Middlesex University
School of Computing Science
Technical Report
ISSN 1462-0871

Comparing HOL, MDG and VIS: A Case Study on the Verification of an ATM Switch Fabric

Paul Curzon[‡], Sofiène Tahar[§] and Jianping Lu[§]
[‡] School of Computing Science, Middlesex University, UK
[§] Dept of Electrical and Computer Engineering, Concordia University,Canada.

July 1999

CS-99-05

Research Administrator
School of Computing Science
Middlesex University
Bounds Green Road
London N11 2NQ
U.K.
Tel +44 181 362 6336
Fax +44 181 362 6411
Email s.alhassan@mdx.ac.uk
http://www.cs.mdx.ac.uk/

Abstract

There exist a wide range of hardware verification tools, some based on interactive theorem proving and other more automated tools based on decision diagrams. In this paper, we compare three different verification systems covering the spectrum of today's verification technology. In particular, we consider HOL, MDG and VIS. HOL is an interactive theorem proving system based on higher-order logic. VIS is an automatic system based on ROBDDs and integrating verification with simulation and synthesis. The MDG system is an intermediate approach based on Multiway Decision Graphs providing automation while accommodating abstract data sorts, uninterpreted functions and rewriting. As the basis for our comparison we used all three systems to independently model and verify a fabricated ATM communications chip: the Fairisle 4×4 switch fabric.

An earlier version of this report appeared as: S. Tahar, P. Curzon, and J. Lu: Three Approaches to Hardware Verification: HOL, MDG and VIS Compared; In: Gopalakrishnan, G. and Windley, P. (Eds.), *Formal Methods in Computer-Aided Design*, Lecture Notes in Computer Science 1522, Springer Verlag, 1998, pp. 433-450.

A comparison of HOL and MDG which this work extends is to appear in the *Nordic Journal of Computing*.

Contents

List of Figures

List of Tables

Glossary

ASM	Abstract State Machine
ATM	Asynchronous Transfer Mode
BDD	Binary Decision Diagram
BLIF-MV	The Intermediate Language used by VIS
CTL	Computation Tree Logic
DAG	Directed Acyclic Graph
FSM	Finite State Machine
HDL	Hardware Description Language
HOL	Higher-Order Logic (an Interactive Proof System)
ITE	If-Then-Else Formulae
LCF	Logic for Computable Functions (an Interactive Proof System)
MDG	Multiway Decision Graph
PVS	Prototype Verification System (an Interactive Proof System)
ROBDD	Reduced Order Binary Decision Diagram
RTL	Register Transfer Level
SMV	Symbolic Model Verifier (an Automated Verification System)
VIS	Verification Interacting with Synthesis (an Automated Verification System)

1 Introduction

Formal hardware verification techniques have established themselves as a complementary means to simulation for the validation of digital systems due to their potential to give very strong results about the correctness of designs. Many academic and commercial verification tools have emerged in recent years, which can be broadly classified into two contrasting formal verification techniques: interactive formal proof and automated decision graph based verification. This paper compares and contrasts such tools using an Asynchronous Transfer Mode (ATM) switch fabric as a case study.

In the interactive proof approach, the circuit and its behavioural specification are represented in the logic of a general purpose theorem prover. The user interactively constructs a formal proof to prove a theorem stating the correctness of the circuit. Many different proof systems with a variety of interaction approaches have been used. In this paper we consider one such system: HOL [13], an LCF style proof system based on higher-order logic.

In the automated decision diagram approach the circuit is represented as a state machine. Techniques such as reachability analysis are used to automatically verify given properties of the circuit or verify machine equivalence. We consider the MDG [5] and VIS [3] tools. The VIS tool is based on a multi-valued extension of pure ROBDDs (Reduced Ordered Binary Decision Diagrams [2]). The MDG system uses Multiway Decision Graphs [5] which subsume ROBDDs while accommodating abstract sorts and uninterpreted function symbols.

As the basis of our comparison, we used HOL, MDG and VIS to independently verify the Fairisle 4×4 switch fabric [18][1]. This is a fabricated chip which forms the heart of an ATM communication switch. The device, designed at the University of Cambridge, is used for real applications in the Cambridge Fairisle network. It switches data cells from input ports to output ports within the ATM switch, arbitrating clashes and sending acknowledgments. It was not designed for the verification case study. Indeed, it was already fabricated and in use, carrying real user data, prior to any formal verification attempt.

We thus based the comparison study on a significant, real hardware design. A similar approach was also taken by Angelo *et al.* [1] when comparing HOL and the Boyer-Moore theorem prover. An alternative approach is to look at a wide range of small examples [17]. The latter helps ensure that conclusions apply to more than just a single example. However, a danger is that issues relevant to real designs are not raised. For example, a major concern for verification technologies is whether they scale to large designs. This is clearly of great interest to industry. If only small examples are considered, the problem does not arise. The two approaches are clearly complementary, and are both of importance.

The outline of the paper is as follows. In the next section we overview related work. In Section 3 we give an overview of the hardware considered: the Fairisle 4×4 switch fabric. We describe its verification using HOL, MDG and VIS in Sections 4,

[1]See URL http://www.cl.cam.ac.uk/Research/HVG/atmproof/ for more details of Fairisle, the 4×4 fabric design and the separate verification projects.

5 and 6, respectively. For each, we overview the verification approach and the structural and behavioural specification methods. We also discuss for each the time taken, the error detection capabilities, and scalability issues. Finally, in Section 7 we draw conclusions comparing and contrasting different aspects of HOL, MDG and VIS.

2 Related Work

There has been a vast amount of work on formal hardware verification. We summarise here the work that is directly related to our study on verifying network hardware components.

Herbert [15] used HOL to formally verify the ECL chip: a local area network interface which formed part of the Cambridge Fast Ring. This is of roughly similar complexity to the circuit we considered, though our HOL proof took less time, demonstrating the increased maturity of the system.

Chen *et al.* at Fujitsu Digital Technology Ltd. [4] verified an ATM circuit that makes high-speed switching operations at 156 MHz and consists of about 111K gates. When the circuit was manufactured it showed an abnormal behaviour under certain circumstances. Using the SMV tool [22], the authors identified the design error by checking some properties expressed in Computational Tree Logic [22]. Due to the restriction of the Boolean computation used by SMV and in order to avoid a state space explosion, they had to abstract the data width of addresses from 8 bits to 1 bit, and the number of addresses in the Write Address FIFO from 168 to 5. Although the design error was diagnosed, there is no proof showing that the abstracted circuit was itself correct. Later, Rajan *et al.* [25] used a combination of simulation, theorem proving and model checking based on PVS [23] to validate a high-level ATM switch model from Fujitsu Ltd. The authors used model checking to verify some control components in the ATM model, and applied exhaustive simulation to verify some operational components. Theorem proving was then applied to verify the whole ATM switch model. They discovered bugs in the high-level ATM model which had been presumed correct by simulation. More recently, Planisamy and Tahar [24] verified using VIS model checking the Egress routing logic of an ATM RCMP (Routing, Cell Counting, Monitoring and Policing) design from PMC Sierra, Inc.

Schneider *et al.* [26] formally verified the Fairisle 4×4 switch fabric using a verification system based on the HOL theorem prover, MEPHISTO. They described the structure of each of the modules used in the hardware design hierarchically down to the gate level and provided their behavioural specifications using hardware formulas. Although they automated the verification of lower-level hardware modules which implement the top-level block units, they did not accomplish the complete verification of the intended overall behaviour of the switch fabric against its implementation.

Other groups have also used the 4×4 fabric as a case study. Coupet-Grimal and Jakubiec have used it in their work using the Coq proof system for hardware verification [16]. Garcez has also verified some properties of the 4×4 fabric using the HSIS model checking tool [10].

A variation of the 4×4 fabric can be used in the design of a larger 16×16 fabric, by connecting 8 of the smaller designs in a delta network. Curzon has verified such a 16×16 fabric using HOL [8]. Voicu *et al.* [28] verified using VIS model checking an abstract version of the port controllers of the Fairisle ATM switch. This latter verification has been combined with that of the switch fabric to obtain the verification of the entire Fairisle ATM switch using VIS [21].

3 The Fairisle 4×4 Switch Fabric

We used the Fairisle switch fabric as the basis of our study. It is a good choice for a comparison study such as this for several reasons. It is a real, fabricated design which was not designed as a verification case study. It therefore gives a real test of the verification systems. Hardware designed to be a verification case study is likely to be over simplified and thus problems that would arise for a real design may be missed. While not being a trivial design, it is simple enough for a verification to be performed in a reasonable amount of time. A significant feature is that it combines control hardware with a datapath. This combination causes problems for traditional BDD based verification systems. Furthermore, control information occurs actually within the data. This prevents verification of a version with a 1-bit reduced width from being possible – a standard technique in decision diagram based verification.

The Fairisle switch forms the heart of the Fairisle network. It consists of a series of port controllers connected to a central switch fabric. In this paper, we are concerned with the verification of the switch fabric which is the core of the Fairisle ATM switch. The port controllers provide the interface between the transmission lines and the switch fabric, and synchronize incoming and outgoing data cells, appending control information to the front of the cells in a routing byte.

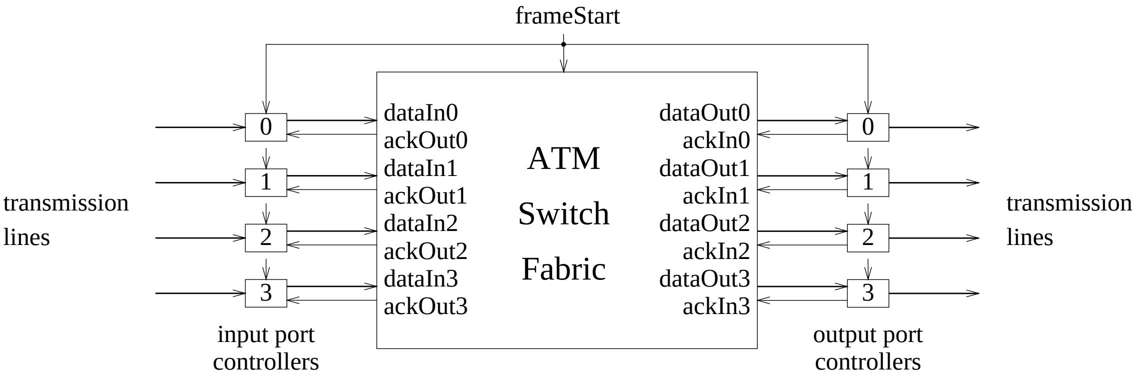

Figure 1: The Fairisle ATM Switch

A cell consists of a fixed number of data bytes which arrive one at a time. The fabric switches cells from the input ports to the output ports according to the routing byte (header) which is stripped off before the cell reaches the output stage of the fabric. If different port controllers inject cells destined for the same output port controller (indicated by *route* bits in the routing byte) into the fabric at the same time, only one will succeed—the others must retry later. The routing byte also includes a priority bit (*priority*). It is used by the fabric during round-robin arbitration giving preference to cells with the priority bit set. The fabric sends a negative acknowledgment to the unsuccessful input ports, and passes the acknowledgment from the requested output port to the successful one.

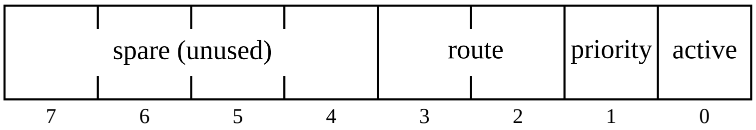

Figure 2: The Routing Byte of a Fairisle ATM Cell

The port controllers and 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* signal, which ensures the port controllers inject data cells into the fabric so routing bytes arrive together. The fabric does not know when this will happen. Instead, it monitors the *active* bit of the routing bytes: when any goes high the cells have arrived. If no input port raises the active bit throughout the frame then the frame is inactive—no cells are processed; otherwise it is active.

The behaviour of the switch fabric is cyclic. In each cycle or frame, it waits for cells to arrive, reads them in, processes them, sends successful ones to the appropriate output ports, and sends acknowledgments. It then waits for the arrival of the next round of cells.

