Nordic Journal of Computing

COMPARING HOL AND MDG: A CASE STUDY ON THE VERIFICATION OF AN ATM SWITCH FABRIC

SOFIÈNE TAHAR[§] AND PAUL CURZON[‡]

[§]Concordia University, Department of Electrical and Computer Engineering
1455 de Maisonneuve West, Montreal, Quebec, H3G 1M8 Canada
tahar@ece.concordia.ca

[‡]Middlesex University, School of Computing Science
Bounds Green Road, London N11 2NQ, UK
P.Curzon@mdx.ac.uk

Abstract. Interactive formal proof and automated verification based on decision graphs are two contrasting formal hardware verification techniques. In this paper, we compare these two approaches. In particular, we consider HOL and MDG. The former is an interactive theorem-proving system based on higher-order logic, while the latter is an automatic system based on Multiway Decision Graphs. As the basis for our comparison we have used both systems to independently verify a fabricated ATM communications chip, the Fairisle 4 by 4 switch fabric.

1. Introduction

Formal hardware verification techniques are attracting widespread interest due to their potential to give very strong results about the correctness of designs. Two very different forms of formal verification have arisen: interactive proof and automated decision graph techniques. The aim of this paper is to compare and contrast these two approaches using the hardware implementation of an Asynchronous Transfer Mode (ATM) switch fabric as a case study. Such a comparison is of use to the research community to help improve both technologies and to researchers concerned with combining the systems to get the best features of both. It is also of use to industrial users considering adopting a hardware verification technology.

We have based this comparison study on a significant, real hardware design. A similar approach was also taken by Angelo *et al.* [1] when comparing two proof systems: HOL and the Boyer–Moore theorem prover. An alternative approach is to look at a wide range of small examples [18]. Such an approach helps ensure that conclusions apply to more than just the example considered. 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.

Received October 1997. Revised November 1998.

In this paper we compare interactive proof and automated decision graph verification. In the interactive proof approach, the circuit and its behavioral specification are represented in the underlying logic of a general-purpose theorem prover. The user interactively constructs a formal proof which proves a theorem stating the correctness of the circuit. Many different proof systems with various forms of interaction have been used for this purpose. In this paper we consider one such system: HOL [14]. It is an LCF style [12] proof system based on higher-order logic.

In the automated decision graph approach the circuit is represented as a decision diagram. Techniques such as reachability analysis are used to automatically verify given properties of the circuit or verify machine equivalence. We consider the MDG system. It uses a new class of decision graphs called Multiway Decision Graphs [6]. They subsume the class of Bryant's Reduced Ordered Binary Decision Diagrams (BDDs) [4] while accommodating abstract sorts and uninterpreted function symbols.

As the basis of our comparison, we have used both HOL and MDG to independently verify the Fairisle [20] 4 by 4 switch fabric. This is a fabricated chip which forms the heart of an ATM communication switch. The device has been used for real applications in the Cambridge Fairisle network, designed at the Computer Laboratory of the University of Cambridge. It does the actual switching of 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 fabricated and in use, carrying real user data, prior to any formal verification attempt.

The outline of the paper is as follows. Section 2 describes related work concerned with the verification of network hardware. In Section 3 we give a brief overview of the particular hardware considered: the Fairisle 4 by 4 switch fabric. We describe its verification using HOL in Section 4 and using MDG in Section 5. For each, we overview the verification method, tools and our experiences on this case study. Finally, in Section 6 we draw conclusions.

2. Related Work

There has been a vast amount of work on formal hardware verification. There exists little, however, that is directly related to our study on verifying network hardware components.

J. Herbert [16] 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.

B. Chen *et. al* at Fujitsu Digital Technology Ltd. [5] 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 behavior under certain circumstances. Using the SMV tool [22], the authors identified the design error by checking some properties expressed in

Computation 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.

K. Schneider *et al.* [23] formally verified the Fairisle 4 by 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 behavioral specifications using so called hardware formulas [23]. These specifications are much lower level than our specifications. The verification of lower-level hardware modules which implement the top-level block units was automated. However, the complete verification of the intended overall behavior of the switch fabric against its implementation was not accomplished. This was done in our HOL verification presented here, though.

Other groups have also used the 4 by 4 fabric as a case study. Jakubiec and Coupet-Grimal are using it in their work using the Coq proof system for hardware verification [17]. Garcez [11] has also verified some properties of the 4 by 4 fabric using the HSIS model checking tool [2]. More recently, Lu [21] used the VIS tool [3] to perform property checking on various abstracted models of the fabric. In addition, he conducted equivalence checking between behavioral and structural specifications of submodules of the fabric written in Verilog. He also re-implemented the whole fabric using the Synopsys synthesis tool and used the Verilog-XL simulator of Cadence to graphically simulate all generated modules.

