure continues where 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 [10] which turns MDGs into a canonical representation. This is not unfortunately of much use in the reachability analysis procedure, because the description of the sets of states involves an implicit existential quantification over abstract variables which removes the canonicity property. See Sections 5.2 and 6.2 for examples of MDGs representing various models.

Algorithms for computing *disjunction*, *relational product* (*conjunction* followed by *existential quantification* [7]), *pruning-by-subsumption* (*PbyS*, for test of set inclusion) and *reachability analysis* (using abstract implicit enumeration) have been developed and implemented in Quintus Prolog in the MDG software package. Except for *PbyS*, the operations are a generalization to first-order terms of algorithms on ROBDD, with some restrictions on the appearance of abstract variables in the arguments [10]. Since in the underlying logic of MDG there is no complement of expressions involving equality over abstract terms, *PbyS* approximates the relative complement between two formulas *P* and *Q*, by removing from *P* those MDG paths (conjuncts) that are subsumed by some paths in *Q*. Namely, if *R=PbyS(P,Q)*, then $\models R \vee (\exists U)Q \Leftrightarrow P \vee (\exists U)Q$ [10].

In addition to the logic operations, we have also included a facility to carry out simple rewriting of terms that appear in MDGs. This allows us to provide a partial interpretation to (some) of the uninterpreted function symbols. For example, if *zero* is an abstract generic constant of sort *wordn* and *eqz(x)* a cross-operator of type [*wordn→bool*], then we could provide a partial interpretation of *eqz* using the rewrite rule *eqz(zero)→1* indicating that *equal-to-zero* is 1 when supplied with the argument *zero* (but not saying anything about the other values). User selected rewrite rules are applied any time when a new term is formed during MDG operations. In general, rewriting simplifies MDGs and helps remove false negative answers during safety property checking, thus likely avoiding nontermination of the reachability analysis procedure for designs that depend on interpretation of operators for correct operation. A detailed description of the operations and algorithms can be found in [10]; some possible solutions to the nontermination problem are addressed in [1], [2] and [28].

Based on these algorithms, the following verification procedures are provided in the MDG software package:

Combinational verification: The MDGs representing the input-output relation of each circuit are computed using the relational product of the MDGs of the components of the circuits. Then, taking advantage of the canonicity of MDGs [10], it is verified whether the two MDG graphs are isomorphic.

Safety properties checking: Using symbolic reachability analysis, the state space of a given sequential circuit (an abstract state machine) is explored in each state. It is verified that the specified property - a logic expression - is satisfied (i.e., it is an invariant over the reachable state space). 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 which are themselves abstract state machines. In other words, the relational product computes the (synchronous) product machine of the component ASMs.

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. This is achieved by forming a circuit consisting of the two circuits, feeding the same inputs to both of them, and verifying an invariant asserting the equality of the corresponding outputs in all reachable states.

Counterexample generation: When invariant checking fails, the MDG tools generate a counterexample to help with identifying the source of the error. A counterexample consists of a list of assumptions, input and state values in each clock cycle, which provides a trace leading from the initial state to the faulty behavior.

These techniques were used for the verification of a set of known (combinational and sequential) benchmark circuits [8]. We report here on the verification of the Fairisle ATM switch fabric which is 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.

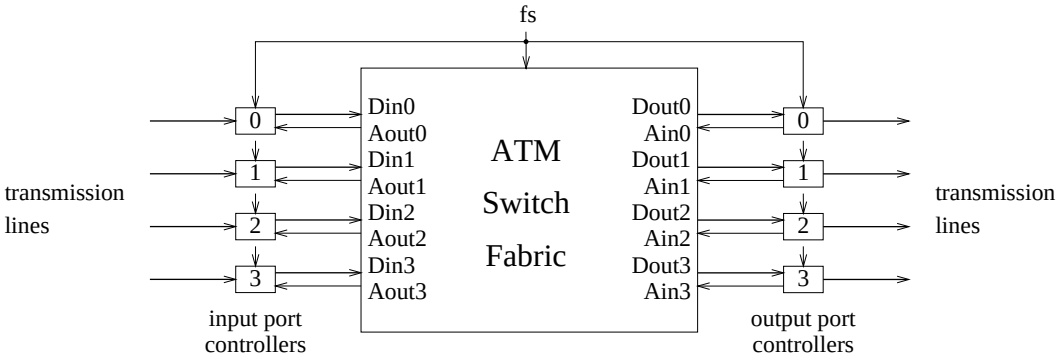


Figure 1: The Fairisle ATM switch

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 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* (*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.

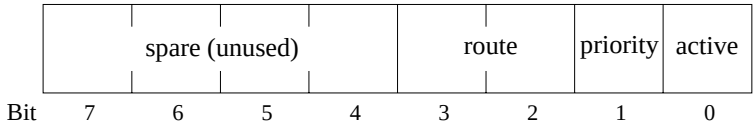


Figure 2: Header of a Fairisle ATM cell

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 is 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.

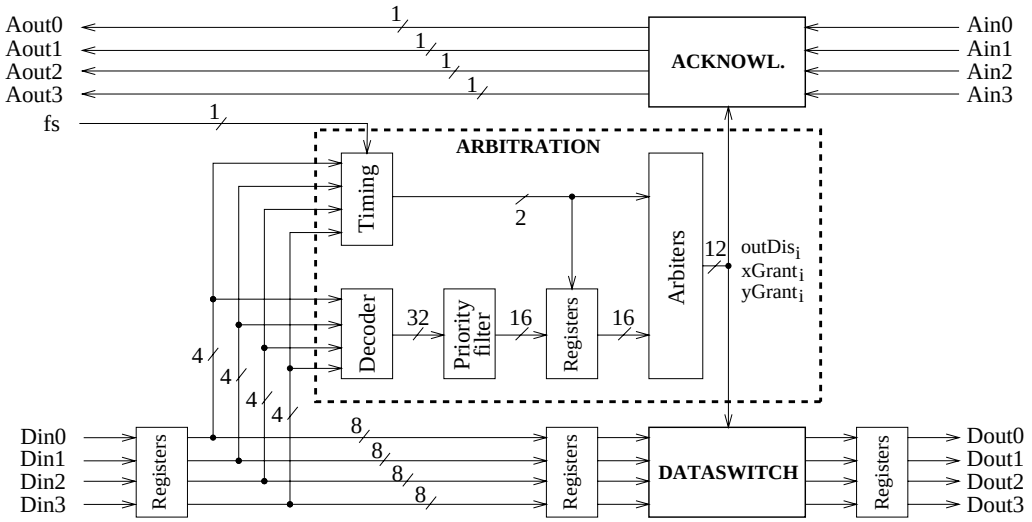


Figure 3: Block diagram of the Fairisle switch fabric

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 output port) make arbitration decisions *(*when two or more cells have the same destination), pass the results to the other modules and control the timing of the other units (Figure 3). The dataswitch unit performs the actual switching of data from input ports to output ports according to the most recent arbitration decision. 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).