Figure 3 shows a block diagram of the 4×4 switch fabric. It is composed of an arbitration unit (timing, decode, priority filter and arbiters), an acknowledgment unit and a dataswitch unit. The timing block controls the timing of the arbitration decision based on the frame start signal and the time the routing bytes arrive. The decoder reads the routing bytes of the cells and decodes the port requests and priorities. The priority filter discards requests with low priority which are competing with high priority requests. It then passes the resulting request situation for each output port to the arbiters. The arbiters (in total four—one for each port) make arbitration decisions for each output port and pass the result to the other units with the grant signal. The arbiters indicate to the other units when a new arbitration decision has been made using the output disable signals. The dataswitch unit performs the switching of data from input port to output port according to the latest arbitration decision. The acknowledgment unit passes acknowledgment signals to the input ports. Negative acknowledgments are sent until a decision is made.

Each unit is repeatedly subdivided down to the logic gate level, providing a hierarchy of modules. The design has a total of 441 basic components including 162 1-bit flip flops. It is built on a 4200 gate equivalent Xilinx programmable gate array. The switching element can be clocked at 20 MHz and frame start pulses occur every 64 clock cycles.

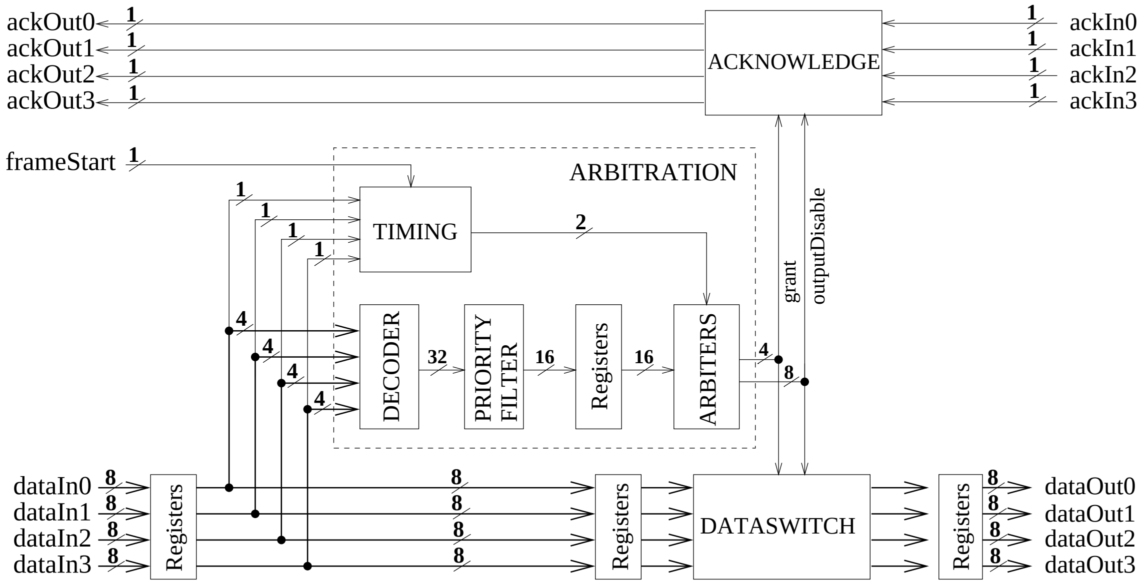

Figure 3: The Fairisle ATM Switch Fabric

4 The HOL Verification of the Fabric

In the first study, the fabric was verified using the HOL Theorem-Proving System. Formal structural and behavioural specifications were written for each module in the design, and a correctness theorem for each proved. These theorems were then used to construct a single correctness theorem for the whole fabric design relating its gate-level structural specification to its high level behavioural specification.

4.1 The HOL Theorem-Proving System

The HOL theorem proving system is an LCF-style [11] interactive theorem prover for higher-order logic [13]. In our work the HOL90 implementation of HOL was used. The original HOL system was intended as a tool for hardware verification. However, it is actually a general-purpose proof system that has subsequently been used in a wide variety of application areas. It provides a range of proof commands of varying sophistication, including rewriting tools and decision procedures. It is also fully user-programmable, allowing user-defined, application-specific proof tools to be developed. The basic interface to the system is a Standard ML interpreter. Standard ML is both the implementation language of the system and the meta-language in which proofs are written. Proofs are input to the system as calls to Standard ML functions.

The system is very flexible and a variety of different proof styles are supported. The main styles, however, are forwards and backwards proof. In the former style, to create new theorems, the user calls functions corresponding to axioms or inference rules. The latter are applied to previously proved theorems. Further theorems are created by applying other inference rules to the newly created theorems. Eventually, in this way, the desired theorem is proved. In backwards proof, the user sets the desired theorem as a goal. Tactics are then applied which break the goal into simpler subgoals in such a way that if a corresponding inference rule was applied to the subgoals, the theorem of the goal would be obtained. Tactics are repeatedly applied to the subgoals until they can be trivially proved, at which point the original goal can be made into a theorem. This is actually done by applying the inference rules which correspond to the applied tactics in a forwards manner, automatically. In practice, a mixture of these two styles is used, with forwards proof interspersed within backwards proofs.

The system represents theorems by a Standard ML abstract type. The only way a theorem can be created is by applying a small set of primitive inference rules that correspond to the primitive rules of higher-order logic. More complex inference rules and tactics must ultimately call a series of primitive rules to do the work. This means that the user can have a great deal of confidence in the results of the system. User programming errors cannot cause a non-theorem to be erroneously proved. That could only occur if there were errors in the few, relatively simple functions corresponding to the primitive inference rules of the system.

4.2 The Structural Specifications

The structural specification of a design describes its implementation: the components it consists of and how they are wired together. The original designers of the fabric used a relatively simple HDL, (Qudos HDL [9]), to give structural descriptions of the hardware. This description was used to simulate the design prior to fabrication. The Xilinx netlist was also generated from this description. The descriptions used in the verification were hand-derived from the Qudos descriptions. An example of a Qudos HDL specification is given below:

```
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;
```

This is the description of a multiplexer circuit (see Figure 4). It takes a 4 bit input `d` and a 1 bit input `x`, producing a 2-bit output `dOut`. `xBar` is an internal signal. The `Clb` statement is a dummy declaration providing information about the way the component design should be mapped into a Xilinx gate array. The multiplexer implementation consists of three components. `XiINV` is an inverter and the `AO` components are AND-OR logic gates. Wiring between modules is indicated by the use of common variable names. For example, `xBar` is an output of the inverter and an input to the AO gates.

The descriptions needed to perform a verification are similar to those used for simulation. However, for verification they must be written in a language with a formally defined semantics, which can be reasoned about easily. In the HOL verification, higher-order logic itself was used for this purpose. The formal specifications

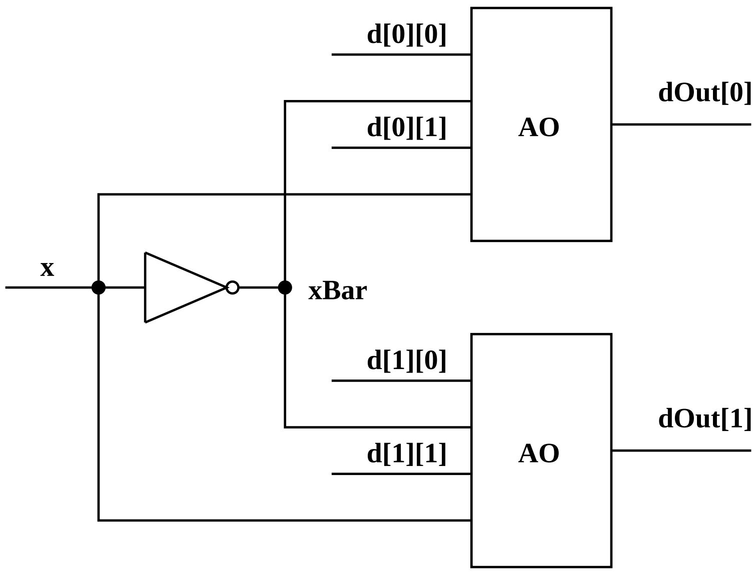

Figure 4: The DMUX4T2 multiplexer circuit

were developed by manually translating the original description into higher-order logic.

Hardware components are modeled in HOL using relations on the inputs and outputs [14]. For example, `XiINV(in,out)` is used to represent an inverter with a single input wire, `in`, and a single output wire, `out`. An input and output wire are in the relation `XiINV` if at all times the output is a negated version of the input. The wires themselves are represented by functions from time to the value on the wire at that time. For an inverter, the values are represented by booleans. However, wires can also hold more complex values such as words (e.g., a byte). The basic building blocks used in the HOL specifications were the basic units of the simulator used by the designers: logic gates (such as `XiINV`) and single bit registers.

Conjunction ($\wedge$) is used to join multiple components. As in Qudos HDL, the wiring is indicated by the use of the same variable as arguments to different modules. Individual bits of words are referenced using the `SBIT` operator. Thus, the following represents a circuit consisting of an inverter and a single AO unit, with the output of the former (`xBar`) being one of the inputs of the latter.

```
XiINV(x,xBar) ∧
AO((y1,xBar, y2, x), dOut)
```

Internal wires can be hidden using the `LOCAL` quantifier. This is actually just an alternative name for the existential quantifier. Thus, to internalize $xBar$ in the above we could write:

```
LOCAL xBar.
  XiINV(x,xBar) ∧
  AO((y1,xBar, y2, x), dOut)
```

The following is a HOL version of the module definition, that corresponds directly to the full Qudos definition of the above multiplexer example:

```
DMUX4T2((d,x),dOut) =
  LOCAL xBar.
   XiINV(x,xBar) ∧
   AO((SBIT 0 d,xBar,SBIT 1 d, x), SBIT 0 dOut) ∧
   AO((SBIT 2 d,xBar, SBIT 3 d, x), SBIT 1 dOut)
```

As can be seen from this example, Qudos structural descriptions can be mimicked very closely in HOL up to surface syntax. However, the extra expressibility of HOL was used to simplify and generalize the description. For example, in HOL words of words are supported. Therefore, a signal carrying 4 bytes can be represented as a word of 4 8-bit words, rather than as 4 separate signals or as one 32-bit signal. Similarly, we can model the input, `d`, of the multiplexer as 2 words of 2 bits (its natural structure). Its structural description then becomes:

```
DMUX4T2((d,x),dOut) =
  LOCAL xBar.
   XiINV(x,xBar) ∧
   AO((SBIT 0 (SBIT 0 d),xBar,SBIT 1 (SBIT 0 d), x), SBIT 0 dOut) ∧
   AO((SBIT 0 (SBIT 1 d),xBar,SBIT 1 (SBIT 1 d), x), SBIT 1 dOut)
```

With the structured version of `d`, the two occurrences of `AO` become the same up to the inner indices. We can therefore improve on the above by using a single occurrence of `AO` and the module duplication operator, `FOR`. It is just a bounded universal quantifier.

```
DMUX4T2((d,x),dOut) =
  LOCAL xBar.
   XiINV(x,xBar) ∧
   FOR i ::  0 TO 1 .
    AO((SBIT 0 (SBIT i d),xBar,SBIT 1 (SBIT i d), x), SBIT i dOut)
```

In HOL, arithmetic can also be used to specify which bit of a word is connected to an input or output of a component. For example, we can specify that for all i, the $2i$-th bit of an output is connected to the i-th bit of a subcomponent. This, again, meant that for the fabric we could avoid writing essentially identical pieces of code several times, as was necessary in the Qudos specifications. When an additional module, used in several places, is introduced, the verification task is reduced. This is because that module needs only be verified once, rather than for every instance.

It should be stressed that while the descriptions of the implementation were modified in the ways outlined above, no simplification was made to the implementation itself to facilitate the verification. The simplifications that were made were to the surface description (such as grouping components into extra modules). The netlists of the structural specifications used were intended to correspond to that actually implemented. This was not checked. One way this could be done is to compare the netlist descriptions derived from the two structural descriptions. However, we did not have a tool to derive the netlist from a HOL description.

4.3 The Behavioural Specifications

The behavioural specification against which the structural specification was verified describes the actual, unsimplified behaviour of the switch fabric. It is presented at a similar level of abstraction to that used informally by the designers. It describes the behaviour over a frame in terms of timing diagrams represented as interval operators. Within the interval, the values output are functions of the values input and state at earlier times.