3. The Fairisle 4 by 4 Switch Fabric

The Fairisle switch forms the heart of the Fairisle network. It consists of three types of components: input port controllers, output port controllers and a switch fabric (Figure 1). Each port controller is connected to a transmission line and to the switch fabric. The port controllers synchronize incoming and outgoing data cells, appending control information to the front of the cells in a routing byte (Figure 2). In this paper, we are concerned with the verification of the switch fabric which is the core of the Fairisle ATM switch.

The routing byte (header) is stripped off before the cell reaches the output stage of the fabric. 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. 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, then only one will succeed. The others must retry later. The routing byte also includes a priority information bit (*priority*) that is used by the fabric during arbitration.

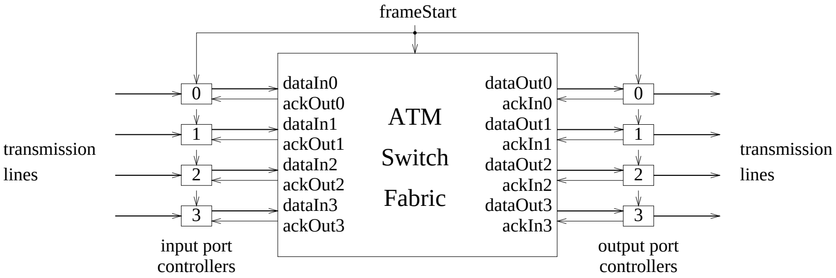

Fig. 1: The Fairisle ATM Switch

Arbitration takes place in two stages. First, high priority cells are given precedence, and for the remaining cells the choice is made on a round-robin basis. The input controllers are informed of whether their cell was successful using acknowledgment lines. The fabric sends a negative acknowledgment to the unsuccessful input ports, and passes the acknowledgment from the requested output port to the successful input port.

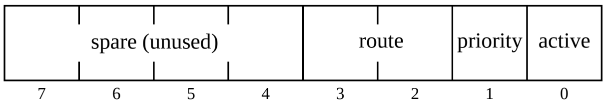

Fig. 2: The Routing Byte (Header) of the Fairisle ATM Cell

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 (Figure 1). It ensures that the port controllers inject data cells into the fabric synchronously so that the routing bytes arrive at the same time.

The behavior 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. The cells from all the input ports start when the *active* bit of any one of their routing bytes goes high. The fabric does not know when this will happen. However, all the input port controllers must start sending cells at the same time within the frame. If no input port raises the active bit throughout the frame then the frame is inactive—no cells are processed. Otherwise it is active.

As shown in Figure 1, the inputs to the fabric consist of the cell data lines, the acknowledgments that pass in the reverse direction, and the frame start signal, *frame start*, which is the only external control signal. The outputs consist of the switched data, and the switched and modified acknowledgment signals. Figure 3 shows a block diagram of the 4 by 4 switch fabric.

It is composed of an arbitration unit (timing, decoder, priority filter and arbiters), an acknowledgment unit and a dataswitch unit. The timing block controls the timing of the decision with respect to the frame start signal and the time the routing byte arrives. 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.

The resulting request situation for each output port are then passed 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. Using the output disable signals, the arbiters indicate to the other units when a new arbitration decision has been made. The dataswitch unit performs the actual switching of data from input port to output port according to the latest arbitration decision. The acknowledgment unit passes appropriate acknowledgment signals to the input ports. Negative acknowledgments are sent until an arbitration decision is made.