5 Specification of the Switch Fabric

Inspired by the design documentation of the Fairisle switch fabric, we derived a 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 behavior of the switch fabric in the form of a state machine and then derive the corresponding MDG model.

5.1 The Behavior of the Switch Fabric

A timing-diagram of the expected input-output behavior of the switch fabric during an active frame is shown 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 any 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 *Din* are *don't care*).

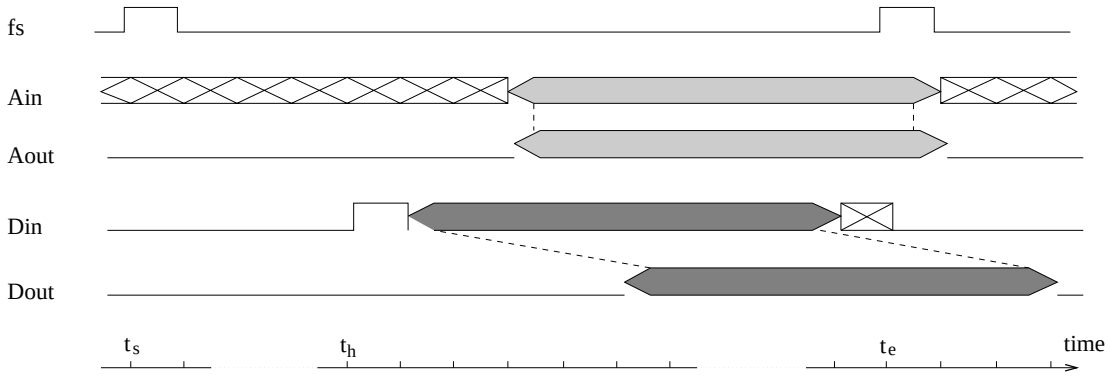


Figure 4: Timing diagram behavior in an active frame

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 ($t'_s=t_e$) may arrive at the earliest 3 cycles after the current frame start (t_s), i.e., $t_e > t_s+2$

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 was carried out.

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 produced.

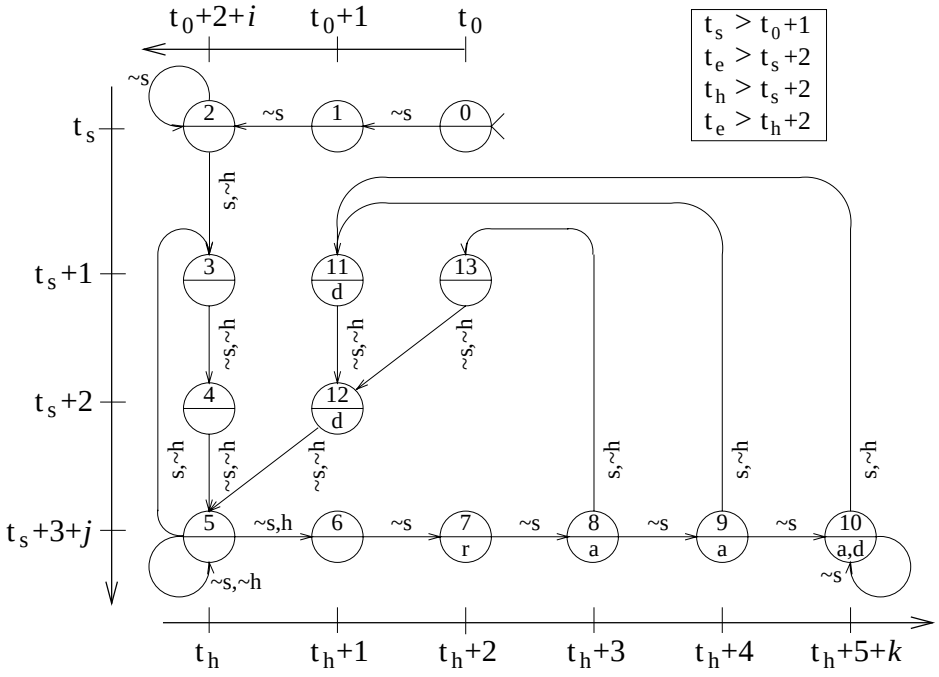


Figure 5: ASM of the switch fabric behavior

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*. The arrival of a next frame start at this point corresponds to the end of an empty frame. Note also that in the case where no headers arrive, the third and fourth environment constraints do not apply to the current 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.
- 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).
- $sr_{i,j}$ (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 MDG Modeling of the Fabric Behavior

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 an ASM. We use a Prolog-style HDL, called 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 behavior MDG 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)