A frame is specified to be an interval of time in which:

- at the start of the interval, the frame start signal is high;
- at the end of the interval, the frame start signal is high;
- the frame start signal is low at all other times in the interval; and
- the end time (t_e) is later than the start time (t_s).

A frame can then be either active or inactive. This is determined by an additional signal `active` derived from information in the cell headers. It indicates the arrival of a cell. An inactive frame is one in which this signal remains low throughout the frame. In an active frame, it remains low until some specified active time (t_h), at which point it goes high. Its value for the remainder of the frame is then unspecified. This is shown in Figure 5. Other restrictions are placed on the precise time within the frame when the active signal can occur. If it arises too close to the ends of the frame, then the fabric does not function correctly. The environment of the fabric must ensure that this does not occur. The precise behaviour in such situations can be omitted from the specifications since it is erroneous. This is a difference between the HOL and the MDG specifications. In the latter the behaviour in all circumstances must always be specified. This means the amount of specification work is less in this respect in HOL.

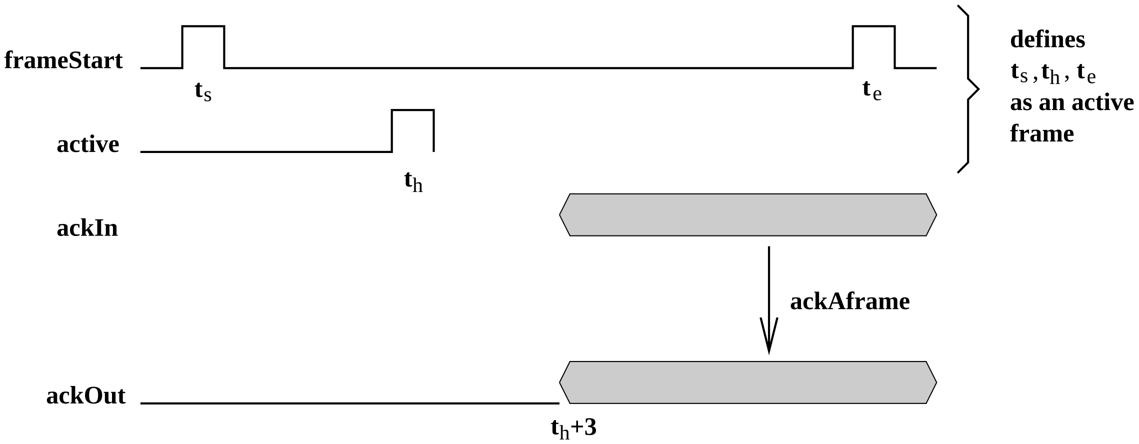

Figure 5: The Timing Diagram for the Acknowledgment Output

As an example of a behavioural specification, consider the specification for the acknowledgment signal on a frame where cell headers arrive at time t_h. The predicate

AFRAME specifies that we are dealing with intervals corresponding to such active frames. The *ackOut* signal must be zeroed (specified by ZEROW) until time $t_h + 3$. Thereafter, its value at a given time is specified by the function AckAframe. It depends on the arbitration decision made. This in turn depends on the value of the data injected into the fabric at time t_h (the header), the value of the last arbitration decision, and the value of the acknowledgments coming in from the output ports at the time in question. This functional behaviour is specified by a function argument to the interval operator, DURING. This specification is illustrated diagrammatically in Figure 5.

$$
\begin{array}{l}
\texttt{(AFRAME } t_s\ t_h\ t_e\ frameStart\ active\texttt{)} \supset \\
\quad \texttt{STABLE } (t_s+1)\ (t_h+3)\ ackOut\ \texttt{(ZEROW ...)} \quad \wedge \\
\quad \texttt{DURING } (t_h+3)\ (t_e+1)\ ackOut \\
\qquad (\lambda t.\ \ \texttt{AckAframe}\ (d\ t_h)\ (last\ (t_h+2))\ (ackIn\ t))
\end{array}
$$

In the above, *last* represents the state of the most recent arbitration decisions. STABLE is an interval operator similar to DURING. It specifies that the given signal has some constant value over an interval.

Other clauses in the specification describe the behaviour over an inactive frame in which no cells arrive. A similar set of clauses are given for the behaviour of the data output lines and the internal state, *last*. A feature of this style of specification is that the cell frame is very explicit in the description. This is a difference to the state machine based specification style used in MDG (described later).

In the HOL verification, it is not sufficient to simply provide a behavioural specification for the whole design. Each module is verified independently, as described in the next section. This means we must provide behavioural specifications for each of the 43 distinct modules in the design. However, once done for a particular module, this work does not need to be repeated if the module is reused.

The specifications of the more complex modules at the top of the design hierarchy were similar to that given above. The simpler ones at the bottom of the hierarchy, for which the frame structure was not applicable, were given point-time specifications rather than interval ones. For example, the specification of DMUX4T2 whose structural specification was given earlier is:

$$
\begin{array}{l}
\texttt{DMUX4T2\_SPEC ((}d\texttt{, } x\texttt{), } dOut\texttt{) =} \\
\quad \forall t.\ \ dOut\ t\ \texttt{= Mux}\ (x\ t)\ (d\ t)
\end{array}
$$

This states that at any time, t, the output, *dOut*, is a function of the inputs, x and d, at that point in time. That function is specified by Mux. It is defined in terms of general operators on the basic datatypes: BV which turns a boolean into a natural number and BITS which selects the indicated bit from each word within a word of words.

$$\texttt{Mux}\ x\ d\ \texttt{= BITS (BV}\ x\texttt{)}\ d$$

4.4 The Verification Process

The verification of the 4×4 switch fabric used standard techniques for hardware verification using higher-order logic [12, 14]. It was structured hierarchically following the module structure of the implementation. This hierarchical, modular nature of the proof facilitated the management of the complexity of the proof. Both the structural and behavioural specifications of each module were given as relations in higher-order logic. This meant that a correctness statement could be stated using logical implication for "implements". In general, the correctness statement thus had the form:

$$\begin{array}{l}\vdash \ \textit{assumptions on environment} \supset \\ \qquad (\textit{structure} \supset \textit{behaviour})\end{array}$$

i.e., under certain assumptions on the environment, the structural specification implements the behavioural specification.

The internal state, which is an explicit argument to the behavioural specification but implicit in the structural description (within the registers), is represented by an existentially quantified variable. Inputs and outputs are represented by universally quantified variables. Thus, the overall correctness statement (with details of word sizes omitted for the sake of exposition) has the form:

$$\begin{array}{l}\forall \mathit{ackIn}\ \mathit{ackOut}\ \mathit{dOut}\ d\ \mathit{frameStart}.\\ \ \ \texttt{ENVIRONMENT}\ \mathit{frameStart}\ d \supset\\ \ \ \ \texttt{FABRIC4B4}\ ((d,\mathit{frameStart},\mathit{ackIn}),\ (\mathit{dOut},\ \mathit{ackOut})) \supset\\ \ \ \ \ \exists \mathit{last}.\\ \ \ \ \ \ \texttt{FABRIC4B4\_SPEC}\ \mathit{last}\ ((d,\mathit{frameStart},\mathit{ackIn}),\ (\mathit{dOut},\ \mathit{ackOut}))\end{array}$$

The correct operation of the fabric relies on an assumption about the environment. In particular, cells must not arrive at certain times within two clock cycles of a frame start. The relation `ENVIRONMENT`, above, specifies this condition in a general way. This differs from the MDG verification where a very specific condition is given which corresponds to one particular way of satisfying the general condition.

A correctness theorem of the above form was proved for each module stating that its implementation down to the logic gate level satisfied the specification. This correctness theorem was proved by appealing to a correctness lemma about the module itself and to the main correctness theorems for its sub-modules. The lemma, in essence, asserts that the module is correct, assuming its sub-modules satisfy their specifications. It can be proved independently of the other modules. It is identical to the full correctness theorem except in one respect. An alternative structural specification for the module is used. It is defined in terms of the *specifications* of the sub-modules rather than their implementations. The sub-modules are thus treated as black boxes. Verifying the full design involves doing this for the top level module. The bottom level of the hierarchy consists of logic gates and single-bit registers. These are only specified behaviourally: they are left as black boxes in the correctness theorem.

The proof of the correctness lemma for each module was split into several parts. These parts corresponded to the separate intervals for each output signal given in the behavioural specification of the module. The proof for each interval was essentially inductive. A lemma was proved that the implementation satisfied the behaviour at the start of the interval. It was also proved that, within the interval, if the behaviour was satisfied at one time point, then it was also satisfied at the subsequent time point. From this it could be deduced that the implementation was correct over the whole interval.

In conducting the overall proof, the verifier needs a very clear understanding of why the design is correct, since a proof is essentially a statement of this. Thus performing a formal proof involves a deep investigation of the design. It also provides a means to help achieve that understanding. Having to write formal specifications for each module helps in this way. Having to formulate the reasons why the implementation has that behaviour gives much greater insight. In addition to uncovering errors, this can serve to highlight anomalies in the design and suggest improvements, simplifications or alternatives [7].

4.5 Time Taken

The module specifications (both behavioural and structural) were written prior to any proof. This took between one and two person-months. No breakdown of this time has been kept. Much of the time was spent in understanding the design. The structural specifications were adapted directly from the Qudos HDL. The behavioural specifications were more difficult. The specifier had no previous knowledge of the design. There was a good English overview of the intended function of the switch fabric. This also outlined the function of the major components. While it gave a good introduction, it was not sufficient to construct an unambiguous behavioural specification of all the modules. The behavioural specifications were instead constructed by analyzing the HDL. This was very time-consuming.

Approximately two person-months were spent performing the verification. Of this, one week was spent proving theorems of general use. Approximately 3 weeks were spent verifying the upper modules of the arbitration unit, and a further week was spent on the top two modules of the switch. 3–4 days were spent combining the correctness theorems of the 43 modules to give a single correctness theorem for the whole circuit. The remaining time of just over two weeks was spent proving the correctness theorems for the 36 lower level units. This can be seen in Figure 6 which shows the cumulative time in person-days (assuming an 8-hour day) taken to verify the separate modules' lemmas. The proofs of the upper-level modules were generally more time-consuming for several reasons: there were more intervals to consider; they gave the behaviour of several outputs; and those behaviours were defined in terms of more complex notions. They also contained more errors which severely hampered progress. The verifier had not previously performed a hardware verification, though was a competent HOL user. Apart from standard libraries, the work did not build directly on previous theories.

It takes several hours of machine time on a Sparc 10 to completely rebuild the

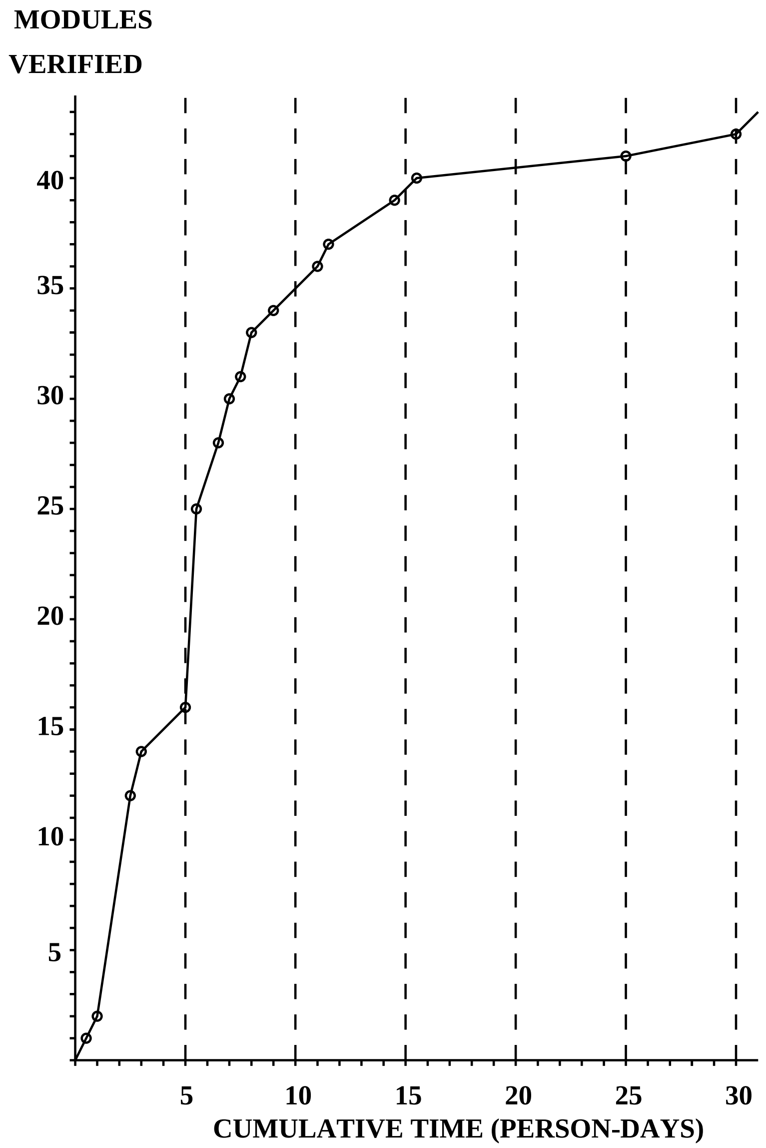

Figure 6: Time Taken to Verify the Fabric Modules using HOL

proofs from scratch by re-running the scripts in batch mode. Single theories representing individual modules generally take minutes to rebuild. A large proportion of the time is actually spent restarting HOL and loading in appropriate parent theories and libraries for each theory. In the initial development of the proof the machine time is generally not critical, as the human time is so much greater. However, since the proof process consists of a certain amount of replay of old proofs, a speed-up would be desirable, for example, when mistakes are made in a proof.

