The Formal Verification of the Fairisle ATM Switching Element: An Overview

Paul Curzon

Technical Report Number 328

March 1994

University of Cambridge
Computer Laboratory
New Museums Site
Pembroke Street
Cambridge
CB2 3QG
United Kingdom

The Formal Verification of the Fairisle ATM Switching Element: An Overview

Paul Curzon

University of Cambridge,
Computer Laboratory
Email: pc@cl.cam.ac.uk

Abstract

We overview the formal verification of an implementation of a self routeing ATM switching element. This verification was performed using the HOL90 theorem proving system so is fully machine-checked. The switching element is in use in a real network, switching real data. Thus, this work constitutes a realistic formal verification case study. We give an informal overview of the switch and element and give a tutorial on the methods used. We overview how these techniques were applied to verify the switching element. We then discuss the time spent on the verification. This was comparable to the time spent designing and testing the element. Finally we describe the errors discovered.

1 Introduction

Communication networks are rapidly becoming all pervasive. Systems are increasingly being networked in the local area with applications using non-local services. In the wide area telecommunications companies are turning to digital networks. As networks become all pervasive, the consequences of errors in the design or implementation of network components becomes increasingly important. This is especially so if networks are used in safety-critical applications where communication problems could cause loss of life. Errors could cause the network to deadlock, particular links to crash, the service to be degraded to an unacceptable level, or even the whole network to crash. Network problems can affect a wide range of users and applications and cause whole systems or companies to grind to a halt. Indeed such problems do frequently occur, as can be seen be scanning Peter Neumann's digest of computer related risks to the public [18]. For example a telephone network crash could cause loss of life because the emergency services can not be contacted. The recent problems of the London Ambulance Service computer aided dispatch system also highlight how problems with communications equipment can lead to loss of life. It is thus desirable that network components are error free. In reality zero-defect systems are probably unachievable; rather this is a goal to be striven for. Whilst the validation of designs is an important area, we are concerned here only with the validation of implementations. The validation of network components is at best difficult. Testing cannot hope to uncover all errors in an implementation because only a small fraction of the possible cases can be considered. Formal verification is a validation technique that can alleviate this problem. Mathematical methods are used to rigorously investigate all valid combinations of inputs.

Asynchronous Transfer Mode (ATM) is being hailed as the solution to many communication problems. In essence it consists of sending data over a packet-switched network using virtual circuits and short fixed size packets known as *cells*. It is a flexible technology and is being adopted by both the computer and telecommunication industries in local and wide area networks in response to changing communication demands. It is likely to be the most important transfer mode of the foreseeable future. However, it represents a large paradigm shift in communications and there is currently little experience from which to derive confidence of correct behaviour. An ATM network design is thus a timely application for verification research.

The work described here is the first step of a larger project to apply formal verification techniques to an ATM network. The design of a communication network is structured into layers that can be naturally exploited by hierarchical proof. We intend to perform a multi-level verification of an ATM network. This contrasts with previous work which has considered aspects of a network in isolation. The main result of this work will be an understanding of how formal verification techniques can and should be applied to the implementation of ATM communication networks.

In the tradition of the Computer Laboratory the investigation is based on a real network moving real user data: the Fairisle ATM network [16]. Fairisle is an experimental ATM network designed and built at the University of Cambridge Computer Laboratory. It is being used as the basis for research into management issues of ATM Networks in addition to applications such as multi-media. The network forms the experimental apparatus upon which real experiments take place. It is also used by real users transmitting real data. It thus provides a realistic case study for the investigation of the formal verification of ATM Networks. In this report, we describe the first results of this work: the formal machine-checked verification of the Fairisle 4 by 4 switching element.

The 4 by 4 switching element forms the heart of the Fairisle switch. It performs the actual switching of cells from input ports to output ports and arbitrates cell clashes. The switching element was designed and implemented prior to any formal verification or specification being carried out. The formal verification work has been performed on completed implementations. This is generally considered to be harder than if the formal specification and verification is integrated into the design process. This problem was exacerbated further since there was little informal documentation. The formal specifications were largely deduced by examining the implementation.