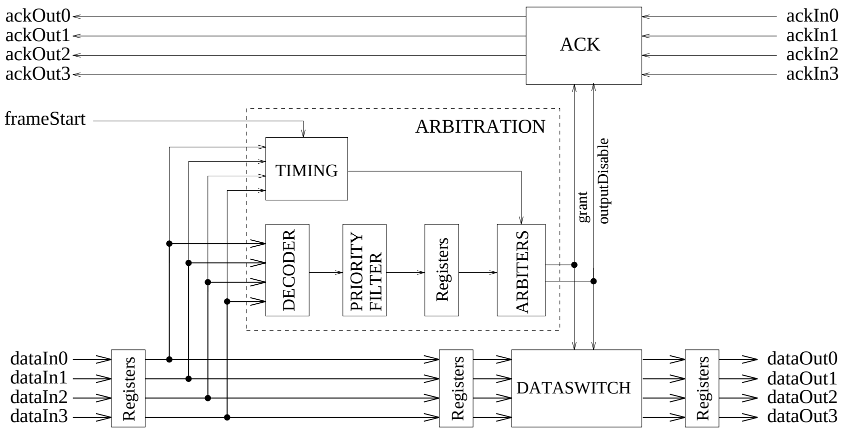

Fig. 3: The Fairisle ATM Switch Fabric

Each of these units is repeatedly subdivided down to the logic gate level, providing a hierarchy of modules. The design has a total of 441 basic components (a multiple input logic gate or single bit flip flop). It is built on a 4200 gate equivalent Xilinx programmable gate array. The switching element can be clocked at 20 MHz and currently frame start pulses occur every 64 clock cycles. The designers originally described the hardware using the Qudos hardware description language (HDL) [10] which was used for generating the Xilinx netlist. The Qudos simulator was used to perform the original (non-formal) validation.

The Fairisle Switch fabric 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 oversimplified and thus miss problems that would arise for a real design. Furthermore, it combines both control hardware with a datapath. This combination causes problems for traditional BDD-based verification systems. Control information also occurs in the data (the routing byte) preventing verification of a version with a reduced-width datapath from being possible. While not being a trivial design, it is simple enough for a verification to be performed in a reasonable amount of time.

4. The HOL Verification of the Fabric

In the first study, the Fabric was verified using the HOL Theorem-Proving System. This is an interactive proof system. 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.

4.1 The HOL Theorem-Proving System

The HOL90 theorem proving system is an LCF-style [12] theorem prover for higher-order logic [14]. 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 hardware description language (HDL), called Qudos HDL, 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.

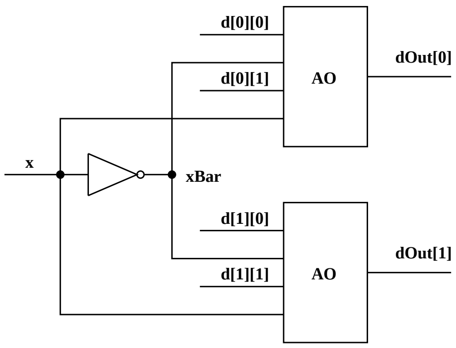

Fig. 4: The DMUX4T2 multiplexer circuit

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 were developed by manually translating the original description into higher-order logic. It would be straightforward to automate this translation process. However, no front-end translator from Qudos HDL to HOL existed and it was beyond the scope of this study to implement one. Clearly this would be desirable if a large number of designs written in Qudos HDL were to be verified.

Hardware components are modeled in HOL using relations on the inputs and outputs [15]. 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)
```

On a small example as above, there is little if any improvement in readability. The advantages are more pronounced in situations where the amount of duplication is greater. The duplication operator reduces the need for the same piece of code to be written out repeatedly, reducing the amount of code and the potential for mistakes. A major advantage is the opportunity it gives for generic specifications where the number of copies is given as an argument to the specification.

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 have been done was 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 Behavioral Specifications

The behavioral specification against which the structural specification was verified describes the actual, unsimplified behavior of the switch fabric. It is presented at a similar level of abstraction to that used informally by the designers. It describes the switch behavior over a frame in terms of timing behavior represented using interval operators. Such a description is essentially a formalization of a timing diagram such as in Figure 5. 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 behavior 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 behavior in all circumstances must always be specified. This means the amount of specification work is less in this respect in HOL.

As an example of a behavioral 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 behavior is specified by a function argument to the interval operator, `DURING`. This specification is illustrated diagrammatically in Figure 5.

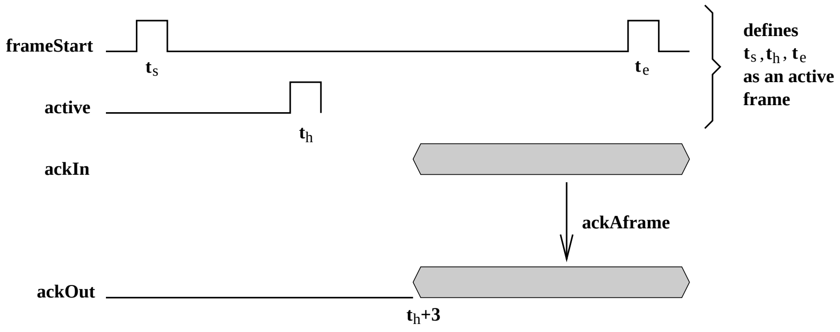

Fig. 5: The Timing Diagram for the Acknowledgment Output

(AFRAME t_s t_h t_e *frameStart* *active*) $\supset$
  STABLE $(t_s + 1)$ $(t_h + 3)$ *ackOut* (ZEROW ...) $\wedge$
  DURING $(t_h + 3)$ $(t_e + 1)$ *ackOut*
    (λ t. AckAframe (d t_h) (*last* $(t_h + 2)$) (*ackIn* t))

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. λ is a lambda operator introducing an un-named function. In the above it is used to give a function that takes a time t as an argument. Thus DURING provides a series of times (corresponding to the interval). These are passed to the un-named function which returns the value for that time.

Other clauses in the specification describe the behavior over an inactive frame in which no cells arrive. A similar set of clauses are given for the behavior 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 from the state machine based specification style used in MDG (described later).

In the HOL verification, it is not sufficient to simply provide a behavioral specification for the whole design. Each module is verified independently, as described in the next section. This means we must provide behavioral 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:

DMUX4T2_SPEC ((d, x), *dOut*) =
  $\forall$ t. *dOut* t = Mux (x t) (d t)

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.