If changes are made to the design, it is important that the new verification can be done quickly. Since proof is very time consuming this is especially important. This problem is attacked in several ways in the HOL approach: the proofs can be made generic; their modular nature means that only affected modules need to be reverified; and proofs of modules which have changed can often be replayed with only minor changes. After the original verification had been completed, several variations on the design were also verified. These included real, fabricated variations that formed part of a 16×16 fabric. Although the 4×4 switch fabric took several months to specify and verify, the modified versions took only a matter of hours or days as can be seen from Figure 7 [6]. Generic proofs were not used to as great an extent as was possible in this study. This was because it was generally found simpler to reason about specific values than general ones.

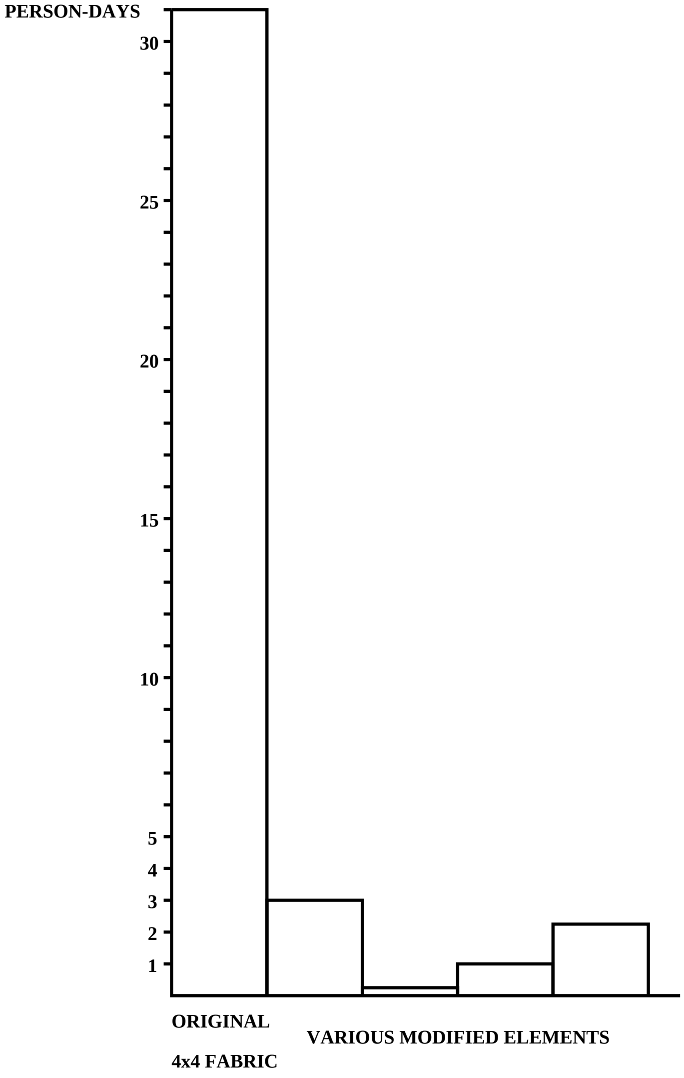

Figure 7: Time Taken to Verify Variations on the Fabric Design using HOL

One of the biggest disadvantages of the HOL system is that its learning curve is very steep. Furthermore, interactive proof is generally a time-consuming activity even for an expert. Much time is spent dealing with trivial details of a proof. Recent advances in the system such as new simplifiers and decision procedures may alleviate these problems. However, more work is needed to bring the level of interaction with the system closer to that of an informal proof.

4.6 Errors

No errors were discovered in the fabricated hardware. Errors that had inadvertently been introduced in the structural specifications (and could just as easily have been in the implementation) were discovered. The original versions of the behavioural specifications of many modules contained errors.

A strong indication of the source of detected errors was obtained. Because each module was verified independently, the source of an error was immediately narrowed down to being in the current module, or in the specification of one of its submodules. Furthermore, because performing the proof involves understanding why the design is correct, the exact location of the error was normally obvious from the way the proof failed. For example, in one of the dataswitch modules, two wires were inadvertently swapped. This was discovered because the subgoal `([T, F] = [F, T])` was generated in the proof attempt. One side of this equality originated from the behavioural specification and one from the structural specification. It was clear from the context of the subgoal in the proof attempt that two wires were crossed. It was also clear which signals were involved. It was not immediately clear which specification (structural or behavioural) was wrong.

A further example of an error that was discovered concerned the time the grant signal was read by the dataswitch. It was specified that the two bits of the grant signal from each arbiter were read on a single cycle. However, the implementation read them on consecutive cycles. This resulted in a subgoal of the form $grant\ t$ = $grant\ (t+1)$. No information was available in the goal to allow this to be proven, suggesting an error. On this occasion it was in the specification.

Occasionally, false alarms occurred: an unprovable goal was obtained, suggesting an error. However, on closer inspection it was found that the problem was that information had been lost in the course of the proof. For example, if an assumption, $t_1 < t_2$, is converted to $t_1 \leq t_2$ during the proof, the information that the two times are not equal is lost. Such a false alarm could lead to an unnecessary change in the implementation being made.

Many trivial typing errors were caught at an early stage by type-checking. However, many other trivial mistakes were made over the size of words and signals. For example, words of size 4 by 2 were inadvertently specified as 2 by 4 words. These errors were found during the proof process. It would have been much better if they had been picked up earlier. This would have been possible if dependent typing had been available [16].

4.7 Scalability

In theory, the HOL proof approach is scalable to large designs. Because the approach is modular and hierarchical, increasing the size of the design does not necessarily increase the complexity of the proof. However, in practice the modules higher in the hierarchy generally take longer to verify. This is demonstrated by the fact that two of the upper most modules took approximately half of the total verification time – a matter of weeks. However, it should be noted that the very top module which simply added various delays to various inputs and outputs of the main module, only took a day to verify. It is, thus, not universally so.

The extra time arises in part because there are more cases to consider. The situation is made worse if the interfaces between modules are left containing a large amount of low-level detail. For example, in the proof of the switch fabric, low-level modules required assumptions to be made about their inputs. These assumptions had to be dealt with in the proofs of higher-level modules adding extra proof work manipulating and discharging them. If the proof is to be tractable for large designs, it is important that the interfaces between modules are as clean as possible. The interfaces of the Fairisle fabric could have been much simpler. We demonstrated this by redesigning the fabric with cleaner interfaces. The new design was also verified [8]. The scalability of the approach beyond the level of the 4×4 fabric is also demonstrated by the fact that a 16×16 delta fabric built from 4×4 elements has also been verified [8]

5 The MDG Verification of the Fabric

In the second study, the same circuit was verified with the MDG System using a decision graph approach. Gate and register-transfer level (RTL) structural specifications of the whole fabric were written and verified to be equivalent. A generic version of the RTL specification was then verified against a high-level behavioural description of the full fabric. Safety properties were also specified and verified against the other descriptions.

5.1 The MDG Verification System

The MDG System is based on a new class of decision diagrams called *multiway decision graphs* (MDGs). MDGs are used to represent sets of states as well as the transition and output relations [5, 29]. Based on a technique called *abstract implicit enumeration*, hardware verification tools have been developed which perform combinational circuit verification, property checking and equivalence checking of two sequential machines [5].

The formal system underlying MDGs is many-sorted first-order logic augmented with a distinction between abstract and concrete sorts. Concrete sorts have enumerations, while abstract sorts do not. A data value can be represented by a single variable of abstract sort, rather than by concrete boolean variables. A data operation can be represented by an uninterpreted function symbol. A multiway decision graph (MDG) is a finite directed acyclic graph (DAG) where the leaf nodes are labelled by formulas, the internal nodes are labelled by terms, and the edges issuing from an internal node are labelled by terms of the same sort. MDGs essentially represent relations rather than functions.

MDGs must be *reduced* and *ordered* in a similar way to Bryant's ROBDDs [2]. The MDG system is based on a carefully chosen set of well-defined conditions which turn MDGs into canonical representations that can be manipulated by efficient algorithms. Algorithms for disjunction, relational product (combination of conjunction and existential quantification), pruning by subsumption (for testing of set inclusion) and reachability analysis (using abstract implicit enumeration [5]) have been developed. In addition, a rewriting ability (unconditional and conditional) is provided. It extends the scope of these applications and can also be used to shrink the MDG size. MDGs permit the description of the output and next state relations of a state machine in a similar way to the way ROBDDs do for FSMs. The model is called an *abstract state machine* (ASM), since it may represent an unbounded class of FSMs, depending on the interpretation of the abstract sorts and operators. For circuits with large datapaths, MDGs are thus much more compact than ROBDDs. As the verification is independent of the width of the datapath, the range of circuits that can be verified is greatly increased.

Like ROBDDs, the MDGs require a fixed node ordering. The variable ordering plays an important role as it determines the canonical attribute of the graphs and the size of the graphs which greatly affects its efficiency [2]. In contrast to VIS which provides heuristics for several node ordering techniques including dynamic ordering, the node ordering in MDG currently has to be given by the user explicitly. Unlike ROBDDs where all variables are boolean, every variable used in the MDGs must be assigned an appropriate sort, and type definitions must be provided for all functions. Rewrite rules may need to be provided to partially interpret the otherwise uninterpreted function symbols. Because of the use of uninterpreted functions, reachability analysis on MDGs may not terminate in some cases when circuits include some specific cyclic behaviour [5]. We did not encounter this problem in the current study.

The MDG tools provide a set of verification applications, including *combinational verification* (equivalence checking of input–output relations for two combinational circuits using the canonicity of MDGs), *invariant checking* (checking if a certain invariant holds in all the reachable states of a sequential machine), *sequential verification* (checking behavioural equivalence of two sequential machines by performing reachability analysis on their product machine), and *model checking* (checking of first-order linear time temporal logic properties based on reachability analysis of MDGs [29]).

When an invariant is not satisfied during the verification process, a counter-example is provided to help with identifying the source of the error. A counter-example consists of a list of assumptions, inputs and state values at each clock cycle, and gives a trace for the erroneous output.

The MDG operators and verification procedures are packaged as MDG tools implemented in Prolog [30]. These MDG tools 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 [5]. In this paper, we investigate the verification of a real circuit–the Fairisle ATM switch fabric. This circuit is an order of magnitude larger than any other circuit verified using MDGs.

5.2 The Structural Specifications

We described the actual hardware implementation of the switch fabric at two levels of abstraction. We gave a description of the original Qudos gate-level implementation and a more abstract Register transfer Level (RTL) description which holds for an arbitrary word width.