We have formalised both the implementation and its behaviour. We then used formal logic to rigorously prove that the behaviour suggested by the description of the implementation satisfies the specified behaviour. In contrast to validation using inexhaustive testing, the results hold for all valid sets of inputs, not just for some small subset. Formal verification corresponds in this sense to exhaustive testing. However, the latter is infeasible for all but very small designs due to the number of cases to consider. Formal verification is feasible because of the use of mathematics (for example using induction) to consider the results of many cases at once.

Whereas testing can normally be performed on the actual fabricated hardware, formal verification can only ever deal with descriptions of the hardware. It corresponds in this way to performing exhaustive testing using a simulator which interprets a hardware description language description of the implementation.

There are several tangible results of this work. We have produced a precise, unambiguous description of the behaviour of the switching element. This will be of use to the designers of the switch. We have produced a proof of a correctness theorem stating that a particular implementation does exhibit that behaviour. As a side effect, we also have similar results for all of the modules used in the implementation. These results can be reused if any modules are used in other designs. Finally, it should be possible to modify the proofs for other designs of the switching element very quickly.

The proofs described were carried out using the HOL90 theorem prover: a Standard ML implementation of the HOL system [7]. It not only provides mechanical assistance to the proof process by providing semi-automatic proof tools but also ensures that the correctness theorem obtained is a theorem. The system will only call something a theorem if it has been rigorously proved. The critical core of code that allows something to be called a theorem is relatively small. This reduces the chances of errors in the theorem prover giving misleading results.

The HOL theorem prover has been used previously to perform machine checked proofs of network components. Herbert formally verified an ECL chip: a local area network interface used as part of the Cambridge Fast Ring [12] [11]. Melham performed a proof of correctness of the T-ring: a very simple ring communication network that was designed as a formal verification case study [17]. Josephs *et al* produced a hand proof in a CSP-like algebraic formalism that switching elements similar to those proposed for the INMOS Transputer could be connected together in a regular way to implement a router of arbitrary size [14].

The outline of the remainder of this report is as follows. In Section 2, we overview the Fairisle network. The purpose of this section is to illustrate the environment in which the switching fabric operates. In Sections 3 and 4, we informally describe the intended behaviour and implementation of the switching element. In Section 5 we give a brief tutorial on formal hardware verification and overview the techniques we employed by considering very simple pieces of hardware. In Sections 6, 7 and 8, we outline the formal behavioural and structural specifications of the switching element and their verification. Complete details of the specifications used and the correctness theorems proved can be found in a companion report [5]. In Section 9 we discuss the time taken to perform the formal specification and verification work. This was comparable to the time originally spent designing, implementing and informally testing the fabric. Finally, in Section 10, we discuss the errors discovered. None were found in the fabricated implementation. This was unsurprising as the element had been in service for some time prior to the formal verification work commencing. Many errors were discovered in the specifications. This was also unsurprising since documentation of the design and its implementation was sparse.

2 The Fairisle Network

The Fairisle network consists of a series of switches, connected to each other. Each host is connected to a switch. Hosts communicate by sending fixed size packets known as *cells* across the network. The source passes a cell to its local switch. The switch then sends the cell on an outgoing link appropriate for the cells ultimate destination. In addition to switching cells from incoming links to outgoing links, the switch performs arbitration between cells that clash, forwarding the successful cells and queueing the unsuccessful ones for later transmission.

Fairisle is a *fast* cell switching network. That is, there is only minimal functionality in the network. For example, there is no error protection on a link-by-link basis. Error detection can be performed outside the network if required. This is possible due to the high quality of the network links. It has the advantage that the switches are relatively simple, fast and also flexible: only applications that require error detection need perform it.

Fairisle is a virtual circuit network. Before data can be sent over the network, a virtual connection must be set up between the source and destination. This is achieved using a signalling protocol. The signalling mechanism allows the source to indicate the destination with which it would like to communicate and also to indicate the bandwidth required. A control cell requesting a connection is sent by the source to its local switch. The switch determines if it has sufficient resources to fulfil the request currently available, and if so forwards the request to subsequent switches. If a route between source and destination with sufficient resources is found, then the resources are allocated to the connection. The source is given a virtual circuit identifier (VCI), to include in its data cells on that connection. Once a virtual circuit has been established, further cells may be sent over the circuit indefinitely with no further signalling overhead, provided the requested bandwidth is not exceeded.