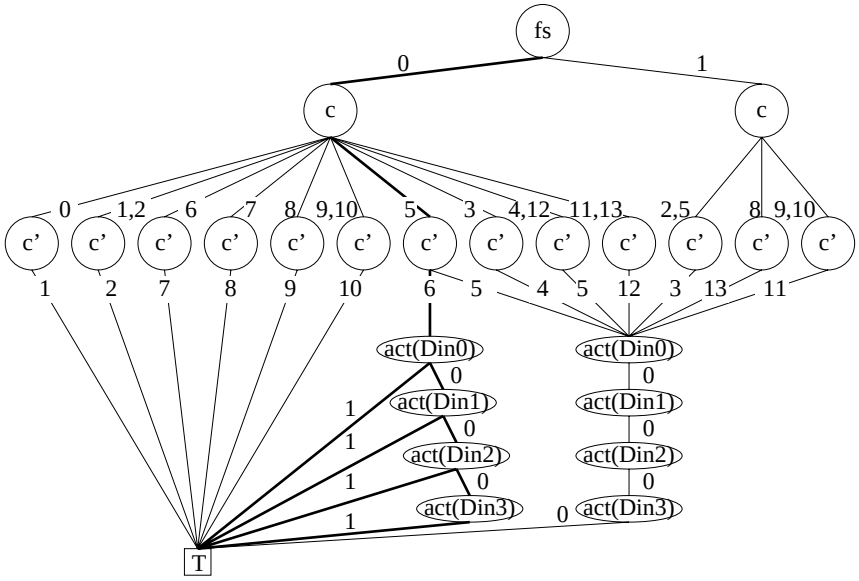


Figure 6: The MDG associated with next-state *c'*

The variables used are:

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.*

We thus constructed an MDG model of the fabric behavioral specification consisting of 16 abstract state variables ($sr_{i,j}$) and 9 concrete state variables (c, co_i and ip_i). Note that the 9 concrete variables are equivalent to 16 Boolean variables if a Boolean encoding is used, i.e., using 4 bits for c of sort *Ctl*, and 2 bits for ip_i of sort *port*.

The description of the fabric behavior 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 for a specific custom variable order (user specified): The transition from state 5 to state 6 under the meta-symbols ~*s* and *h* of Figure 5 is encoded by the following set of highlighted paths in the MDG graph: (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 shown 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.

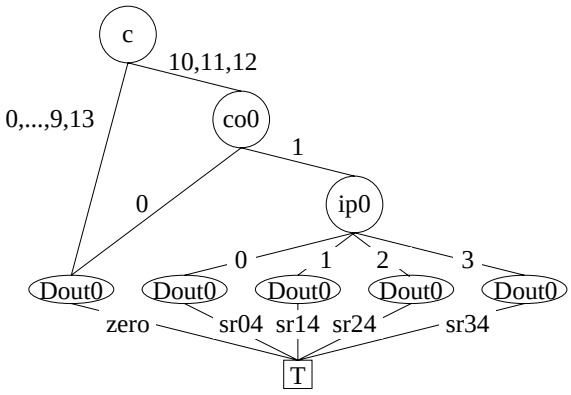


Figure 7: The MDG associated with $Dout_0$

6 Implementation of the Switch Fabric

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 it comes with a library of predefined basic components (logic gates, multiplexors, registers, bus drivers, ROMs, etc.), each of which is modeled as an ASM. Based on the gate-level description, we also produced an RTL implementation by abstracting the data Boolean signals Din_i/$Dout_i$ (i=0,1,2,3) as n-bit abstract words, and by describing the dataswitch using abstract multiplexors instead of logic gates. This led to a simpler representation of the dataswitch using a smaller number of more abstract components, making its switching behavior more natural.

6.1 Gate-Level Netlist

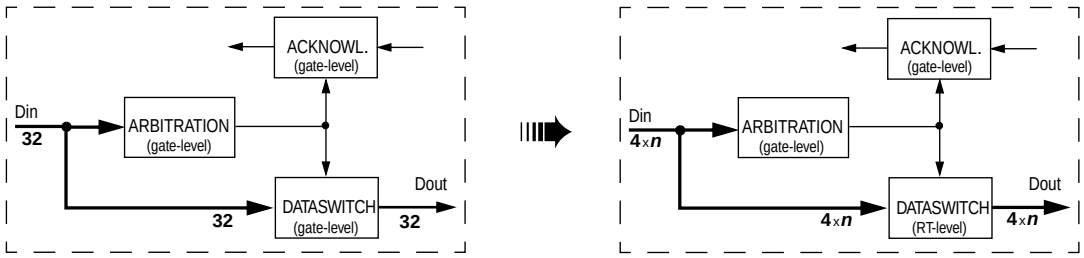
Many of the modules in the original Qudos description were large and logically unrelated, reflecting the mapping to a Xilinx gate array. Using a method similar to that used by Curzon in HOL [13], we organized the model in several levels of hierarchy, making use of modularity within MDG-HDL that is lacking 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. All elements of the Qudos library used in the original design have equivalent atomic components provided in the MDG software. Note that all data sorts in the MDG-HDL gate-level description are of the concrete Boolean sort *bool*.

6.2 RTL Model

The data inputs and outputs of the switch fabric are 4 bytes wide. In Qudos HDL there is no facility for describing such high-level words, and thus the data-in and data-out lines are modeled as 32 individual lines. In MDG-HDL, this could also be described 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., *wordn*, as in the behavioral specification (Section 5.2). Such high-level words are of arbitrary size, i.e., generic with respect to the word sizes.

An immediate consequence of modeling data as a compact word of abstract sort is that we can simplify the model of the dataswitch unit by using abstract data multiplexors instead of collections of logic gates.



Original Gate-level Model

Abstracted RTL Model

Figure 8: Model abstraction for the switch fabric

Figure 8 shows the abstraction of the switch fabric model. The arbitration block accepts *n*-bit data of sort *wordn*. Therefore, we must introduce a set of uninterpreted functions (cross-operators) that extract (decode) the fields *active*, *priority*, and *route* from the now abstract headers as illustrated in Figure 9. For example, the active bit is obtained using the function *act* of type [*wordn* $\rightarrow$ *bool*].