As with the HOL study, we translated the Qudos HDL gate-level description into a suitable HDL description, here a Prolog-style HDL, called MDG-HDL. As in the HOL study, extra modularity was added over the Qudos descriptions, while leaving the underlying implementation unchanged. A structural description is usually a (hierarchical) network of components (modules) connected by signals. The MDG-HDL comes with a large library of predefined, commonly used, basic components (such as logic gates, multiplexers, registers, bus drivers, ROMs, etc.) Multiplexers and registers can be modeled at the Boolean or the abstract level using abstract terms as inputs and outputs.

As an example, the following is the MDG-HDL description of the `DMUX4T2` module given in Section 4.2:

$$
\begin{array}{l}
\textbf{module}(\texttt{DMUX4T2}\\
\quad \textbf{port}(\textbf{inputs}((d0, bool), (d1, bool), (d2, bool), (d3, bool)), (x, bool)),\\
\qquad\quad \textbf{outputs}((dOut0, bool), (dOut1, bool))),\\
\quad \textbf{structure}(\\
\qquad\quad \textbf{signals}(xBar, bool),\\
\qquad\quad \textbf{component}(\texttt{InvX}, \texttt{NOT}(\textbf{input}(x), \textbf{output}(xBar))),\\
\qquad\quad \textbf{component}(\texttt{AO\_0}, \texttt{AO}(\textbf{input}(d0, xBar, d1, x), \textbf{output}(dOut0))),\\
\qquad\quad \textbf{component}(\texttt{AO\_1}, \texttt{AO}(\textbf{input}(d2, xBar, d3, x), \textbf{output}(dOut1))))).
\end{array}
$$

Here, the components `NOT` and `AO` are basic components provided by the MDG-HDL library. Note also that the data sorts of the interface and internal signals must always be specified. MDG does not provide a replication facility equivalent to `FOR` nor an ability to structure words, so this description cannot be simplified (abstracted) as in HOL.

Besides the gate-level description, we also provided a more abstract (RTL) description of the implementation which holds for arbitrary word width. Here, the data-in and data-out lines are modeled using an abstract sort *wordn*. The *active*, *priority* and *route* fields are accessed through corresponding cross-operators (functions). In addition to the generic words and functions, the RTL specification also abstracts the behaviour of the dataswitch unit by modeling it using abstract data multiplexers instead of logic gates. We thus obtain a simpler implementation model of the dataswitch which reflects the switching behaviour in a more natural way and is implemented with fewer components and signals. For example, a set of four `DMUX4T2` modules is modeled using a single multiplexer component. For more details about the abstraction techniques used, refer to [27].

5.3 The Behavioural Specifications

MDG-HDL is also used for behavioural descriptions. A behavioural description is given by high-level constructs as ITE (If-Then-Else) formulas, CASE formulas or tabular representations. The tabular constructor is similar to a truth table but allows first-order terms in rows. It can be used to define arbitrary logic relations. In the MDG study, we gave the behavioural specification of the switch fabric in two different forms: (1) as a complete high-level behavioural state machine and (2) as a set of properties which reflect the essential behaviour of the switch fabric as it is used in its environment.

5.3.1 ASM Behavioural Specification

Starting from timing-diagrams describing the expected behaviour of the switch fabric, we derived a complete high-level behavioural specification in the form of an abstract state machine (ASM). This specification was developed independently of the actual hardware design and includes no restrictions with respect to the frame size, cell length and word width. It assumes that the environment maintains certain timing constraints on the arrival of the frame start signal and headers, however. This ASM reproduces the exact behaviour of the switch fabric during the initialization phase, the arrival of a frame start, the arrival of the routing bytes, and the end of the frame. The generation of the acknowledgment and data output signals is described by case analysis on the result of the round-robin arbitration. This is done in MDG-HDL using ITE and tabular constructs.

A schematic representation of the ASM specification of the 4×4 switch fabric is shown in Figure 8. The symbols t_0, t_s, t_h and t_e in the figure represent the initial time, the time of arrival of the frame start signal, the time of arrival of the routing bytes and the time of the end of a frame, respectively. There are 14 conceptual states. States 0, 1 and 2 along the time axis t_0 describe the initial behaviour of the switch fabric. States 2, 3, 4 and 5 along the time axis t_s describe the behaviour of the switch on the arrival of a frame start signal. States 6 to 13 along the time axis t_h describe the behaviour of the switch fabric after the arrival of the headers. Waiting loops in states 2, 5 and 10 are illustrated in the figure by the non-zero natural numbers i, j and k, respectively. Figure 8 also includes many meta-symbols used to keep the presentation of the diagram simple. For instance, the symbols s and h denote a frame start and the arrival of a routing byte (header), respectively, and the symbol "$\sim$" denotes negation. The symbols a, d and r inside a conceptual state represent the computation of the acknowledgment output, the data output and the round-robin arbitration, respectively. The absence of an acknowledgment or a data symbol means that no computation takes place and the default value is output. The operations are defined by separate state machines.

To formally describe this ASM using MDGs, we first introduced some basic sorts, constants and functions (cross-operators), e.g. a concrete sort $port = \{0, .., 3\}$, an abstract sort $wordn$, a constant $zero$ of sort $wordn$ and a cross-operator rou of type $[wordn \rightarrow port]$ representing the route field in a header. Further, the generation of the acknowledgment and data output signals is described by case analysis on the

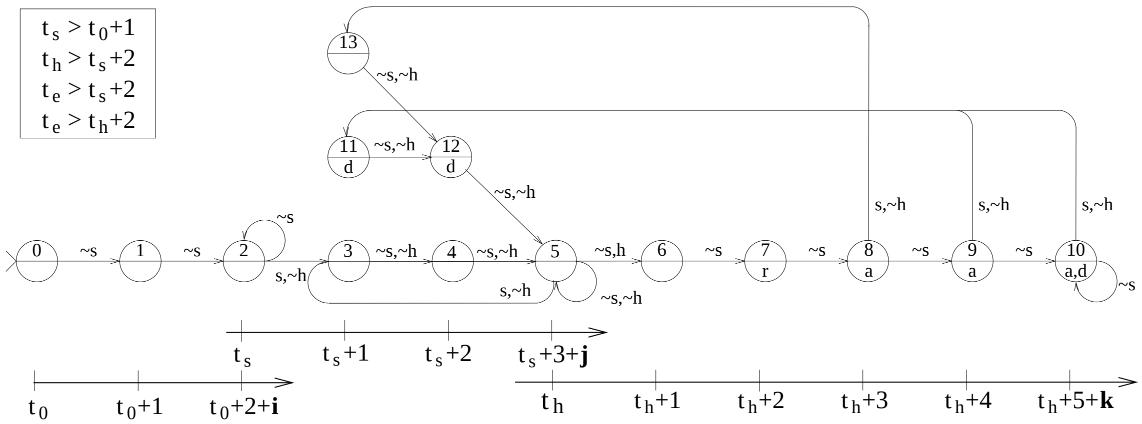

Figure 8: ASM Behavioural Specification

result of the round-robin arbitration. This is done in MDG-HDL using if-then-else constructs. For example, the acknowledgment output is described by four formulas determining the value of $ackOut_i$, $i \in \{0, .., 3\}$:

if $((co_0 = 1)$ **and** $(ip_0 = i))$ **then** $(ackOut_i = ackIn_0)$
ef $((co_1 = 1)$ **and** $(ip_1 = i))$ **then** $(ackOut_i = ackIn_1)$
ef $((co_2 = 1)$ **and** $(ip_2 = i))$ **then** $(ackOut_i = ackIn_2)$
ef $((co_3 = 1)$ **and** $(ip_3 = i))$ **then** $(ackOut_i = ackIn_3)$
else $(ackOut_i = 0)$

Here co_i $(i \in \{0, .., 3\})$, of sort *bool*, and ip_i $(i \in \{0, .., 3\})$, of sort *port*, are state variables generated by the round-robin computation which correspond to the output disable and grant signals, respectively (Figure 3).

5.3.2 Specification of Properties

Although the above ASM specification describes the complete behaviour of the switch fabric, we also provided a set of properties which reflect the essential behaviour of the switch fabric, e.g., for checking of correct priority computation, circuit reset or data routing. These were used in an early stage of the project to validate the fabric specification and implementation. If we consider the behaviour of the fabric when operating in the intended real Fairisle switch environment, its cyclic behaviour can be simulated as an *environment state machine* having 68 states as shown in Figure 9. It can be checked that this state machine is an instance of the general timing state machine (Figure 8) with cell length of 53 and frame size of 64. The machine generates the frame start signal, *frame start*, the headers, h, and the data, d, in the states as indicated in Figure 9. Normally, d is a fresh abstract variable representing data in the cell; and h can be instantiated according to the property to be verified. This diagram allowed us to map the time points t_0, t_s, t_h and t_e to specific states, e.g. t_s to states 3 or 66; t_h to state 12; and t_e to state 66.

Based on this environment state machine, we described the properties as invariants which should hold in all reachable states of the specification model. In the

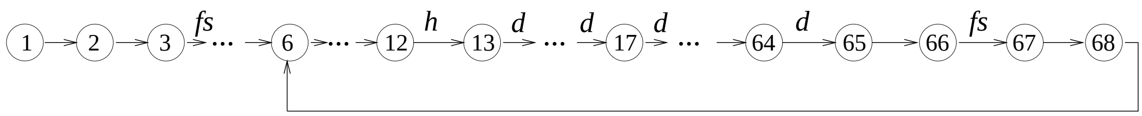

Figure 9: The Environment State Machine of the Fairisle ATM

following, we give an example property, P, which checks for correct routing to port 0. More precisely:

P: 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 port has its priority bit set, then the value on $dataOut0$ will be equal to the value of $dataIn0$ four clock cycles earlier.

Let s be a state variable of the environment state machine of a concrete sort having the enumeration [1..68]. P is expressed in MDG-HDL using an ITE construct as:

P: **if** $(s \in \{17,..,68\})$ **and** $priority[0..3] = [1,0,0,0]$ **and** $route[0] = 0$
then $dataOut[0] = dataIn'[0]$

where $priority[0..3]$ indicates the priority bits for all input ports, $route[0]$ represents the routing bits for input port 0 and $dataIn'[0]$ is the data input on port 0 delayed by 4 clock cycles. Further examples of properties are described in [27].

5.4 The Verification Process

Using a hierarchical approach, we first verified the original gate-level implementation of the switch fabric against the RTL implementation. We then verified the RTL implementation against the behavioural specification given as an abstract state machine (ASM). We thus obtained complete verification from high-level behaviour down to the gate level. In an early stage of the project, we also verified some specific properties that reflect the behaviour of the fabric in its real operating environment as described above.

5.4.1 Equivalence Checking

The correctness of equivalent behaviour between the original Qudos gate-level implementation and the abstract (RTL) hardware model is established if the two machines produce the same data outputs for all input sequences. This, however, cannot be done for an arbitrary word size n since the gate-level description is not generic. We hence instantiate the data signals of the abstract model to be 8 bits wide. This can be realized within the MDG environment using uninterpreted functions which encode and decode abstract data to boolean data and vice-versa [27]. For instance, decoding is realized using 8 uninterpreted functions bit_i (i: 0..7) of type $[wordn \rightarrow bool]$, which extract the ith bit of an n-bit data word. We hence decode the 4 n-bit data lines to a 32-bit bundle. Encoding, on the other hand, is done using one uninterpreted function *concat*8 of type $[(bool \times \cdots \times bool) \rightarrow wordn]$ which concatenates any 8 boolean signals to a single word and thus encodes a bundle of 32 boolean data signals to 4 signals of sort $wordn$. In addition, we used a few rewriting rules to map 8-bit constants of concrete sort to generic ones of abstract sort. Using the sequential equivalence checking facility of the MDG tools, we verified that the abstract machine is equivalent to the original gate-level one for a word size equal to 8, i.e.

$$\textit{Gate-level structure} \;\equiv\; RTL_{(8)} \; \textit{structure} \qquad (1)$$

where $RTL_{(8)}$ means the 8-bit version (instance) of the n-bit RTL implementation. Here we mean equivalence in the sense described for MDG sequential verification in Section 5.1. Note that since the data abstraction affects only the dataswitch unit, the verification reduced to the equivalence of the dataswitch blocks at the two levels.