Each switch on the successful path, allocates a local VCI to the circuit. It records the VCI that the next switch on the chosen route is using to identify the circuit in a table along with the associated outgoing link. When a cell arrives on the incoming link with a given VCI, the switch looks up the VCI in the table, and replaces the incoming VCI with that used by the next switch before forwarding the cell. This avoids problems of universal naming of virtual circuits and also means that the VCI's can be kept short. This has a dual advantage. Cells can be kept short and the appropriate outgoing link can be quickly determined.

The Fairisle switch consists of two types of component: port controllers and a switching fabric. The port controllers map VCIs, manage queues and determine the appropriate outgoing links that the cells must be switched to. They also manage the signalling protocol to set up virtual connections. Each port controller is connected to one input and one output link of the switch, and to the switching fabric. We can notionally view each port controller as being made up of two halves: an input port controller and an output port controller. A cell arrives at an input port controller on an incoming transmission line. It then passes through the fabric arriving at an output port controller. The output port controller transmits the cell on its outgoing transmission line. This is illustrated in Figure 1.

The switching fabric switches cells from input port controllers to output port controllers. It is a very regular interconnection network and is the place where cells contend for resources. If different port controllers inject cells into the fabric at the same time that are destined for the same output port controller, then only one will initially succeed. The others will be rejected and must retry later. The switching fabric does the arbitration between such cells. It is also possible in large fabrics, for cells destined for different output ports to clash. This is termed *blocking*. It occurs in fabrics which are not fully connected; that is the internal routes between different input and output ports share common internal paths. The Fairisle switching fabric consists of a series of identical switching elements connected in a regular array. The simplest switching fabric consists of a single crossbar element. Such a fabric is non-blocking—cells will only clash with cells destined for the