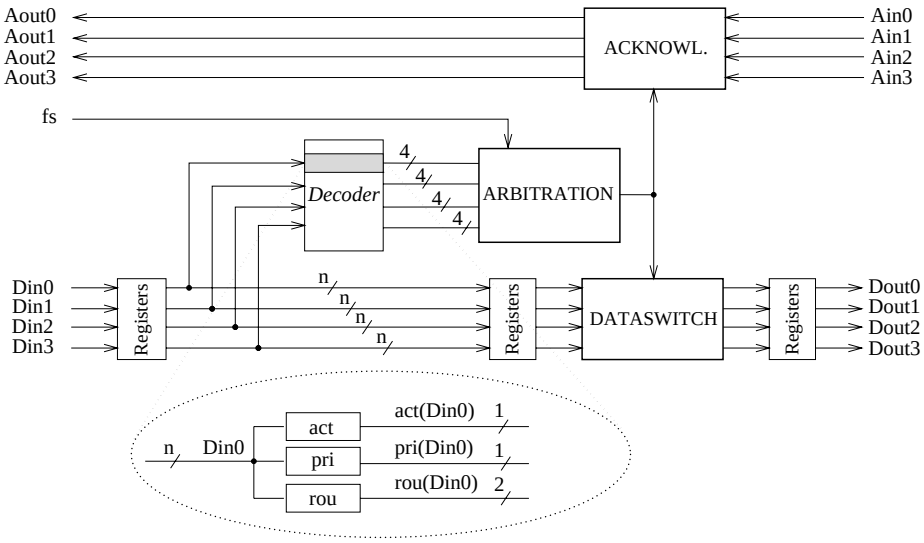
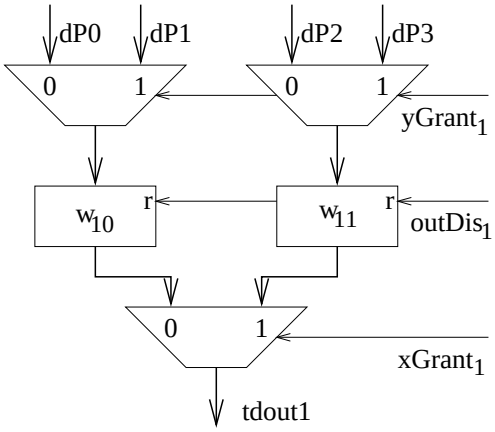


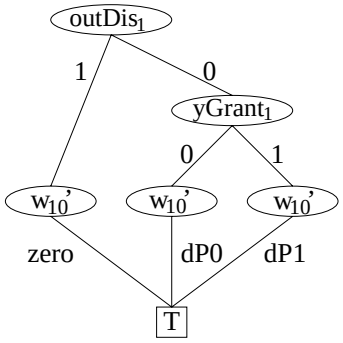
Figure 9: Extraction of control fields of the header

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

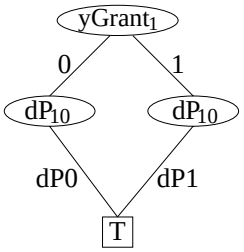
Given the RTL netlist, a single Abstract State Machine of the fabric is obtained by composing, for each primary output or register, the MDGs of the components in its cone of influence and abstracting the interconnection signals [27], like in the ROBDD case. We thus obtain a set of state-transition and output relations, one for each state variable / output that jointly define the state transition relation of the overall machine. 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*. The control signals $xGrant_i$, $yGrant_i$ and $outDis_i$ are displayed in Figure 3. Given the MDGs of a multiplexor and a register with synchronous reset as shown in Figure 10(c) and (d), the MDG of w_{10}' (next value of w_{10}) is obtained by relational product of the MDGs of the instances of the multiplexor on the input and the register, and existential abstraction of the (combinational) interconnection variable dP_{10} (the output of the left multiplexor in Figure 10(a)). The result of the composition is shown in Figure 10 (b). The MDGs associated with the other registers and the output signals are obtained in a similar way.



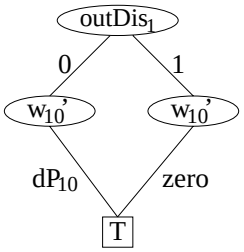
(a) RTL netlist



(b) MDG of signal w_{10}'



(c) MDG of a Multiplexor



(d) MDG of a Register

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

Table 1 summarizes the number of network components, the number of network signals and the number of state variables for each of the three modeling hierarchies, gate level, RTL and behavioral level. Signals and state variables are further itemized into concrete and abstract sorts.

Table 1: Modeling Statistics for the Three Description Levels

Level	Number of Components	Number of Signals		Number of State Variables	
		Abstract	Concrete	Abstract	Concrete
Gate Level	504	519		162	
		0	519	0	162
RT Level	298	307		50	
		36	271	20	30
Behavioral Level	NA	NA		25	
		-	-	16	9

7 Validation of the Specification by Property Checking

To verify the switch fabric, we wish to carry out a proof of correctness of the implementation model against the behavioral model (specification). Before doing so, however, we must ensure that the specification itself is correct with respect to the Fairisle network environment. Therefore, we first applied property checking to ascertain 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.

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 top-level timing specification in [13] and the other design documentation of the Fairisle switch [21]. 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 (i.e., next t_s + 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 (i.e., next t_s), 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 [25]. This is elaborated in the following sections.

7.1 Properties Specification

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*, 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.

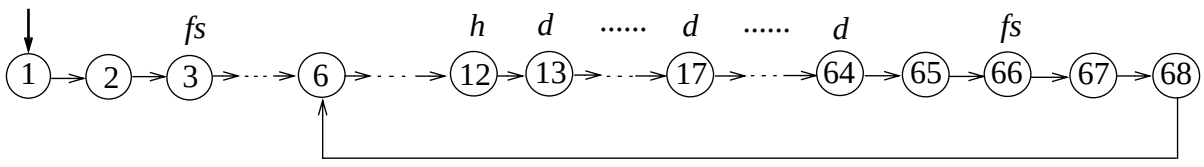


Figure 11: Environment state machine - frame generator

One cycle starting from state 6 and back corresponds to one frame. With this diagram, we can map the time points t_s, t_h and t_e (next t_s) to states. In this case, t_s corresponds to state 3 or 66; t_h corresponds to state 12; and t_e=t_s. Then, between states 17 and 68 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 of the frame generator rather than time points. It can be verified that the generator state machine is an instance of the general timing state machine (Figure 5) with cell length of C ≥ 53 and frame size of F ≥ 64.

The environment state machine of Figure 11 can be defined using a state variable *s* of concrete sort having the enumeration [1..68]. Accordingly, we now restate the previous properties as invariants using ITE (If-Then-Else) formulas of the Prolog-style MDG-HDL based on that machine.