```
Mux x d = BITS (BV x) d
```

4.4 The Verification Process

The verification of the 4 by 4 switch fabric used standard techniques [13]. 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. It gave a natural sub-division of the proof. Each module could be verified separately and independently of the others. Both the structural and behavioral 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 [13, 15]:

$$\vdash \; \textit{assumptions on environment} \supset (\textit{structure} \supset \textit{behavior})$$

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

The internal state, which is an explicit argument to the behavioral 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:

```
∀ ackIn ackOut dOut d frameStart.
  ENVIRONMENT frameStart d ⊃
   FABRIC4B4 ((d,frameStart,ackIn), (dOut,ackOut)) ⊃
    ∃ last.
     FABRIC4B4_SPEC last ((d,frameStart,ackIn), (dOut,ackOut))
```

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 the environment corresponds to one particular instance of the general environment condition used here. Note that the MDG approach can be advantageous for a large scale verification as in some circumstances, only the behavior of the switch in its working environment is of practical interest.

A correctness theorem of the above form was proved for each module stating that its implementation (in terms of logic gates) satisfied its specification. This was proved in several independent steps.

(1) All the module's submodules were verified (by applying this verification approach recursively). The correctness theorems obtained state that the structural specification (implementation) of each submodule implies its behavioral specification.

(2) The module was verified under the assumption that all its submodules correctly implemented their behavioral specifications. In essence the submodules were treated as black-boxes for the purposes of this step—their behavioral specifications were used in the proof rather than their implementations. A version of the structural specification that refers to the behavioral specifications of the submodules rather than their implementations was used.

(3) The new descriptions of the implementation of the module used in the previous step were replaced by the original descriptions that refer to the implementations of the submodules. This was done by appealing to the correctness statements for the submodules. This gave the desired theorem stating the correctness of the module.

Verifying the full design involved doing the above for the top level module. The bottom level of the hierarchy consisted of logic gates and single-bit registers. These were only specified behaviorally: they were 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 behavioral specification of the module. The proof for each interval was essentially inductive. A lemma was proved that the implementation satisfied the behavior at the start of the interval. It was also proved that, within the interval, if the behavior 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 behavior gives much greater insight. In addition to uncovering errors, this can serve to highlight anomalies in the design and suggest improvements, simplifications or alternatives [8].

4.5 Time Taken

The module specifications (both behavioral 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 behavioral 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 behavioral specification of all the modules. The behavioral 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 module's 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 behavior of several outputs; and those behaviors 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.

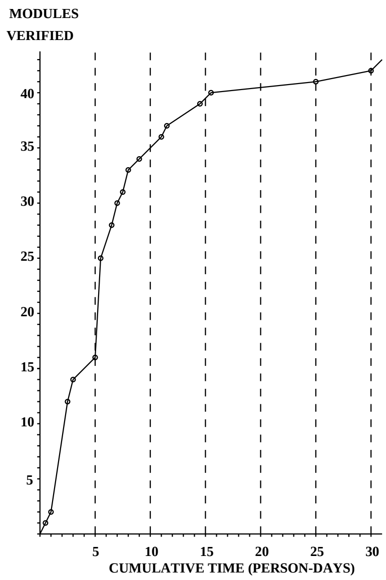

Fig. 6: Time taken to verify the fabric modules using HOL

It takes several hours of machine time on a Sparc 10 to completely rebuild the 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 an existing design is to be adapted for new purposes, 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 by 16 fabric. Although the 4 by 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. Generic proofs were not used to as great an extent as was possible in this study. This was because it was generally easier to reason about specific values than general ones.

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 HOL structural specifications (and could just as easily have been in the implementation) were discovered. The original versions of the HOL behavioral 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 behavioral 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.

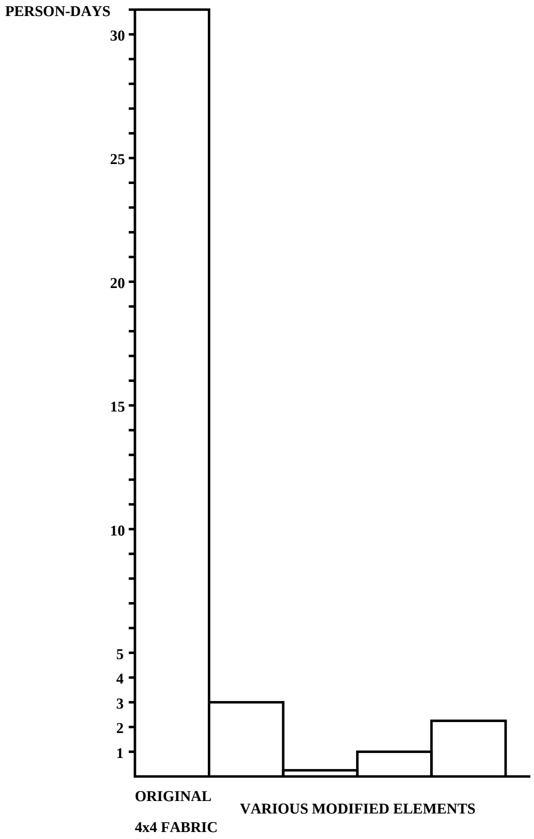

Fig. 7: Time taken to verify variations on the fabric design using HOL [7]

It was also clear which signals were involved. It was not immediately clear which specification (structural or behavioral) 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. An example where information is lost is where an assumption, $t_1 < t_2$, is converted to $t_1 \leq t_2$ during the proof (perhaps to resolve the goal with a lemma containing the latter as an assumption). Here 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 [17].

4.7 Scalability