Based on implicit reachability analysis, we checked the equivalence of the behavioural ASM specification against the RTL hardware model when both are seen as abstract state machines. That is, we ensured that the two machines produce the same observable behaviour by feeding them with the same inputs and checking that an invariant stating the equivalence of their outputs holds in every state using reachability analysis of the product machine [5]. For this product machine, an MDG representing a set of total states encodes a relation between 39 concrete and 36 abstract state variables [27]. The relation may depend on data values, encoded using cross-terms, however. In ROBDDs, 8 boolean variables would be needed for each abstract variable of the MDGs (i.e., 288 boolean variables for data). In MDGs, the encoding is done using abstract data, yet isomorphic graph sharing is exploited as in ROBDDs. Decisions on values of abstract data are represented by cross-terms which also compose nodes in the MDGs. 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. Using abstract reachability analysis, the verification succeeded for an arbitrary word width, n, and any frame size and cell length that respect the environment assumptions of the specification, i.e.

$$\begin{aligned}&\textit{assumptions on environment} \supset \\ &\qquad (RTL_{(n)} \; \textit{structure} \;\equiv\; ASM_{(n)} \; \textit{behaviour})\end{aligned} \qquad (2)$$

$RTL_{(8)}$ is an *instance* of $RTL_{(n)}$. The two descriptions provide exactly the same semantics. They differ only in the syntactic use of the abstract data sort $wordn$ instead of the concrete sort $word8$. The same reasoning is true for a behaviour $ASM_{(8)}$ which is an 8-bit instance of the behaviour $ASM_{(n)}$. From the n-bit generic result in (2), we hence deduce through instantiation:

$$\begin{aligned}&\textit{assumptions on environment} \supset \\ &\qquad (RTL_{(8)} \; \textit{structure} \;\equiv\; ASM_{(8)} \; \textit{behaviour})\end{aligned} \qquad (3)$$

By combining the two verification steps (1) and (3), we hierarchically obtain a complete verification of the switch fabric from a high-level behaviour down to the gate-level implementation, i.e.

$$\begin{aligned}&\textit{assumptions on environment} \supset \\ &\qquad \textit{Gate-level structure} \equiv ASM_{(8)} \textit{ behaviour}\end{aligned} \tag{4}$$

In summary, thanks to the management of the proof in two steps and to the independence of the second verification step from the datapath width, we have been able to avoid a state explosion induced by data. Note, however, that we have not formally shown, using the MDG tools, the meta-rewriting for theorem (4) nor the instantiation in theorem (3). The experimental results on a SPARC station 10 are recapitulated below in Table 1, including the CPU time, memory usage and the number of MDG nodes generated.

5.4.2 Property Checking

Prior to the full verification, we also checked both behavioural and RTL structural specifications against several specific safety properties of the switch. This is useful as it gives a quick confidence check at low cost. To verify the properties (invariants), we compose the fabric with both the environment state machine described above and an additional delay circuit used to remember the input values that are to be compared with the outputs. This allows us to state the properties in terms of the equality between input and output signals. Combining these machines, we obtain the required platform for checking if the invariant properties hold in all reachable states of the specification [27]. Experimental results for the verification of four example properties are shown in Table 1 (where the previously described property P is labelled $P3$). Although the properties we verified do not represent the complete behaviour of the switch fabric, we were able to detect several injected design errors in the structural description.

5.5 Time Taken

The user time required for the specification and verification is hard to determine since it included the improvement of the MDG package, writing documentation, etc. The figures given here are therefore estimates. The translation of the Qudos design description to the MDG-HDL gate-level structural model was straightforward and took about one person-week. The description of the RTL structural specification including modeling required about one person-week. The time spent for understanding the expected behaviour and writing the behavioural specification was about one person-week. The time taken for the verification of the gate-level description against the RTL model, including the adoption of abstraction mechanisms and correction of description errors, was about two person-weeks. The verification of the RTL structural specification against the behavioural model required about one person-week of work. The user time required to set up four properties, build the environment state machine, conduct the property checking on the structural specification and interpret the results was about one person-week. Checking of these same properties on the behavioural specification took about one hour. The average time for the injection and verification of an introduced design error was less than one person-hour. The experimental results in machine time are shown in Table 1 which gives the CPU time (on a SPARC station 10), memory usage and the number of MDG nodes generated.

Verification	CPU Time (s)	Memory (MB)	MDG Nodes Generated
Gate-Level to RTL	183	22	183300
RTL to Beh. Model	2920	150	320556
P1: Data Output Reset	202	15	30295
P2: Ack. Output Reset	183	15	30356
P3: Data Routing	143	14	27995
P4: Ack. Output	201	15	33001
Error (i)	20	1	2462
Error (ii)	1300	120	150904
Error (iii)	1000	105	147339

Table 1: Experimental Results for the MDG Verification

Like ROBDDs, the MDGs require a fixed node ordering. The variable ordering plays an important role as it determines the canonical attribute of the graphs and the size of the graphs which greatly affects its efficiency. A bad ordering easily leads to a state space explosion as occurred after an early ordering attempt. In contrast to VIS which provides heuristics for several node ordering techniques including dynamic ordering, node ordering in MDG has to be given by the user explicitly. This takes much of the verification time. On the other hand, unlike ROBDDs where all variables are Boolean, time must be spent assigning to every variable used an appropriate sort and type definitions must be provided for all functions. In some cases, rewrite rules may need to be provided to partially interpret the otherwise uninterpreted function symbols.

Because the verification is essentially automatic, the amount of work re-running a verification for a new design is minimal compared to the initial effort since the latter includes all the modeling aspects. Much of the effort is spent on determining a suitable variable ordering. Depending on the kind of design changes adopted, it is not obvious if the original variable ordering could still be used on a modified design without major changes.

The MDG gate level specification is a concrete description of the fabricated implementation. In contrast, the RTL structural and ASM behavioural specifications are generic. They abstract away from frame, cell and word sizes, provided the environment timing assumptions are kept. Design implementation changes at the gate-level that still satisfy the RTL model behaviour would hence not affect the verification against the ASM specification. For property checking, specific assumptions about the operating environment were made, (e.g., that the frame interval is 64 cycles). This is sound since the switch fabric will in fact be used under the behest of its operating environment (the port controllers). However, while this reduces the verification cost, it has the disadvantage that the verification must be completely redone if the operating environment changes. Still, the work required is minor as only a few parameters have to be changed in the description of the (simple) environment state machine described above.

5.6 Errors

As with the HOL study, no errors were discovered in the implementation. For experimental purposes, however, we injected several errors into the structural specifications and checked them using either the set of properties or the behavioural model. Errors were automatically detected and identified using the counter-example facility. The injected errors included the main errors introduced accidently in the HOL study, discussed in Section 4.6. We summarize here three further examples. (i) We exchanged the inputs to the JK Flip-Flop that produces the output disable signal. This prevented the circuit from resetting. (ii) We used, at one point, the priority information of input port 0 instead of input port 2. (iii) We used an AND gate instead of an OR gate within the acknowledgment unit, thus producing a faulty $ackOut[0]$ signal. Experimental results for these three errors, which have been checked by verifying the RTL model against the behavioural specification, are reported in Table 1.

While checking properties on the hardware structural description, we also discovered some errors that we mistakenly introduced in the structural specifications. However, we were able to easily identify and correct these errors using the counter-example facility of the MDG tools. Also, during the verification of the gate-level model, we found a few errors in the description that were introduced during the translation from Qudos HDL to MDG-HDL. These were easily removed by comparing both descriptions, since they included the same collection of gates. Finally, many trivial typing errors were highlighted at an early stage of the description process by the error messages output after each compilation of the specification's components.

5.7 Scalability

Like any FSM-based verification system, the MDG proof approach is not directly scalable to large designs. This is due to the possible state space explosion that results from large designs. Unlike other ROBDD-based approaches, however, MDGs do not need to cope with the datapath complexity since they use data of abstract sort and uninterpreted functions. Still, a direct verification of the gate-level model against the behavioural model or even against the set of properties is practically impossible. We overcame this problem by providing an abstract RTL structural specification which we instantiated for the verification of the gate-level model. To handle large designs, major efforts are in general required to set up the appropriate model abstraction levels.

6 The VIS Verification of the Fabric

In the third study [20] the VIS tool [3] was used to perform property checking on various abstracted models of the fabric. In addition, equivalence checking was conducted between behavioural and structural specifications of sub-modules of the fabric written in Verilog. The whole fabric was also re-implemented using the Synopsys synthesis tool and all generated modules graphically simulated using the Verilog-XL simulator of Cadence.

6.1 The VIS Verification System

VIS [3] is a decision diagram based tool that integrates the verification, simulation and synthesis of finite-state hardware systems. It uses a Verilog front-end and supports fair CTL model checking, language emptiness checking, combinational and sequential equivalence checking, cycle-based simulation, and hierarchical synthesis. Its fundamental data structure is a multi-level network of latches and combinational gates. The variables of a network are multi-valued, and logic functions over these variables are represented by an extension of BDDs: multi-valued decision diagrams.

VIS operates on the intermediate format BLIF-MV. It includes a compiler from Verilog to BLIF-MV. It extracts a set of interacting FSMs that preserves the behaviour of the Verilog program defined in terms of simulated results. Through the interacting FSMs, VIS performs fair CTL model checking under Buchi fairness constraints. The language of a design is given by sequences over the set of reachable states that do not violate the fairness constraint. Also VIS can check the combinational and sequential equivalence of two designs. Sequential verification involves building the product FSM, and checking whether a state where the values of corresponding outputs differ can be reached from the set of initial states of the product machine. If model checking or equivalence checking fails, VIS reports the failure with a counter-example.

6.2 The Structural Specifications

The Verilog structural specification of the fabric is very similar to the other descriptions. A big advantage of the VIS Verilog front-end is the ease of importing existing (industrial) designs with no extra overhead of manual translation. Moreover, it allows the direct interaction of VIS with other commercial tools for simulation and synthesis. However, the fabric structure had to be reduced to 4 bits and the datapath further to 1 bit to enable the model checking procedure to terminate. The control path could not be reduced below 4 bits as the data includes the header control information. For more details about the abstraction and reduction techniques adopted refer to [20].

The following is the Verilog description of the `DMUX4T2` module given in Section 5.2. CLBMAP5i20 and `AO` is an AND-OR module defined in terms of Verilog library components:

```
module DMUX4T2(d,x, dOut);
input [3:0] d; input x; output [1:0] dOut; wire xBar;
begin CLBMAP5i20 Clb(d[1:0],x,d[3:2],dOut[1:0]);
      not InvX (xBar, x);
      AO B0 (d[0],xBar,d[1],x,dOut[0]);
```

```
    AO B1 (d[2],xBar,d[3],x,dOut[1]);
end;
```

6.3 The Behavioural Specifications

We gave the behavioural specification of the fabric in two forms: an RTL description as a state machine of the whole fabric and a set of liveness and safety properties covering its essential behaviour. In addition, as with HOL, behavioural specifications of submodules of the design hierarchy were developed. VIS-Verilog HDL is used for behavioural specification. It contains two new features over standard Verilog: a nondeterministic construct, \$ND, to specify non-determinism on wire variables; and symbolic variables which use an enumerated type mechanism similar to the one available in the MDG system.

As an example, consider the specification of the timing module (Figure 10), which determines when the arbitration unit is triggered. The module has the following behaviour. The *routeEnable* signal is normally low. After the *frameStart* signal goes high, it waits until any of the active bits (*anyActive* signal) goes high for one cycle, returning to low until the next frame. We use symbolic variables to express the timing states: RUN, WAIT and ROUTE.

```
typedef enum { RUN, WAIT, ROUTE } timing _state;
module TIMING (frameStart, clock, anyActive, routeEnable) ;
...
always   (posedge clock)   begin   case   (state)
  RUN: if (frameStart == 1)   state = WAIT;
  WAIT: if ((frameStart == 0) && (anyActive == 1))   state = ROUTE;
  ROUTE: begin if (frameStart = 0) && ( anyActive == 1 ))   state = ROUTE;
  else   state = WAIT;
endcase   end
endmodule;
```

Both behavioural and structural specification were written in Verilog, so we were able to perform their simulation in Verilog-XL directly. It was very useful for detecting some syntax and semantic errors of the descriptions before performing equivalence or model checking. In addition, we extracted some safety properties from the generalization of simulation vectors. These safety properties were further used in model checking, enabling the detection of design errors that were omitted by simulation. The Verilog-XL graphical interface also eased the analysis of counter-examples which were generated by model and equivalence checking. Furthermore, as the RTL behavioural specification was written in Verilog, we were able to synthesize the structural specification with some timing constraints directly using the Synopsys Design Compiler. We performed equivalence checking between the submodules of the RTL behavioural specification and the submodules of the synthesized structural one to ensure the correctness of the synthesis.