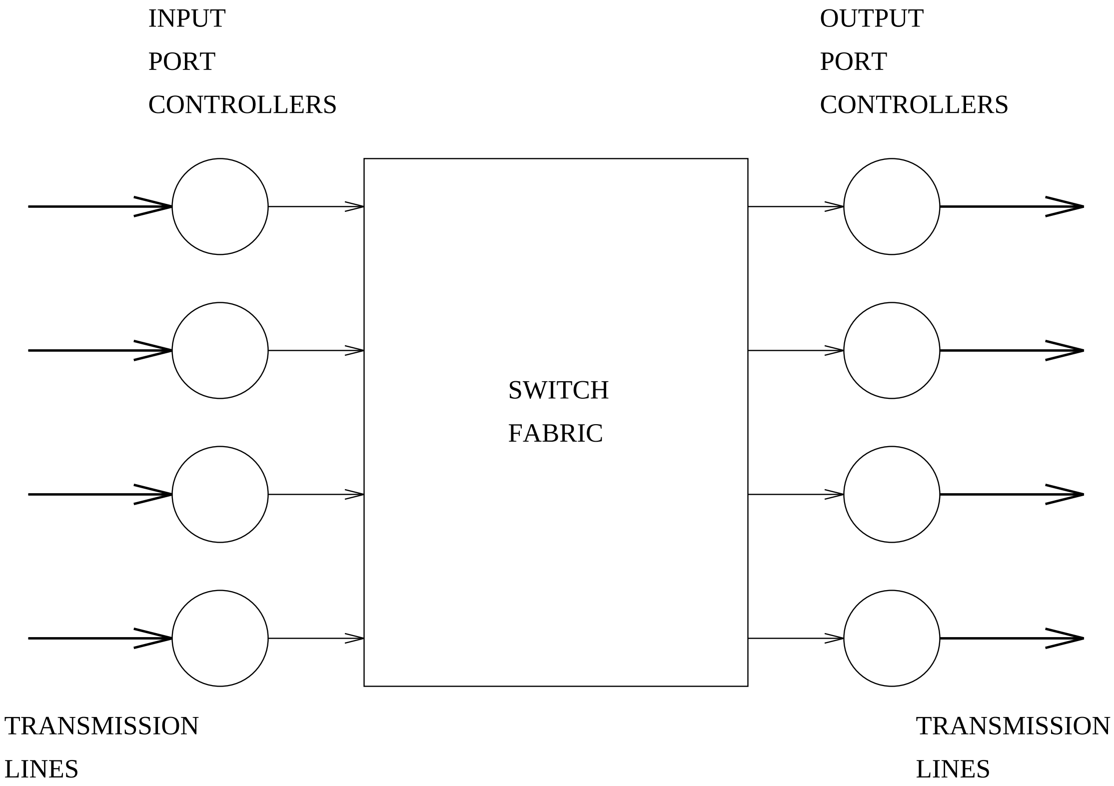

Figure 1: The Fairisle Switch

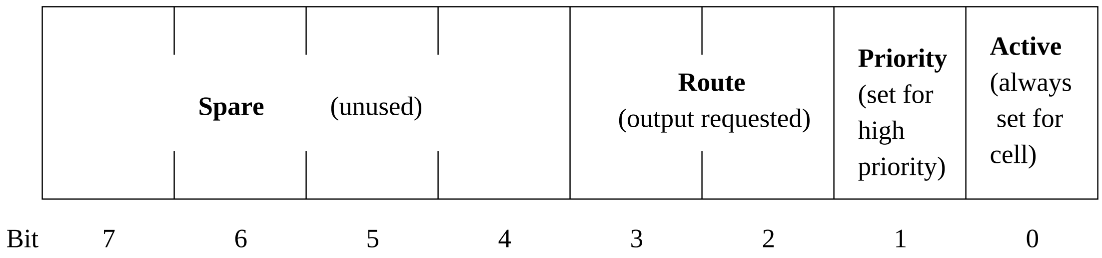

Figure 2: The Routeing Tag for a 4 by 4 Switch

same output port. This is because the elements are fully connected. Each input of the fabric has a dedicated link to each output.

The fabric is self-routeing. The port controllers append to each cell a *routeing tag*. It has the format shown in Figure 2. It indicates the outgoing transmission link the cell should be transmitted on. The routeing information is removed as the cell passes through the fabric. The routeing tag also includes one bit of priority information which is used by the fabric when arbitrating clashes. The fabric does not make use of the information in the original header which is only used by port controllers. It is just treated as additional data in the cell. The routeing tag contains all the control information.

Arbitration takes place in two stages. Firstly, high priority cells are always given precedence over low priority ones. Of the remaining cells, the choice is made on a round-robin basis. The input port controllers are informed of whether their cell was successful using acknowledgement lines. The fabric sends a negative acknowledgement to the unsuccessful input ports, but passes the acknowledgement from the requested output port to the successful input ports. This means the output port controllers may reject cells even if they successfully passed through the fabric.

3 The Behaviour of the Fairisle Switching Element

The Fairisle switching element is a 4 by 4 crossbar switch. That is, it connects 4 input ports to 4 output ports. It can be used on its own as a 4 by 4 fabric or in conjunction with other elements to give larger fabrics. Its purpose is to detect cells, to detect clashes between cells, to arbitrate between cells which have clashed with reference to their priorities, to switch data from the 4 inputs to the outputs, and to send appropriate acknowledgements back to the inputs. A cell can be switched from an input port to an output port without interference with cells travelling from a different input port to a different output port. However, 2 cells will clash if destined for the same output port at the same time.

The element has four inputs: the clock signal, the data-in, acknowledgement-in, and frame start signals. It has two outputs: the data-out and acknowledgement-out signals. Cells arrive from and are transmitted to the input and output port controllers along the data-in and data-out lines, respectively. There are 4 of each, one per port, consisting of 8 unidirectional lines. Cells are thus input a byte at a time. The acknowledgement signals allow acknowledgement information