- *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]) $\wedge$ *priority*[0..3] = [1,0,0,0] $\wedge$ *route*[0] = 0 **then** $Dout_0 = Din_0$' **else** don't care

- *Property 4*: **If** ($s \in$ [15,...,66]) $\wedge$ *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 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 of sort *wordn* disjoint from all other variables and it represents any (*don't care*) value. As the logic expression of the property represents a set of states, *x* is implicitly existentially quantified [10].

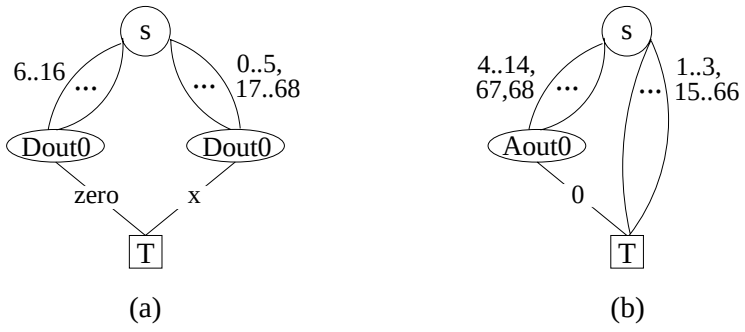


Figure 12: MDGs of *Property 1* and *Property* 2

7.2 Nondeterministic Frame Generator

The environment of the switch fabric periodically generates the frame start signal *fs* during one 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 is generated at the 8th 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). The period of the *fs* signal is thus at least 9 cycles for the cell transmission not to be aborted. The specification and the implementation of the fabric should operate under any frame size satisfying this constraint.

As an alternative to the previous method, we may carry out the property verification using a nondeterministic frame-pulse generator, as shown in Figure 13. The variable *y* in the figure is a free input that nondeterministically controls the choice of the frame size. The generator shown in Figure 11 is hence a specific instance of the nondeterministic generator for a typical frame size of 53 + 11 = 64 bytes. With this nondeterministic machine, t_s (t_e) corresponds to state 3 or 15 and t_h corresponds to state 12. The remaining bytes of the cell following the header are switched to the output port during the loop at state 14. It can be also verified that the generator state machine is an instance of the general timing state machine (Figure 5) with cell length of C ≥ 1 and frame size of F ≥ 11.

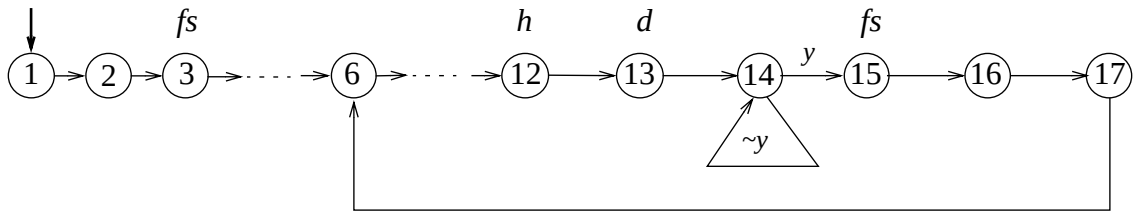


Figure 13: Nondeterministic frame generator model

The environment state machine of Figure 13 can be defined in a similar way to the previous case using a state variable *s* of concrete sort having the enumeration [1..17]. However, since we no longer have a one-to-one correspondence between the states of this machine and the data octets in the cell (they are all but one covered by state 14), we had to add a 5-state counter (idle + 4 counting states) *c* of enumeration [0..4] to indicate the required 4-cycle delay needed for stating the properties. The counter is started (i.e., *c* = 1) when the appropriate state of the generator model is reached. It is then re-initialized once the end of the frame (state 17) is reached. The properties specification now refers to the generator and counter states, otherwise their structure is the same as before.

- *Property 1*: **If** $(s \in [6,...,12]) \vee (c \in [1,...,4])$ **then** $Dout_0 = zero$ **else** don't care

- *Property 2*: **If** $(s \in [4,...,13,16,17]) \vee (c = 2)$ **then** $Aout_0 = 0$ **else** don't care

- *Property 3*: **If** $((s=14 \wedge c=4) \vee (s=15 \wedge c \in [3,4]) \vee (s=16 \wedge c \in [2,...,4]) \vee (s=17 \wedge c \in [1,...,4]))$ $\wedge$ *priority*[0..3] = [1,0,0,0] $\wedge$ *route*[0] = 0 **then** $Dout_0 = Din_0'$ **else** don't care

- *Property 4*: **If** $(s \in [14,15] \wedge c \in [1,2]) \wedge$ *priority*[0..3] = [1,0,0,0] $\wedge$ *route*[0] = 0 **then** $Aout_0 = Ain_0$ **else** don't care

7.3 Properties Verification

To verify these safety properties, we composed the fabric with the environment state machine as shown in Figure 14. As there is a 4-clock-cycle delay for the cells to reach the output ports, a delay circuit (four-stage shift register) is used to memorize the input values that are to be compared with the outputs. We thus can state the properties in terms of the equality between Din_0' and $Dout_0$ (e.g., *Property 3*). By composing these machines (the dashed frame in Figure 14) and the delay counter, we obtain the required platform for verification.