The HOL proof approach has the potential to be scalable to large designs in a way that the MDG approach is not. Because the HOL approach is modular and hierarchical, increasing the size of the design does not necessarily increase the complexity of the proof. Only new modules need to be verified—the new proof is built on top of the original. 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 [9].

5. The MDG Verification of the Fabric

In the second study, the same circuit was verified using a decision graph approach. In our work, a new class of decision diagrams called *multiway decision graphs* (MDGs) was used to represent sets of states as well as the transition and output relations [6]. Based on a new technique called *abstract implicit enumeration*, hardware verification tools have been developed which perform combinational circuit verification, safety property checking and equivalence checking of two sequential machines [6].

5.1 The MDG Verification System

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 labeled by formulas, the internal nodes are labeled by terms, and the edges issuing from an internal node are labeled 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 [4]. Like ROBDDs, the MDGs require a fixed node ordering. Currently, the node ordering 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.

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. We call the model 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.

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. These include 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 [6]). 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. Rewrite rules may need to be provided to partially interpret the otherwise uninterpreted function symbols.

Based on the above operators and algorithms, a set of verification applications have been developed including:

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

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

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

Counter-example generation: When the invariant is not satisfied during safety property checking or sequential verification, a counter-example will be provided to help the user identify the errors. 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 [25]. 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 [6]. 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

The actual hardware implementation of the switch fabric was described at two levels of abstraction, namely 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, the Qudos HDL gate-level description was translated 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. 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:

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

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.

The final goal of the MDG verification approach is to verify the hardware implementation against a high-level behavioral specification described in terms of an abstract state machine. This cannot be done using the gate-level model as such a direct verification may rapidly lead to a state explosion. Hence, besides the gate-level description, a more abstract (RTL) description of the implementation was also provided and 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 behavior 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 behavior 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 [24].

5.3 The Behavioral Specifications

MDG-HDL is also used for behavioral descriptions. A behavioral 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, the behavioral specification of the switch fabric was given in two different forms: (1) as a complete high-level behavioral state machine and (2) as a set of properties which reflect the essential behavior of the switch fabric as it is used in its environment. This latter form was mainly introduced to validate the specifications in early stages of the project.

5.3.1 ASM Behavioral Specification

Starting from timing-diagrams describing the expected behavior of the switch fabric, a complete high-level behavioral specification was derived 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. A schematic representation of the ASM specification of the 4 by 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 behavior of the switch fabric. States 2, 3, 4 and 5 along the time axis t_s describe the behavior of the switch on the arrival of a frame start signal. States 6 to 13 along the

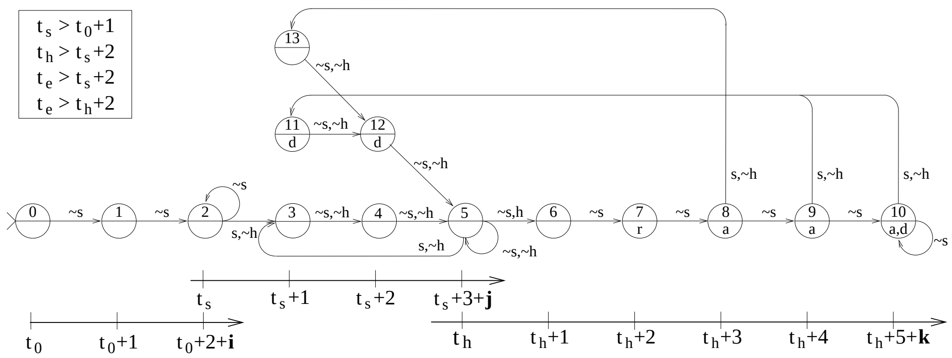

Fig. 8: ASM Behavioral Specification

time axis t_h describe the behavior of the switch fabric after the arrival of the headers. The 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 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. Thus, $\sim s$ and $\sim h$ in Figure 8 mean the absence of a frame start signal and the absence of a header, respectively. 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, some basic sorts, constants and functions (cross-operators) were first introduced, 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 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 $ackOut_i$, $i \in \{0, .., 3\}$:

$$
\begin{array}{l}
\textbf{if } ((co_0 = 1) \textbf{ and } (ip_0 = i)) \textbf{ then } (ackOut_i = ackIn_0) \\
\textbf{ef } ((co_1 = 1) \textbf{ and } (ip_1 = i)) \textbf{ then } (ackOut_i = ackIn_1) \\