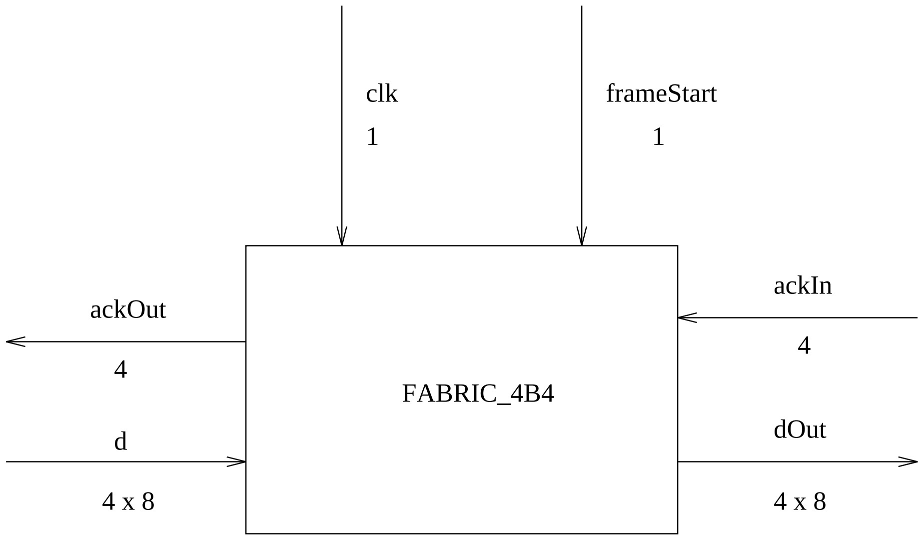

Figure 3: The 4 x 4 switching element

to travel in the reverse direction to the cells. There are 4 acknowledgement-in lines and 4 acknowledgement-out lines. Each output port controller is connected to the fabric by a single acknowledgement-in line. Similarly each input port controller is connected to the fabric by one of the acknowledgement-out lines.

The port controllers and fabric all use the same clock so bytes are read in on each link synchronously. They also use a higher level cell frame clock—the *frame start* signal. It ensures that the port controllers inject cells into the fabric synchronously so that the routeing bytes arrive at the same time. The interval between successive occasions when the frame start signal goes high determines the cell size. Since it is generated externally, the fabric element is not restricted to any particular cell size. In the current implementation of the Fairisle switch, the frame start signal occurs every 64 byte clock cycles.

The behaviour of the switching element is cyclic. In each cycle, the element waits for cells to arrive, reads them in, processes them, sends successful ones to the appropriate output ports and sends acknowledgements. It then waits for the next round of cells to arrive. The boundaries of separate cycles are determined by the frame start signal. Whenever it goes high, a new cycle commences. When cells are not being offered by the input ports, they must inject zeros into at least the first bit of each byte. This is the active bit of the cell header. When a new cycle starts (as signified by frame start going high), the fabric watches this bit from each of the input ports. As soon as one goes high (at the *header time*), then that marks the start of the cells from all the input ports. The fabric does not need to know when this will happen. Indeed it could occur at different times on subsequent cycles. However, all the input port controllers must start sending cells at the same time within the cycle, since any which have not set the active bit at the header time are assumed not to be transmitting cells for the whole of the cycle. If no input port raises