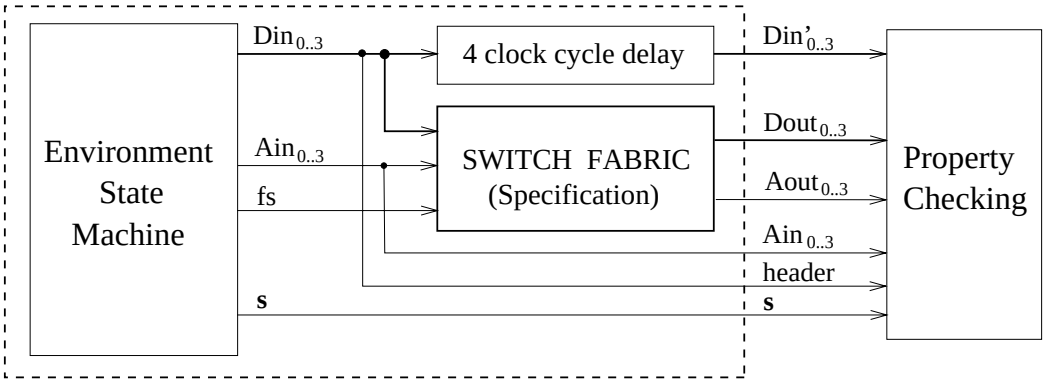


Figure 14: Composed state machine for property checking

By exploring the state space of the composed machine, we check in each reachable state if the outputs satisfy the logic expression of the property which should be true over all the reachable states. The experimental results from the verification of the properties (*Properties 1* to *4*) stated in Sections 7.1 and 7.2 are given in Table 2, including comparative results between the nondeterministic frame generator of Figure 13 and the more complex deterministic machine shown in Figure 11. The experiments were done on a SPARC station 20 with 128 MB of memory, and include the CPU time in seconds, the memory usage in megabytes and the number of MDG nodes generated. As expected, the execution times with the simple nondeterministic generator are up to two times shorter than those with the deterministic one. For instance, while the verification based on the deterministic machine needs 68 transitions, only 17 transitions are needed using the nondeterministic one.

Table 2: Property checking with deterministic and nondeterministic machines

Verification	CPU time (in sec)		Memory (in MB)		MDG Nodes generated	
	Determ.	Nondeter.	Determ.	Nondeter.	Determ.	Nondeter.
Property 1	486	367	40	33	90908	75117
Property 2	890	432	57	40	113666	80290
Property 3	344	208	37	31	83985	69833
Property 4	977	737	57	39	123423	84773

It is known that in some cases ([28] and [1]) the abstract reachability analysis may not terminate. This happens when either the set of states is infinite and cannot be represented finitely using the mechanisms currently available in the MDG tools (see [1] for a method that can alleviate this problem), or because the design is dependent on a particular interpretation of the function symbols

which are uninterpreted in the model (this is often resolved by providing a partial interpretation through rewrite rules as discussed earlier). In our case, the only uninterpreted function symbols are the cross-operators for extracting the various fields from the (abstract) cell headers. The implementation of the controller is completely independent of the interpretation of these extraction functions, and the abstract values carried by the cells are not modified by the switch. Consequently, the problem of nontermination does not occur in this case.

Basically, MDG-based reachability analysis terminates on a class of circuits whose control state machine has a cyclic behavior. The basic technique is the initial state generalization [10]. The ATM switch fabric example we considered here falls into this category. In Figure 5, when a header arrives at state 5, the state machine will eventually come back to this state (after a frame is transferred). As long as we generalize the data register values (from a constant to a variable) at state 5, the reachability analysis will terminate. In our experiment, we generalize the data registers at state 0. Given the fact that no data operations are performed during state 0 to state 5, this is the same as if the data registers were generalized at state 5. Therefore, nontermination is not a problem in this particular example.

8 Verification by Equivalence Checking

Our primary goal was to show that the original gate-level implementation of the switch fabric complies with the specification of the behavioral model. Since the implementation at the gate level is too big to be verified at once, we used the abstracted RTL model to close the semantic gap between the implementation and the behavioral model, and performed the verification hierarchically in two steps. (1) We verified that the RTL and the behavioral models exhibit the same behavior for arbitrary word width, and frame size and cell length. (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 using cross-operators. In the following sections, we will elaborate on each of these steps. The verification was achieved automatically in an acceptable amount of CPU time as will be shown in Section 8.3. 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. Behavioral Model Verification

To verify the RTL implementation against the behavioral specification, we made 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 was done for an arbitrary word width, and any frame size and cell length that respect the environment assumptions of the specification as expressed in Figure 5.

For the product of the machines from Sections 5.2 and 6.2, an MDG representing a set of total-states encodes a relation between 36 (16 + 20) abstract and 39 (9 + 30) concrete 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 15.

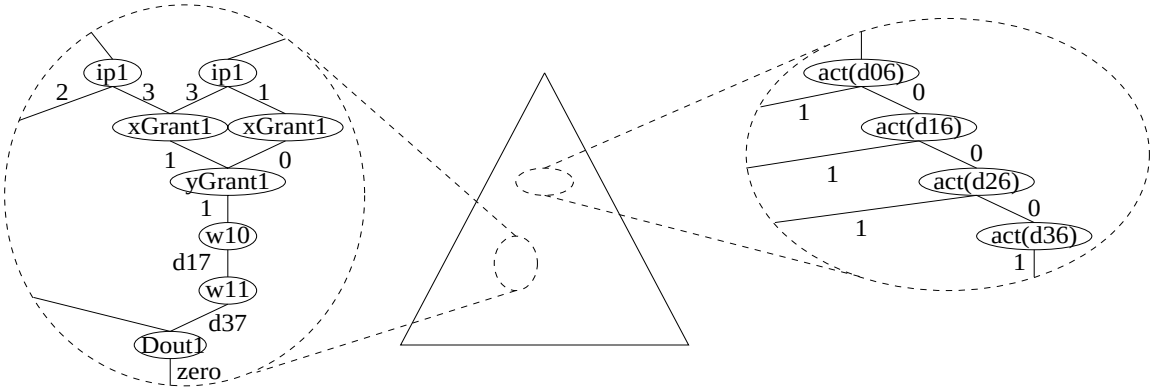


Figure 15: MDG encoding a set of reachable abstract states

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 of the ROBDD structure. 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 15, where the variables d_{ij} represent the values of Din_i at iteration j of the reachability analysis procedure—the j-th transition from the initial state). Decisions on the values of abstract data are represented by cross-terms which also contribute nodes in the MDGs (e.g., $act(d_{06})$ in Figure 15). Although cross-terms add complexity to the graph structure in general, the overhead is much smaller than the explosion caused by binary data encoding.