\textbf{ef } ((co_2 = 1) \textbf{ and } (ip_2 = i)) \textbf{ then } (ackOut_i = ackIn_2) \\
\textbf{ef } ((co_3 = 1) \textbf{ and } (ip_3 = i)) \textbf{ then } (ackOut_i = ackIn_3) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else } (ackOut_i = 0)
\end{array}
$$

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 behavior of the switch fabric, a set of properties were also provided. They reflect the essential behavior 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 implementation. If we consider the behavior of the fabric when operating in the intended real Fairisle switch environment, its cyclic behavior can be simulated as an *environment state machine* having 68 states as shown in Figure 9. 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 is mapped to states 3 or 66; t_h to state 12; and t_e to state 66. Thus a time point t ranging, for example, between t_s+1 and t_h+3 is expressed as the states ranging between states 4 (3+1) and 15 (12+3) of the environment machine.

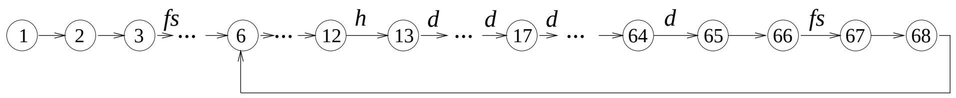

Fig. 9: The Environment State Machine of the Fairisle ATM

Based on this environment state machine, the properties were described as invariants which should hold in all reachable states of the specification model. In the 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]. Using the mapping of t_h+5 and t_e+2 to the respective states 17 (12+5) and 68 (66+2), 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 [24].

5.4 The Verification Process

In the MDG approach, the original gate-level implementation of the switch fabric was first verified against an RTL implementation. The RTL implementation was then verified against a behavioral specification given as an abstract state machine (ASM). Thus obtaining a complete verification from high-level behavior down to the gate level. In an early stage of the project, some specific properties that reflect the behavior of the fabric in its real operating environment as described above were also verified.

5.4.1 Equivalence Checking

The behavioral equivalence 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 (cross-operators) which encode and decode abstract data to boolean data and vice-versa [24]. 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 $concat8$ 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 $wordn$. Using the sequential equivalence checking facility of the MDG tools, the abstract machine was verified to be 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 dealt mainly with the equivalence of the dataswitch blocks at the two levels.

Based on implicit reachability analysis, the equivalence of the behavioral ASM specification against the RTL hardware model was checked when both are seen as abstract state machines. This ensures that the two machines produce the same observable behavior 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 [19]. For this product machine, an MDG representing a set of states encodes a relation between 39 concrete and 36 abstract state variables [19]. The relation may depend on data values, encoded using cross-terms, however. Cross-terms are those terms resulting from the application of a cross-operator to an abstract variable, e.g. $act(d)$, where act is a cross-operator of type $[wordn \rightarrow bool]$ and d is an abstract variable of sort $wordn$. 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.

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

$RTL_{(8)}$ 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 behavior $ASM_{(8)}$ which is an 8-bit instance of the behavior $ASM_{(n)}$. From the n-bit generic result in (2), we hence deduce through instantiation:

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

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

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

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 I, including the CPU time, memory usage and the number of MDG nodes generated.

5.4.2 Property Checking

Prior to the full verification, both behavioral and RTL structural specifications were also checked 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 (delayed) input and fabric output signals. Combining these machines, we obtain the required platform for checking if the invariant properties hold in all reachable states of the specification. Experimental results for the verification of four example properties are shown in Table I (where the previously described property P is labeled $P3$). Although the properties verified do not represent the complete behavior 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 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 behavior and writing the behavioral 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 behavioral 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 behavioral 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 I 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 I: Experimental Results for the MDG Verification

A disadvantage of MDGs is that much verification time is spent finding an optimal variable ordering. This is crucial since a bad ordering easily leads to a state space explosion. This occurred after an early ordering attempt. For more information about the variable ordering problem, which is common to all ROBDD-based systems, see [4].