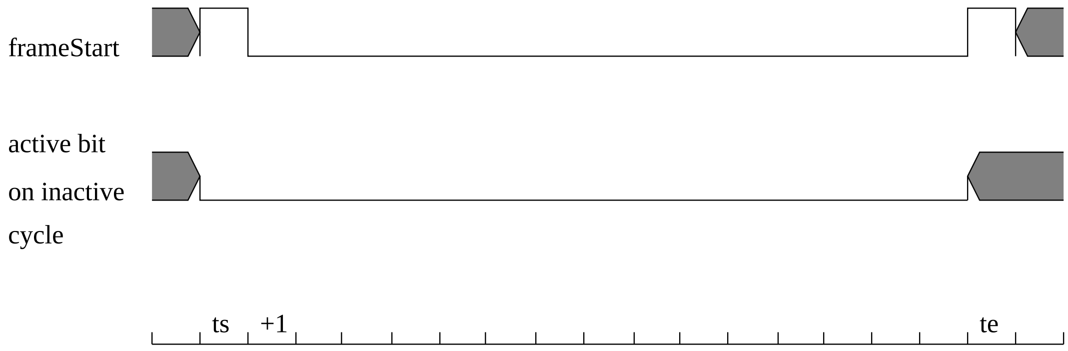

Figure 4: The timing diagram for an inactive frame

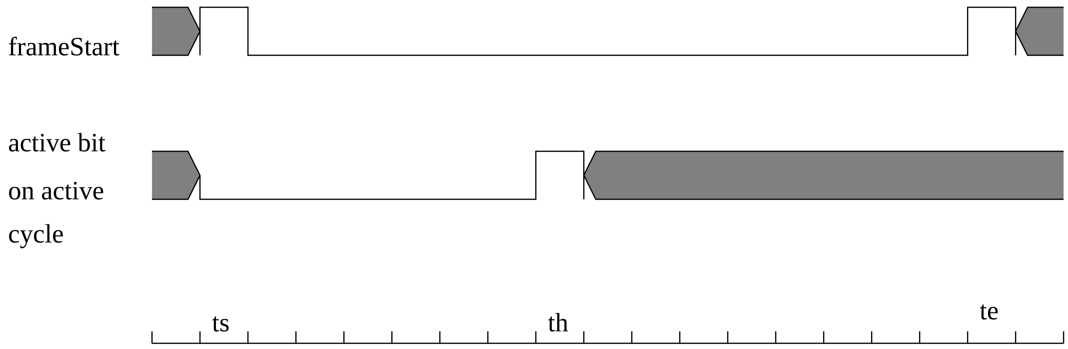

Figure 5: The timing diagram for an active frame

the active bit throughout the cycle then the cycle is *inactive*—no cells are processed. Otherwise it is an `active cycle`.

The way that the fabric is implemented means that in the two cycles prior to the frame start signal arriving, the active bits must be low. This is needed for the fabric element to initialise itself correctly for the forthcoming cycle. Thus, the cell length in bytes must always be at least two less than the duration of the frame. To ensure that this condition is met, the input port controllers must receive the frame start signal before the fabric or be able to predict when it will arrive. Timing diagrams showing the frame start and active bit for inactive and active cells are given in Figures 4 and 5, respectively.

On receiving a set of headers on a particular cycle, the fabric processes them. Each is a request for access to some output port. If only one input port is requesting access to a particular output port then that output will automatically be successful. However, if two or more input ports are requesting access to the same output port, then only one can be successful in this frame. The others must retry in a future frame. The fabric performs the arbitration of such clashes. The first criteria used is that any cell with the priority bit set in the header will be given precedence over those that do not. Thus, if there are one or more cells of high priority requesting a given output

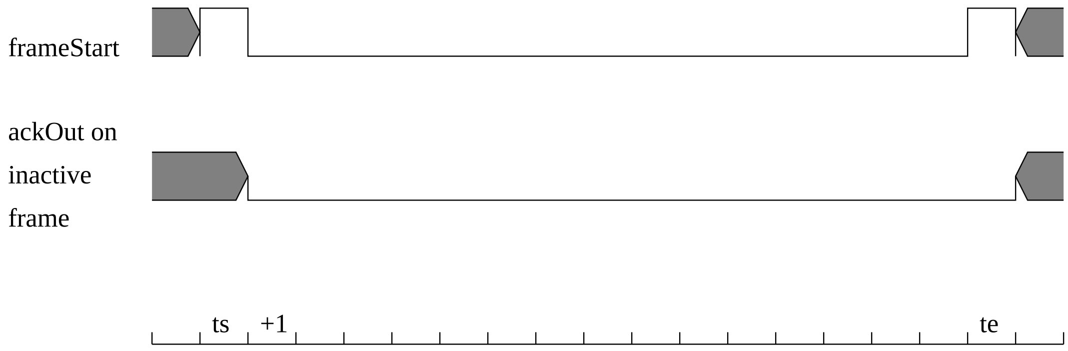

Figure 6: The timing diagram of the acknowledgement signal for an inactive frame

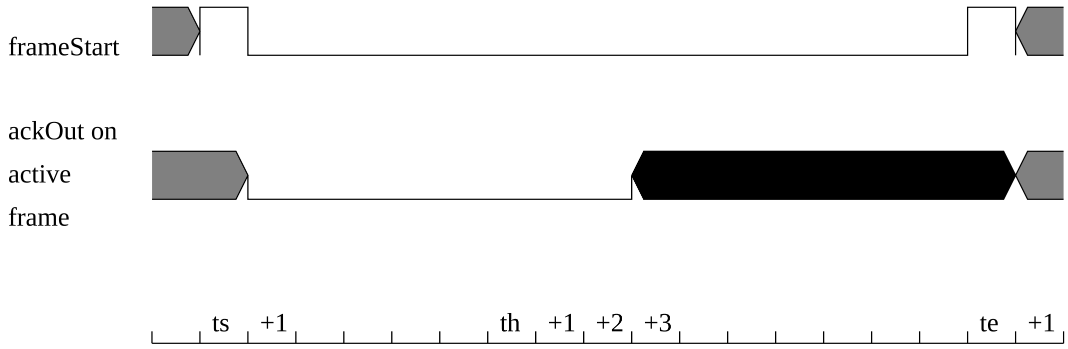

Figure 7: The timing diagram of the acknowledgement signal for an active frame

port, any that are requesting the output port with a low priority will automatically fail. For those remaining, round robin arbitration is performed. The fabric remembers which input port was most recently successful for each output port. For a given output port the new successful input port will be the next in a fixed cyclic order after the most recently successful one that is making a request for the output port in question.