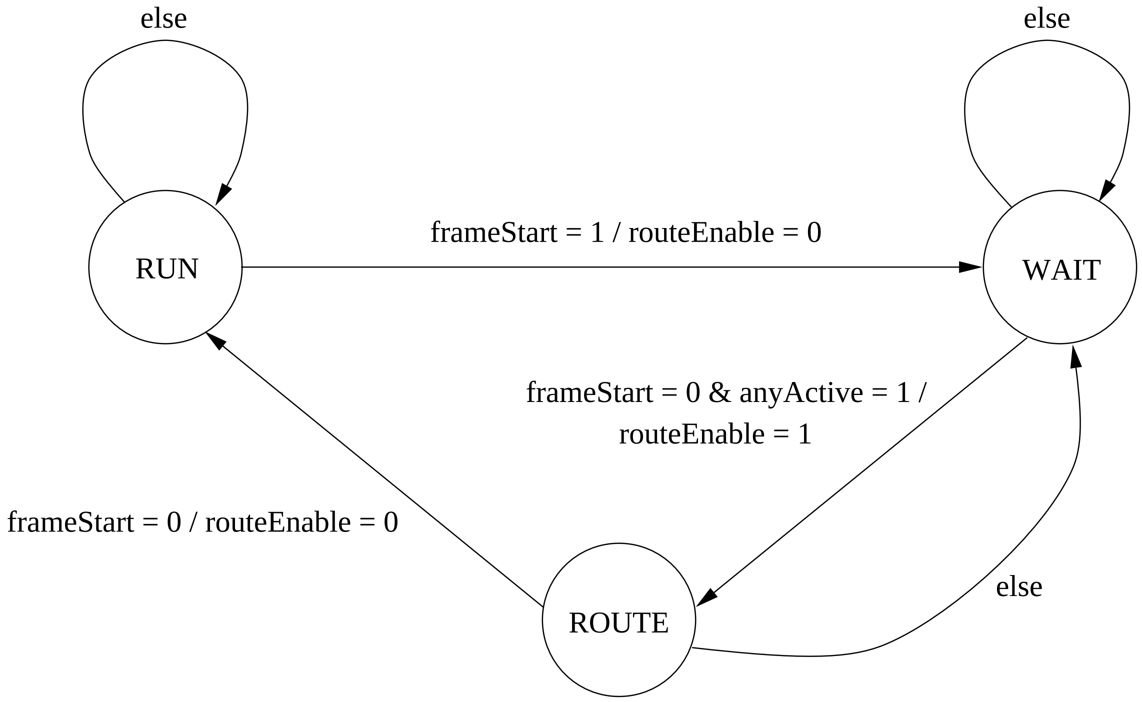

Figure 10: Behavioural FSM of the Fabric Timing Block

6.4 The Verification Process

As mentioned in the previous sections, we translated the original Qudos HDL gate-level description of the switch fabric into Verilog HDL. We also derived a complete high-level behavioural specification in the form of a finite state machine according to the timing diagrams describing the expected behaviour of the switch fabric. This specification was developed independently of the actual hardware design and uses a different design hierarchy to the structural one. Using these Verilog specifications, we attempted to obtain a complete verification of the switch fabric from a high-level behavioural specification down to the gate-level implementation through equivalence checking. This verification was similar to that in the MDG case. However, it did not succeed in VIS due to state space explosion. We therefore attempted to separately verify the submodules of the fabric based on the same design hierarchy as the structural one. This is similar to the HOL study, and involved writing separate Verilog RTL behavioural specifications for each submodule. We succeeded in verifying the equivalence of the behavioural specification of each submodule and its corresponding structural specification by VIS sequential equivalence checking. Through this verification, we checked that the implementation of each submodule satisfies its specification. Unlike the HOL verification, we could not verify the correctness of the connections among the submodules of the switch fabric. For real designs, this step would be useful to verify if the logic synthesis is correct.

As an alternative to equivalence checking, we attempted model checking of properties of the switch fabric. Unlike MDG, property checking is the main verification approach in VIS. Model checking needs to be performed on a closed system [19].

However, the switch fabric is not a closed machine. We thus built an environment state machine [20] based on the behaviour of the port controller. This environment state machine is similar to the environment described in Section 4.2, except that we compressed the 68 states into 7 states in order to ease the model checking in VIS. Again we failed to verify the whole switch fabric due to the state space explosion. We succeeded in model checking a simplified fabric with its datapath and control path reduced from 8 bits to the minimum 1 bit and 4 bits, respectively.

6.4.1 Property Checking

Unlike in MDG, we extensively used property checking to verify the fabric in VIS as it is optimized for model checking. Moreover, thanks to the expressiveness of CTL, properties can be defined more easily in VIS. With MDG a property (invariant) is described in MDG-HDL using ITE and tabular constructs. Before using model checking to verify the overall behaviour of the switch fabric, we set up an environment state machine and developed a set of properties. The nondeterministic construct ($ND) of VIS-Verilog HDL eases the establishment of an environment state machine. We used it to express the inputs of the switch fabric. CTL can represent both safety and liveness properties. The latter can be used to detect deadlock or livelock which is difficult using simulation. 58 CTL properties were verified. We first verified a number of safety properties including all those used in the MDG study. In addition, we verified many CTL liveness properties. Example properties that we checked on the fabric model can be found in [20].

For instance:

```
AG ((dIn0[3:0]=0011 * dIn1[1]=0 * dIn2[1]=0 * dIn3[1]=0 * state=S2)
-> AX AX AX AX AX (dOut0==dIn0));
```

is a CTL formula equivalent to the property described in Section 5. Here $dIn[0]$ indicates the active bit for input port 0, $dIn[2]$ and $dIn[3]$ represent the routing bits for input port 0. $state = S2$ denotes the state when the routing tag arrives. $dIn0$ and $dOut0$ express the input data cell and output data cell, respectively. We can also express this particular behaviour as a liveness property as follows:

```
AG(dIn0[0]=1 * dIn0[2]=0 * dIn0[3]=0 * state=S2 -> EF(dIn0==dOut0));
```

Due to the state space explosion, we succeeded in checking only a few properties on the abstracted fabric directly. Instead, we adopted several techniques that divide a property into sequentially or parallelly related sub-properties in a similar manner to the compositional reasoning proposed in [19]. Details about the specific property division techniques we used are reported in [20].

6.4.2 Equivalence Checking

Besides property checking, VIS supports combinational and sequential equivalence checking of two circuits. Since we did post-design verification of the switch fabric, we attempted sequential equivalence verification between the Verilog structural description (which we translated from the original Qudos HDL implementation), and the Verilog behavioural description of the fabric based on its FSM behaviour specification. If both descriptions are equivalent, the correctness of the fabric is proved. We first provided a complete behavioural description of the whole switch fabric as one module and tried to verify its equivalence against the implementation of the whole fabric including all connections of submodules. However, we did not succeed in verifying it in VIS after three days of continuous execution on a Sun Sparc 20 workstation due to state space explosion, even though we used different variations of model abstraction, combination with environment state machine, dynamic ordering, etc. We hence followed a second approach that modularizes the fabric to several parts that are similar to the hierarchical modules of the structural description, where each module will be, in addition, described in terms of its behaviour specification. This second approach has the shortcoming that while we are able to check the correctness of separate submodules of the fabric structure, it is difficult to ensure the correctness of the network connecting all the submodules. This shortcoming is overcome by simulation and property checking. This second approach, however, has the advantage that the developed behavioural descriptions of the submodules are close to that used in industrial design synthesis, and hence fits in with on-the-fly verification.

For the Fairisle switch fabric (see Figure 3), we verified the sequential equivalence of the Timing, Priority_decode, Arbiters and Arbitration modules. In addition, we checked the combinational equivalence of the Acknowledgment module. The verification of the Arbitration module consumed an excessive amount of CPU time (see Table 3). We also failed to verify in VIS the Dataswitch module and obviously the whole fabric. Table 3 gives the verification CPU time and number of latches for the modules verified using equivalence checking.

6.5 Time Taken

The translation of the Qudos structural description to Verilog was straightforward taking about three person-days. The time spent for understanding the expected behaviour and writing the behavioural specification was around ten person-days. The time taken for the simulation of both RTL behavioural and structural specification in Verilog-XL, including the development of test-bench files, was about three person-days. The verification of the RTL behavioural specification against the structural specification was done automatically, and took around one person-day. The user time required to set up 58 CTL properties, build the related environment state machine, construct the appropriate property division and conduct the model checking took approximately three weeks. The injection and verification of an error took less than one hour.

The experimental results of model checking, which were obtained on a Sun Sparc 20, are shown in Table 2. VIS generates comparatively more BDD nodes than the MDG system does. This is due to the data abstraction within MDG that is absent in VIS. The equivalence checking of the whole switch fabric ran for three days before running out of memory. The same problem occurred with the dataswitch module.

Equivalence checking of the arbitration module was successful but it took two days of machine time. The lower level modules such as the timing unit were verified in seconds. We also failed to verify the properties on the original switch fabric after two days of machine time. Finally we reduced the datapath of the switch fabric from 8 to 1 bit. The successful model checking results in Table 2 are based on this reduced model.

Design errors can be detected by either equivalence or model checking. Like MDG, VIS provides a counter-example generation facility to help identify the source of design errors. Since VIS is based on ROBDDs, the node ordering has a dramatic influence on the speed of both equivalence checking and model checking. Unlike MDG, VIS provides dynamic ordering facilities to reduce the cost of manual variable ordering. The algorithm used is so efficient that it enhanced the model checking and equivalence checking by up to 15 times in our example.

The experimental results given in Table 2 were obtained using VIS dynamic ordering. It is to be noted that in some cases a manually optimized ordering, e.g., an interleaved order of the bits of the data words, would have enhanced the VIS verification.

Verification	CPU Time (s)	Memory (MB)	Nodes Generated
P1: Data Output Reset	3593.4	32.4	93,073,140
P2: Ack. Output Reset	833.0	4.5	28,560,871
P3: Data Routing	3679.7	40.9	79,687,784
P4: Ack. Output	414.8	5.3	4,180,124
Error (i)	82.5	3.5	1,408,477
Error (ii)	49.3	2.4	250,666
Error (iii)	15.4	1.1	85,238

Table 2: Experimental Results for the VIS Model Checking

Module	Number of Latches	CPU time (seconds)
Acknowledgement	0	1
Timing	2	1
Arbiters	12	13
Priority_decode	16	27
Arbitration	30	67860
Dataswitch	64	–
Switch_fabric	190	–

Table 3: Experimental Results for the VIS Equivalence Checking

We also applied cascade and parallel property divisions (practical approaches to compositional reasoning) [20]. Using these techniques, we enhanced the model

Errors	CPU time
Error (i)	20.6
Error (ii)	24.0
Error (iii)	1.7

Table 4: Experimental Results for the VIS Equivalence Checking with Errors

checking by up to 200 times. However, we had to establish environment state machines and abstract the circuit first. The derivation of reduced models from the original structure and the division of properties was very time consuming. For a cascade property division, we built a new partial environment state machine for each target sub-circuit. For parallel property division, we disassembled a circuit at different symmetric locations and later composed the properties.

6.6 Errors

As in the HOL and MDG studies, no errors were discovered in the switch fabric implementation. We injected the same errors as for MDG into the implementation and checked them using either model checking or equivalence checking. Experimental results are reported in Tables 2 and 4. Injected errors were automatically detected and, using the counter-example facility, further viewed graphically with Verilog-XL. Through checking the equivalence between the RTL behavioural and the structural specifications of the submodules, we discovered errors that we mistakenly introduced in the structural specification. Also, during model checking, we found connection errors that were mistakenly introduced in the RTL behavioural and structural specifications. We easily identified and corrected these errors from the counter-examples.