Because the verification is essentially automatic, the amount of work rerunning 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 behavioral 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 behavior 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, i.e. 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 environment state machine (which is a simple machine as 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 behavioral 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 behavioral specification, are reported in Table I.

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 behavioral 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. In order to handle large designs, major efforts are in general required to set up the appropriate model abstraction levels.

6. Conclusions

The MDG and HOL structural descriptions are very similar, both to each other and to the original designer's description. 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 only minimal additional effort, however.

The behavioral descriptions of the two approaches are totally different. The MDG specification is based on a state machine model while the HOL one is based on interval operators. The latter explicitly describes the timing behavior in terms of frames which correspond to whole ATM cells arriving. This contrasts with the MDG specifications where the frame abstraction is not used: the description is firmly at the byte level. Both describe the behavior in a clear and comprehensive form. Which of these is preferred is perhaps a matter of taste.

An advantage of MDG 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. This is useful for verifying that a specification satisfies its requirements. It can greatly reduce the full verification cost by catching errors at an early stage.

Writing the behavioral specifications took longer in HOL, as separate specifications were needed for each module. In MDG this was not necessary because the whole design was verified in one go, rather than a module at a time. This also reduced the MDG verification time because fewer mistakes

Feature	MDG	HOL
Specification		
Behavioral Specification – Time Taken	+	
Structural Specification – Time Taken		+
Behavioral Specification – Expressibility	+	++
Structural Specification – Expressibility		++
Verification		
Machine Time Taken	++	
Total Verification Time Taken	+	
Verification Time for Design Modifications	+	+
Property Checking	++	+
Scalability		++
Confidence in Tool		++
Error Detection		
Detect Errors	++	++
Locating Errors	++	+
Avoid Error Introduction	+	
False Error Reports	++	
Design Aid		
Impart Understanding of Design	+	++
Suggesting Design Improvements	+	++

TABLE II: Summary of the Comparison

were made. Moreover, note that as the HOL project preceeded the MDG work, the understanding of the switch behavior achieved during the former helped to speed up writing the MDG behavioral specification to some extent.

The HOL verification was much slower, taking a matter of months. This time includes the verification of each of the modules and the verification of their combination. Using HOL, a large number of lemmas had to be proved and much effort was required to interactively create the proof scripts. For example, the time spent for the verification of the dataswitch unit was about 3 days. Here the proof script was about 530 lines long (17 KB). The MDG verification was achieved automatically without the need of a proof script. All that was required was the careful management of the MDG node ordering (as with ROBDDs). However, this is a matter of hours or at most a few days of work.

In both the HOL and MDG approaches, the amount of work necessary to verify a modified design, once the original has been verified, is greatly reduced. Both allow generic verification to be performed, though HOL has the potential to be more flexible. Because MDG is automated and fast, the re-verification times would largely be just the time taken to modify the specifications and to find a new variable ordering. In the HOL approach, the behavioral specifications of many modules and the proof scripts themselves may need to be modified.

An advantage of the HOL approach in contrast to the MDG method is the confidence in the tool the LCF approach offers. Although the MDG software package has been successfully tested on several benchmarks and has been considerably improved, it is not yet a mature tool. It cannot guarantee the same level of proof security as HOL. The main advantage of the MDG approach is that it is much quicker and is automatic. On the other hand the theorem proving approach is potentially scalable and involves a comprehensive investigation of why the design works correctly. However, these advantages are only likely to be realized in practice if the level of proofs which must be provided to the system can be raised closer to the level of informal proofs.

Both approaches successfully highlight errors, and help determine their location. However, the way this information manifests itself differs. MDG is more straightforward, outputting a trace of the input sequence that leads to the erroneous behavior. 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.