Once it has been decided which requests are successful, acknowledgements are sent, and the successful cells switched to the requested output port.

From the time instance after the start of the cycle, the fabric keeps the acknowledgement-out lines low, indicating that no positive acknowledgement is so far forthcoming. If no cell was injected into the fabric by an input port, its acknowledgement-out line will remain low for the whole cycle. Otherwise it will remain low at least until the fabric has completed the arbitration. For the implementation we consider, this decision is completed 3 time instances after the header time. From that time until at least the end of the cycle, unsuccessful input port controllers continue to see a low acknowledgement-out line. The successful input port controllers are forwarded the acknowledgement from the output port controller that they were requesting (i.e., the signal arriving on the acknowledgement-in of the requested output port controller is placed directly on

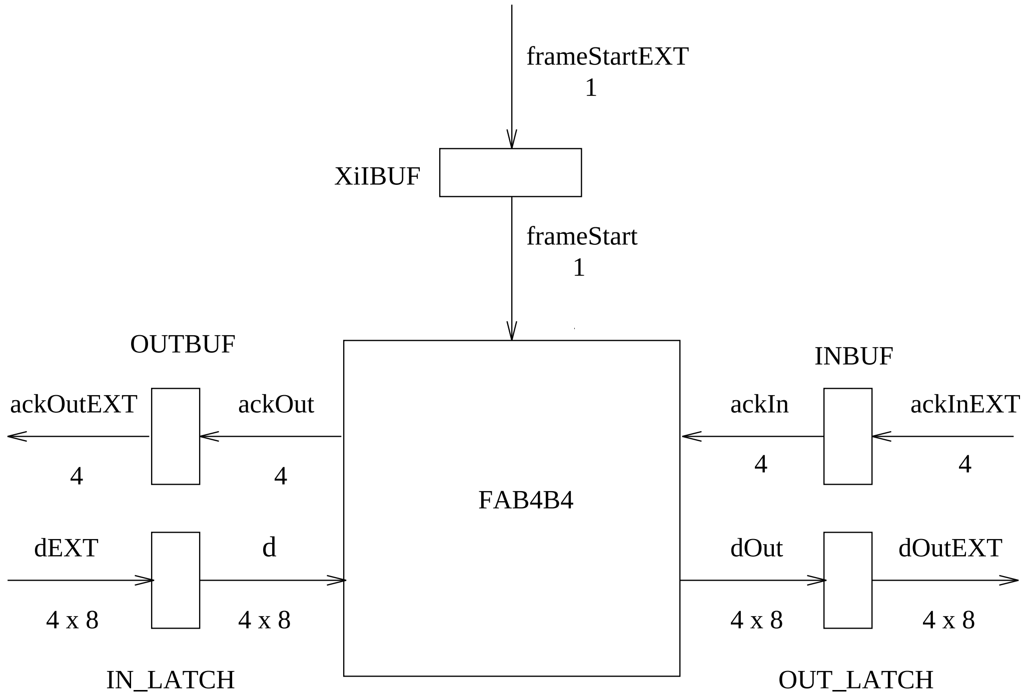

Figure 8: The 4 x 4 switching element: I/O buffers

the acknowledgement-out line). This allows the output port controllers to refuse cells when they are running short of buffer space. The timing diagrams for the acknowledgement signal in inactive and active frames are shown in Figures 6 and 7, respectively.

From 3 time instances after the start of the cycle until the arbitration is completed, the data-out bytes contain a zero in at least the active bit. Cells from unsuccessful input ports are discarded by the fabric element, as are the headers of successful ones. The bodies of the successful cells are output on the appropriate data-out line from 5 time instances after the header time until 3 time instances after the start of the next cycle. The final bytes of a cell are thus output after the end of the frame as delimited by the frame start signal. The last byte output will be the one input just before the frame start signal which ends the frame in which the cell arrived. This cannot be part of the cell since as discussed earlier it must have a low active bit. The output port controller must not treat it as part of either this cell or the subsequent one. It must thus be aware of the length of cells. It must then always discard the byte arriving immediately after the end of the cell.