6.7 Scalability

The VIS proof approach is not directly scalable to large designs due to state space explosion. To solve this problem the datapath complexity must be decreased by abstraction and reduction. In a large design like the switch fabric, we also had to apply compositional reasoning [19]. The environment state machine must imitate the behaviour of the interfaced modules. It must also have fewer components than the original models. Consequently, the environment state machine is especially hard to develop when the concurrent interaction between the target model and its associated models is complex.

7 Conclusions

In the previous sections we have given overviews of the specification and verification of the Fairisle switch fabric using the 3 separate tools: HOL, MDG and VIS. In this section we directly compare the advantages of the three systems based on those experiences.

Structural Specification The structural descriptions are very similar. HOL provides significantly more expressibility allowing more natural specifications. Some generic features were included in the MDG description that were not in the HOL description. This could have been done with minimal effort, however. Due to its Verilog front-end, (commercial) designs can be imported into VIS with no extra overhead of a manual translation, which is one reason for its popularity. This also allows direct interaction with commercial tools for simulation and synthesis.

Behavioral Specification The behavioural descriptions are totally different. The MDG and VIS specifications are based on a state machine model while HOL's is based on interval operators explicitly describing the timing behaviour in terms of frames corresponding to whole ATM cells arriving. In the MDG and VIS specifications the frame abstraction is not used: the description is firmly at the byte level. Verilog allows direct testing of the specifications using commercial simulation facilities, however. Unlike Verilog descriptions, HOL's higher-order logic and the MDG-HDL descriptions are not directly executable. All describe the behaviour in a clear and comprehensive form. Writing the behavioural specifications took longer in HOL and VIS, as separate specifications were needed for each module. In MDG this was not necessary as the whole design was verified in one go. This also reduced the MDG verification time because fewer mistakes were made.

Property Checking An advantage of both MDG and VIS is that a property specification is easy to set up and verify. Expected operating conditions can be used to simplify this, even if the full specification is more general. For both systems it was necessary to introduce an environment state machine in order to restrict the possible inputs to the switch fabric. It is verified that the specification satisfies its requirements under specific working conditions. It can greatly reduce the full verification cost by catching errors at an early stage. In this respect VIS, with its very efficient CTL based model checking, outperforms its MDG counterpart. Properties are easier to describe in CTL than are invariants in the MDG system.

Verification Time The HOL verification was much slower, taking several months. This time includes the verification of each of the modules and of their combination. Much of the time was spent on the connection of the highest level modules (which VIS failed on). Using HOL, many lemmas had to be proved and much effort was required to interactively create the proof scripts. For example, the time spent verifying the dataswitch was about three days. The proof script was over 500 lines long (17 KB). The MDG and VIS verifications were achieved automatically without the need of a proof script. For MDG, however, careful management of the MDG node ordering was needed (which currently has to be done manually). This could take hours or a few days of work. In contrast, VIS provides several options for variable ordering heuristics which eliminate the ordering overhead. However, major effort was spent here developing abstract models of the switch fabric units to manage the state explosion of the boolean representation. Furthermore, the HOL and MDG verifications succeeded in verifying the whole switch fabric but VIS failed to verify even the smallest 1-bit datapath version of the fabric using equivalence checking. Additional time was spent hierarchically verifying submodules as with HOL but their combination could not be verified.

In the MDG and VIS verifications we imposed two more assumptions on the environment of the switch fabric than in HOL. These assumptions are taken from the informal description of the switch fabric, and reduce the complexity of our automatic verification by a factor of 2. The HOL proof, however, is consequently more general.

Design Modification In all the approaches, the work needed to verify a modified design is greatly reduced once the original has been verified. MDG and HOL allow generic verification to be performed (e.g. word sizes are unspecified), though HOL is more flexible. No generic verification is possible in VIS. Because MDG and VIS are automated and fast, re-verification times are largely the time taken to modify the specifications and, for MDG, to find a new variable ordering. With HOL the behavioural specifications of many modules and the proof scripts themselves may need to be modified. For model checking in VIS, new environment machines, and model abstraction and reduction techniques may be required.

Tool Confidence An advantage of the HOL approach over the others is the confidence in the tool the LCF approach offers. Although the VIS and to a certain extent the MDG software packages have been successfully tested on several benchmarks and have been considerably improved, they cannot guarantee the same level of proof security as HOL. Compared to MDG, VIS is a more mature tool. It is implemented in a well-engineered fashion in C as compared to the prototype implementation of MDG in Prolog. Moreover, VIS is very widely used in both academia and industry, giving confidence in its correctness.

Error Detection All the approaches highlight errors, and help determine their location. However, the way this information manifests itself differs. VIS and MDG are more straightforward, outputting a trace of the input sequence that leads to the erroneous behaviour. Errors are detected automatically and can be diagnosed with the help of the counter-example facility. In addition, due to its front-end, VIS counter-examples can be analyzed using commercial tools such as XL-Verilog. In HOL, errors manifest themselves as unprovable goals. The form of the goal, the context of the proof and the verifier's understanding of the proof are combined to track down the location, and understand its cause.

Design Aid With the MDG (and to a certain extent the VIS) verification approach the verifier does not need to be concerned with the internal structure of the

Table 5: Summary of the Comparison

Area	**Feature**	**HOL**	**MDG**	**VIS**
Specification	Behavioral Specification - Time Taken		++	+
	Structural Specification - Time Taken	+	+	++
	Behavioral Specification - Expressibility	++	+	
	Structural Specification - Expressibility	++		+
	Executability of Specifications		+	++
Verification	Full Verification Completed	++	+	
	Machine Time Taken		+	+
	Total Verification Time Taken		++	+
	Verification Time for Modified Designs	+	+	+
	Safety Property Checking		+	++
	Liveness Property Checking		+	++
	Equivalence Checking		++	+
	Scalability	++		
	Confidence in Tool	++		+
Errors	Detect Errors	++	++	++
	Locating Errors	+	++	++
	Avoid Error Introduction	+	+	+
	False Error Reports		++	+
Design	Impart Understanding of Design	++	+	+
	Suggesting Design Improvements	++	+	+
	Commercial Front End			++
	Integration into Design Cycle		+	++

design being verified. This means that no understanding of the internals is obtained by doing the verification. In contrast, with HOL, a very detailed understanding of the internal structure is needed. The verifier must know why the design works the way it does. The process of doing the verification helps the verifier achieve this understanding. This means that internal idiosyncrasies in the implementation are likely to be spotted, as are other potential improvements.

A summary of the main comparison points is given in Table 5. Each system is given a rough rating of either "++", "+" or nothing to indicate how favorably the system comes out with respect to that feature. In conclusion, the major advantages of HOL are: the expressibility of the specification language; the confidence afforded in its results; the potential for scalability and the insight into the design that is obtained. The strength of MDG and VIS is in their speed; their relative ease of use and their error detection capabilities. MDG has the advantage of using abstract data types and uninterpreted functions with a rewriting facility, hence allowing larger circuits to be verified—but with the drawback that an MDG verification may not terminate in some cases. VIS is a very efficient model checker supporting the CTL expressiveness for both liveness and safety properties. Moreover, VIS outperforms MDG due to its maturity in the use of efficient graph manipulation techniques. The

VIS Verilog front-end and mature C implementation make VIS very attractive to industry.

Acknowledgments

We are grateful for the advice of X. Song and E. Cerny at the University of Montreal, I. Leslie and M. Gordon at the University of Cambridge, Z. Zhou at Texas Instruments, M. Langevin at Nortel, R. Brayton at Berkeley, F. Somenzi at Colorado, and H. Thimbleby, H. Goodman and R. Butterworth at Middlesex University.

References

[1] C.M. Angelo, D. Verkest, L. Claesen and H. de Man. On the Comparison of HOL and Boyer-Moore for Formal Hardware Verification. *Formal Methods in System Design*, 2:45–72, Kluwer, 1993.

[2] R. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[3] R. Brayton et al. VIS: A System for Verification and Synthesis In R. Alur and T. Henzinger, eds, *Computer Aided Verification*, LNCS 1102, 428–432, Springer-Verlag, 1996.

[4] B. Chen, M. Yamazaki and M. Fujita. Bug Identification of a Real Chip Design by Symbolic Model Checking. In *Proceedings of the International Conference on Circuits and Systems*, 132–136, 1994.

[5] F. Corella, Z. Zhou, X. Song, M. Langevin, and E. Cerny. Multiway Decision Graphs for Automated Hardware Verification. *Formal Methods in System Design*, 10(1):7–46, 1997.

[6] P. Curzon. Tracking Design Changes with Formal Machine-checked Proof. *The Computer Journal*, 38(2):91–100, July 1995.

[7] P. Curzon and I.M. Leslie. A Case Study on Design for Provability. In *Proceedings of the International Conference on Engineering of Complex Computer Systems*, pages 59–62, IEEE Computer Society Press, 1995.

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

[9] K. Edgcombe. *The Qudos Quick Chip User Guide*. Qudos Limited.

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

[11] M.J.C. Gordon, A.J. Milner and C.P. Wadsworth. Edinburgh LCF: A Mechanical Logic of Computation. Volume 78 of LNCS, Springer Verlag, 1979.

[12] M.J.C. Gordon. HOL: A Proof Generating System for Higher-order Logic. In G. Birtwistle and P.A. Subrahmanyam, eds, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer Academic Publishers, 1988.

[13] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-order Logic.* Cambridge University Press, 1993.

[14] F.K. Hanna and N. Daeche. Specification and Verification of Digital Systems using Higher-order predicate logic. *IEE Proceedings*, 133, E-5, 242-254, September 1986.

[15] J.M.J. Herbert. Case Study of the Cambridge Fast Ring ECL Chip using HOL. Technical Report 123, University of Cambridge, Computer Laboratory, February 1988.

[16] S. Coupet-Grimal and L. Jakubiec. Hardware Verification using Co-induction in Coq. In *Proceedings of the International Conference on Theorem Proving in Higher-Order Logics*, Nice, France, September 1999.

[17] T. Kropf *Formal Hardware Verification: Methods and Systems in Comparison.* LNCS 1287, State-of-the-Art Survey, Springer Verlag, 1997.

[18] I.M. Leslie and D.R. McAuley. Fairisle: An ATM Network for the Local Area. *ACM Communication Review*, 19(4):327–336, 1991.

[19] D.E. Long. Model Checking, Abstraction and Compositional Verification. *Ph.D thesis*, Carnegie Mellon University, July 1993.

[20] J. Lu and S. Tahar Practical Approaches to the Automatic Verification of an ATM Switch Fabric using VIS. In *Proceedings of the IEEE Great Lakes Symposium on VLSI*, 368–373, 1998.

[21] J. Lu, S. Tahar, D. Voicu, and X. Song. Model Checking of a real ATM Switch. In *Proceedings of the IEEE International Conference on Computer Design*, 195–198, Austin, Texas, October 1998,

[22] K. L. McMillan. *Symbolic Model Checking.* Kluwer, 1993.

[23] S. Owre, N. Shankar, and J. Rushby. PVS: A Prototype Verification System. In *Proceedings of CADE 11*, Saratoga Springs, New York, June 1992.

[24] M. Palanisamy and S. Tahar. Formal Verification of an RCMP Routing Logic. In *Proceedings of the 11th International Conference on Microelectronics*, Kuwait City, Kuwait, November 1999.

[25] S. Rajan, M. Fujita, K. Yuan, and M. Lee. High-Level Design and Validation of ATM Switch. In *Proceedings of the IEEE International High Level Design Validation and Test Workshop*, Oakland, California, November 1997.

[26] K. Schneider and T. Kropf. Verifying Hardware Correctness by Combining Theorem Proving and Model Checking. In J. Alves-Foss, editor, *Higher Order Logic Theorem Proving and Its Applications: Short Presentations*, 89–104, 1995.

[27] S. Tahar, X. Song, E. Cerny, Z. Zhou, M. Langevin, and O. Ait-Mohamed. Modeling and Verification of the Fairisle ATM Switch Fabric using MDGs. *IEEE Transactions on CAD*, 18(7):956–972, July 1999.

[28] D. Voicu, E. Cerny, and X. Song. Formal Verification of an ATM Switch Port Controller. In *Proceedings of the 5th Int. Conf. on Electronic Devices and Systems*, Czech Rep., June 1998.

[29] Y. Xu, E. Cerny, X. Song, F. Corella and O. Ait-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, 219–231, 1998.

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