The MDG verification approach is a black-box approach—the verifier does not need be concerned with the internal structure of the design being verified. This means that no understanding of the internals is obtained by doing the verification. In contrast, HOL is a white-box approach. To complete the verification, 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 idiosyncracies in the implementation are likely to be spotted, as are other potential improvements. This occurred in the HOL verification of the 4 by 4 fabric and of the variations of it used for the 16 by 16 fabric [9].

A summary of the main comparison points is given in Table II, split into the major areas of specification (both effort involved in creating and general readability/understandability); verification; error detection and correction; and the extent to which a verification in the two systems has potential to help improve designs beyond just finding errors. MDG and HOL are each given a rating of either a double-plus, single-plus or nothing to indicate the degree to which the system comes out favorably with respect to that feature. These ratings are clearly fairly coarse.

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 is in its speed, its relative ease of use and its error-detection capabilities.

Acknowledgments

We are grateful to Xiaoyu Song and Eduard Cerny at the University of Montreal, Canada, Zijian Zhou at Texas Instruments, USA, and Michel Langevin at Nortel, Canada for initiating and advocating this study. Ian Leslie and Mike Gordon at Cambridge University were also of great help. This work was partially funded by NSERC research grant OGP0194302, and EPSRC research agreements GR/J11133 and GR/K10294 while the second author was at the University of Cambridge.

References

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

[2] Aziz, A., *et al.* 1994. HSIS: A BDD based Environment for Formal Verification. In *Proceedings of Design Automation Conference*, New York, USA, 454–459.

[3] Brayton R. K. *et al.* 1996. VIS: A system for Verification and Synthesis. In *Computer Aided Verification*, Alur, R. and Henzinger, T., Editors, Volume 1102 of Lecture Notes in Computer Science, Springer-Verlag, 428–432.

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

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

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

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

[8] Curzon, P., and Leslie, I. M. 1995. A Case Study on Design for Provability. In *Proceedings of International Conference on Engineering of Complex Computer Systems*, Fort Lauderdale, Florida, USA, 59–62.

[9] Curzon, P., and Leslie, I. M. 1996. Improving Hardware Designs whilst Simplifying their Proof. In *Designing Correct Circuits*, Sheeran, M. and Satnam Singh, S., Editors, Workshops in Computing, Springer-Verlag.

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

[11] Garcez, E. H. A. 1995. The Verification of an ATM Switching Fabric using the HSIS Tool. Technical Report WSI-95-13, University of Tuebingen, Dept. of Computer Science, Tuebingen, Germany.

[12] Gordon, M. J. C., Milner, A. J., and Wadsworth, C. P. 1979. *Edinburgh LCF: A Mechanical Logic of Computation*. Volume 78 of Lecture Notes in Computer Science, Springer Verlag.

[13] Gordon, M. J. C. 1987. HOL: A Proof Generating System for Higher-order Logic. In *VLSI Specification, Verification and Synthesis*, Birtwistle, G. and Subrahmanyam, P. A., Editors, Kluwer Academic Publishers, 73–128.

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

[15] Hanna, F. K., and Daeche, N. 1986. Specification and Verification of Digital Systems using Higher-order Predicate Logic. *IEE Proceedings*, 133, E-5 (Sept.), 242–254.

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

[17] Jakubiec, L., Coupet-Grimal, S., and Curzon, P. 1997. A Comparison of the Coq and HOL Proof Systems for Specifying Hardware. In *International Conference on Theorem Proving in Higher Order Logics: B-Track*, Gunter, E. and Felty, A., Editors, 63–78.

[18] Kropf T. 1997. *Formal Hardware Verification: Methods and Systems in Comparison*. Volume 1287 of Lecture Notes in Computer Science, State-of-the-Art Survey, Springer Verlag.

[19] Langevin, M., Tahar, S., Zhou, Z., Song, X., and Cerny, E. 1996. Behavioral Verification of an ATM Switch Fabric using Implicit Abstract State Enumeration. In *Proceedings of International Conference on Computer Design*, Austin, Texas, USA, 20–26.

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

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

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

[23] Schneider, K., and Kropf, T. 1995. Verifying Hardware Correctness by Combining Theorem Proving and Model Checking. In *International Workshop on Higher Order Logic Theorem Proving and Its Applications: B-Track*, Alves-Foss, J., Editor, 89–104.

[24] Tahar, S., Zhou, Z., Song, X., Cerny, E., and Langevin, M. 1996. Formal Verification of an ATM Switch Fabric using Multiway Decision Graphs. In *Proceedings of Great Lakes Symposium on VLSI*, Ames, Iowa, USA, 106–111.

[25] Zhou, Z. 1996. Multiway Decision Graphs and Their Applications in Automatic Formal Verification of RTL Designs. PhD thesis, University of Montreal, IRO Department, Montreal, Canada.