8.2 Gate-level vs. RTL Verification

The equivalence of the behaviors of the gate-level and the RTL models cannot be established for an arbitrary word size n since the gate-level description is not generic. As mentioned earlier, we must somehow "instantiate" the abstract data signals of the RTL model to 8 bits. This can be realized within the MDG environment using *uninterpreted functions* that 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

$$[(bool \times bool \times bool \times bool \times bool \times bool \times bool \times bool) \rightarrow wordn]$$

which concatenates any 8 Boolean signals to a single word and thus encodes each 8-bit bundle to a signal of sort *wordn*.

Using these functions, four symmetric configurations are possible for performing the equivalence verification. This is illustrated in Figure 16 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 shown in Figure 16) are not functional blocks that we add to the implementation description but they represent the

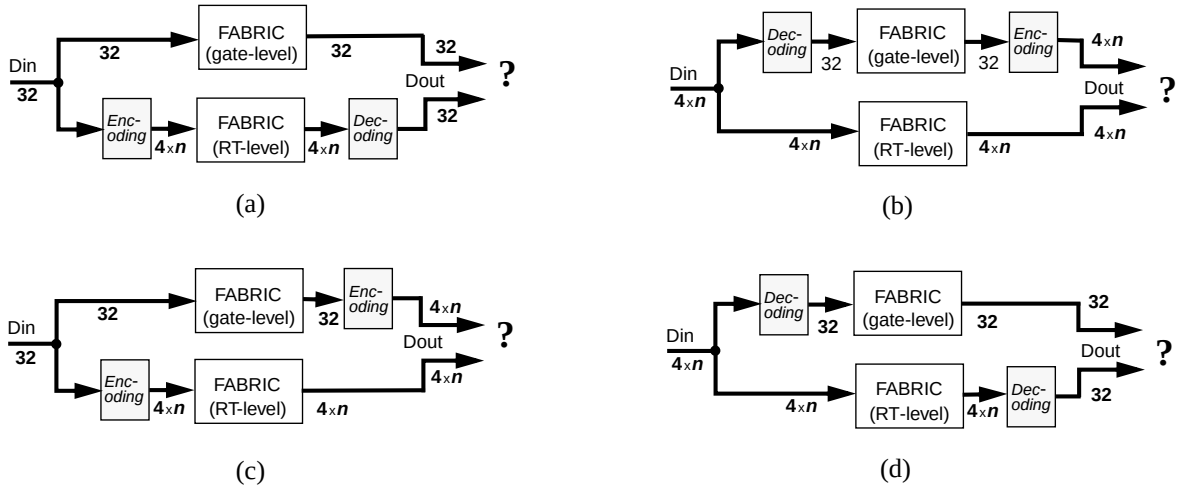


Figure 16: Different configurations for the RTL model verification

uninterpreted functions that we inserted into the logic formula of the equivalence property $D_{out\text{-}gate} = D_{out\text{-}RTL}$. Any one of these configurations could be used in the verification, however, case (d) in Figure 16 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 8-bit Boolean vector value we have to provide a correspondence to a specific constant of abstract sort, e.g., (0,0,0,0,0,0,0,0) ↔ *zero*, (0,0,0,0,0,0,1,0) ↔ *two*, where *zero* and *two* are generic constants of abstract sort *wordn*.

Since the data abstraction affects only the dataswitch block (Figure 8), the verification dealt mainly with the equivalence of the dataswitch blocks at the two levels. The verification run time for the (best) variant (d) in Figure 16 is given in Table 4.

8.3 Experimental Results

First, we report on the experimental results of comparing the behavioral model with the RTL model using MDGs. The experiments were done on a SPARC station 20 with 128 MB of memory. Table 3 shows the size of some typical MDGs generated during the reachability analysis and the CPU times for constructing them.

The MDG q_j encodes the frontier set of states at the end of iteration j, while v_j encodes the set of all reached states. q_0 is the MDG encoding the initial total states. The MDG q_7 encodes 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 behavioral and RTL models was detected.

Table 3: Size and generation times for some MDGs

MDG	Number of Nodes	CPU time (sec)
q_0	71	1
q_7	8321	80
v_7	8572	5
q_8	11665	150
v_8	20225	10
q_9	25985	300
v_9	46160	10
q_{10}	4995	1100
v_{10}	51106	60
q_{11}	1	60

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 RTL and gate-level models, respectively. 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. These rewriting rules must be declared by the user to partially interpret the function symbols for use by a rewriting algorithm included in the MDG package [29].

By combining the above two verification steps, we obtained a complete verification of the switch fabric from high-level behavior down to the gate-level implementation. The experimental results on a SPARC station 20 are recapitulated in Table 4, including the CPU time, the memory usage and the number of MDG nodes generated. It is apparent that in the RTL vs. behavioral verification a large set of states is encoded. Because of the abstract data representation and graph sharing in MDG, the encoding resulted in an acceptable number of nodes, however. In Section 8.4 we shall see that using a binary data representation and ROBDD encoding, the same verification becomes more costly and even impossible in the same amount of memory.

The verification of the RTL model against the gate level consumed less CPU time and memory, because it was reduced to comparing only the dataswitch block outputs of both machines (the same arbitration and acknowledgment units were used in both models). The gate-level dataswitch model contains 168 components including 64 (Boolean) state variables while the RTL model of the dataswitch block contains 20 components including 8 abstract state variables. Note that a similar verification of the dataswitch block through equivalence checking in VIS failed because of the state space explosion induced by the ROBDD encoding and the use of Boolean state variables at both levels [22].

Table 4: Experimental results of equivalence checking

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

Our verification confirms the results obtained by Curzon using HOL [13] where he indicated that no errors were discovered in the implementation. 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 (Figure 3). 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 error. Table 5 shows the results of these three experiments, including the CPU time for performing the reachability analysis and for generating counterexamples, the memory usage, and the number of MDG nodes generated.

Table 5: 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

As in Curzon's work [13] 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* [22] 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 one used in VIS. The model for VIS was written in Verilog, in contrast to our MDG-HDL, thus allowing the direct test of the specifications using commercial simulators. Unlike Verilog descriptions, Curzon's higher-order logic specifications as well as our MDG-HDL descriptions are not directly executable.

In Curzon's verification with HOL, much work was needed to prove a large number of lemmas and to set up the proof scripts interactively, e.g., the time spent on the verification of the dataswitch unit was about one week [13]. The related proof script was approximately 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 for the careful management of variable ordering (which so far has to be done manually). The VIS verification was also conducted automatically. In addition, VIS provides several options for dynamic variable ordering. Major effort was spent, however, in 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 data version of the fabric using equivalence checking (4 bits are needed to contain the 4 header bits—see Figure 2).

Curzon reported in [14] that using HOL the detection and correction of errors in one erroneous 4 by 4 switch fabric 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 that can be analyzed using commercial simulators. We are in the course of developing a VHDL to MDG-HDL translator which will make 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 and to provide the required time reference for safety-property checking. In addition, to cope with the state-space explosion of ROBDDs in VIS during CTL model checking, several compositional and property division techniques were adopted [22]. On the other hand, while VIS allows the model checking of both safety and liveness properties, only safety invariants have been verified in the MDG approach. Recently, however, an MDG model checking algorithm based on a restricted first-order linear temporal logic (*L-MDG*) [26] was developed and will be included to the MDG software package to allow the checking of liveness properties as well.

One potential difficulty with verification techniques based on abstract implicit reachability analysis as embodied in the MDG tools is the problem of nontermination. As in the case of safety property verification as discussed in Section 7.3, this was not a source of concern in the equivalence verification and for the same reasons.

More details on the comparison of HOL, MDG and VIS for hardware verification using the Fairisle switch fabric as a case study can be found in [24].

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. To gain confidence in the developed behavioral 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. We verified the equivalence of the RTL model and the behavioral model, and 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. 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 in the same amount of workstation memory. We have demonstrated that formal verification of a realistic (albeit still relatively small) piece of communication hardware can be accomplished efficiently 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, safety property 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 reduction technique (as confirmed in [22]). In general, if the control needs *n* bits, then it is impossible to reduce the data width to less than *n*. Thus, for datapaths containing mixed control/data information, ROBDD based data reduction techniques are not quite 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).

It is reported in [13] that the time spent on simulation would have been on 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.

Our experimental work used the 4x4 version of the Fairisle switch fabric as available from Cambridge. Redesigning the fabric for 8 or 16 connections was not practical for us, and, given that the MDG tools are a prototype, we do not think that it would be possible to carry out successfully the verification of the larger versions on the available workstations (SPARC 20 with 128MB of memory). However, work is under way to verify a 16x16 design consisting of 8 (4 + 4) 4x4 fabrics in a hierarchical way using a hybrid MDG-HOL approach: deploy MDG-based model checking to verify the 4x4 fabric and then use the results to prove in HOL that the 16x16 design is correct.

Acknowledgments

We are grateful to Paul Curzon at Middlesex University, UK, and Francisco Corella at Hewlett-Packard Company, USA, for encouragement and helpful comments. This work was partially funded by NSERC-Nortel Cooperative Research Grant CRD No. 191958 and NSERC research grant No. OGP0194302. The experiments were carried out on workstations on loan from the Canadian Microelectronics Corporation.

References

[1] O. Aït-Mohamed, X. Song, and E. Cerny, "On the Non-Termination of MDG-Based Abstract State Enumeration," Proc. IFIP Conference on Correct Hardware and Verification Methods (CHARME'98), Montreal, Canada, pp 218-235, October 1997.

[2] O. Ait Mohamed, E. Cerny, X. Song, 1998, "MDG-based Verification by Retiming and Combinational Transformations," Proc. IEEE 8th Great Lakes Symposium on VLSI (GLS-VLSI'98), Lafayette, Louisiana, USA, pp. 356-361, February 1998.

[3] A. Aziz et al., "HSIS: A BDD based Environment for Formal Verification," Proc. ACM/IEEE Design Automation Conference (DAC'94), New York, USA, pp. 454-459, June 1994.

[4] 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.

[5] R. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," IEEE Transactions on Computers, vol. C-35, pp. 677-691, August 1986.

[6] J. Burch, D. Dill, "Automatic Verification of Pipelined Microprocessor Control," In: D. Dill (editor), Computer Aided Verification, Lecture Notes in Computer Science 818, Springer Verlag, pp. 68-80, 1994.

[7] 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.

[8] 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, pp. 79-113, 1997.

[9] 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.

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

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

[12] O. Coudert, C. Berthet, J. Madre, "Verification of Synchronous Sequential Machines using Boolean Functional Vectors," In: L. Claesen (editor), Proc. IFIP International Workshop on Applied Formal Methods for Correct VLSI Design, Leuven, Belgium, pp. 111-128, November 1989.

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

[14] 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.

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

[16] E. Garcez, "The Verification of an ATM Switching Fabric using the HSIS Tool", Technical Report, WSI-95-13, Tübingen University, Germany, 1995.

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

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

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

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

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

[22] 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, pp. 368-373, February 1998.

[23] 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), pp. 89-104, August 1995.

[24] S. Tahar, P. Curzon, and J. Lu, "Three Approaches to Hardware Verification: HOL, MDG and VIS Compared," Proc. International Conference on Formal Methods in Computer-Aided Design (FMCAD'98), Palo Alto, California, USA, November 1998. Lecture Notes in Computer Science, Springer Verlag, 1998.

[25] G. Thuyau 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.

[26] Y. Xu, E. Cerny, X. Song, F. Corella, and O. Aït-Mohamed, "Model Checking for a First-Order Temporal Logic using Multiway Decision Graphs," In: A. Hu, and M. Vardi (editors), Computer Aided Verification, Lecture Notes in Computer Science 1427, Springer Verlag, pp. 219-231, 1998.

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

[28] Z. Zhou, X. Song, S. Tahar, E.Cerny, F. Corella and M. Langevin, "Formal Verification of the Island Tunnel Controller using Multiway Decision Graphs," In: M. Srivas, and A. Camilleri (editors.), Formal Methods in Computer-Aided Design, Lecture Notes in Computer Science 1166, Springer Verlag, pp. 233-246, 1996.

[29] Z. Zhou and N. Boulerice, "MDG Tools (V1.0) User's Manual," University of Montreal, Dept. D'IRO, 1996.