4 The Implementation of the Switching Element

The switching element is implemented on a 4200 gate equivalent Xilinx programmable gate array. The design consists of the basic switching element, with its inputs and outputs connected to buffers. The data input and output lines are also latched. This is shown in Figure 8.

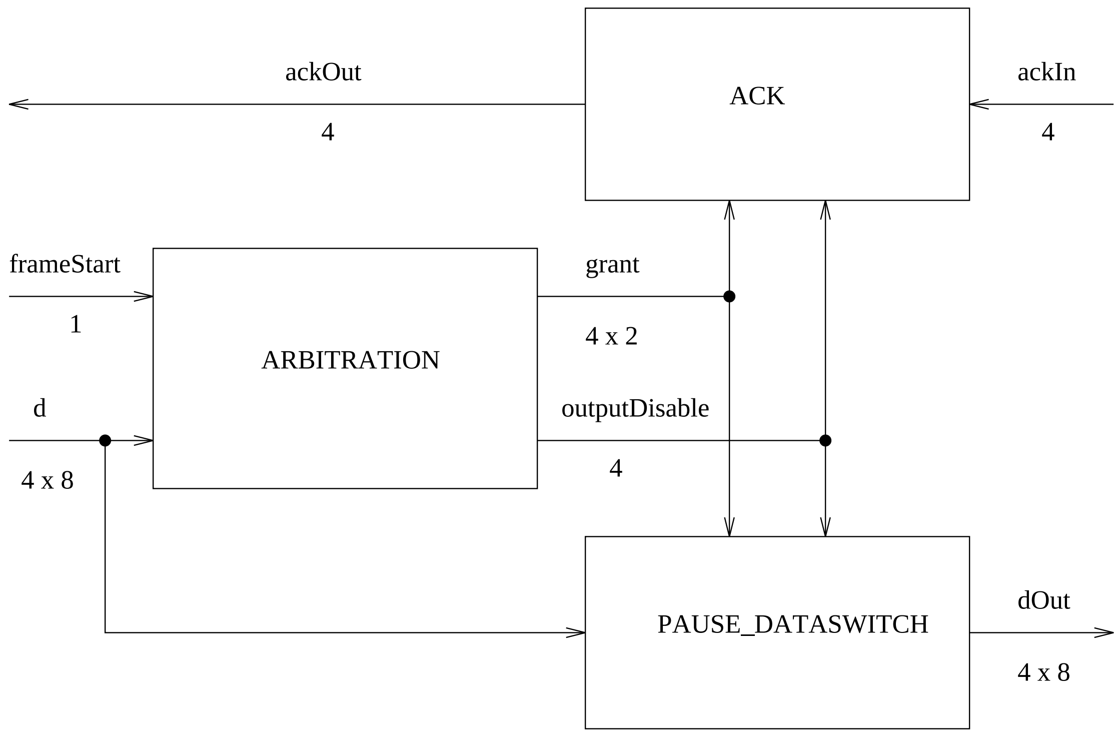

Figure 9: The 4 x 4 switching element: main units

The switching element consists of 3 units: an arbitration unit, acknowledgement unit and dataswitch (see Figure 9). The arbitration unit arbitrates when 2 or more cells are destined for the same output port, and governs the timing of the other units. The dataswitch performs the actual switching of data from input port to output port as demanded by the arbitration unit. It controls the data-out lines. The acknowledgement unit sends appropriate acknowledgement signals to the input ports. It controls the acknowledgement output lines.

These units are repeatedly subdivided until eventually the logic gate level is reached, providing a hierarchy of units. The switching element consists of 43 different elements. At its deepest point this hierarchy consists of 10 levels. The design has a total of 441 basic components where a basic component is a logic gate with any number of inputs (AND, OR, NOR, inverter, or AND-OR) or a one-bit flip flop. The basic components used in the formal verification were roughly those considered basic by the Qudos simulator [6] and that could be used in a Xilinx netlist. The Qudos simulator was used to perform the original (non-formal) validation. The hierarchy is illustrated in Figure 10.

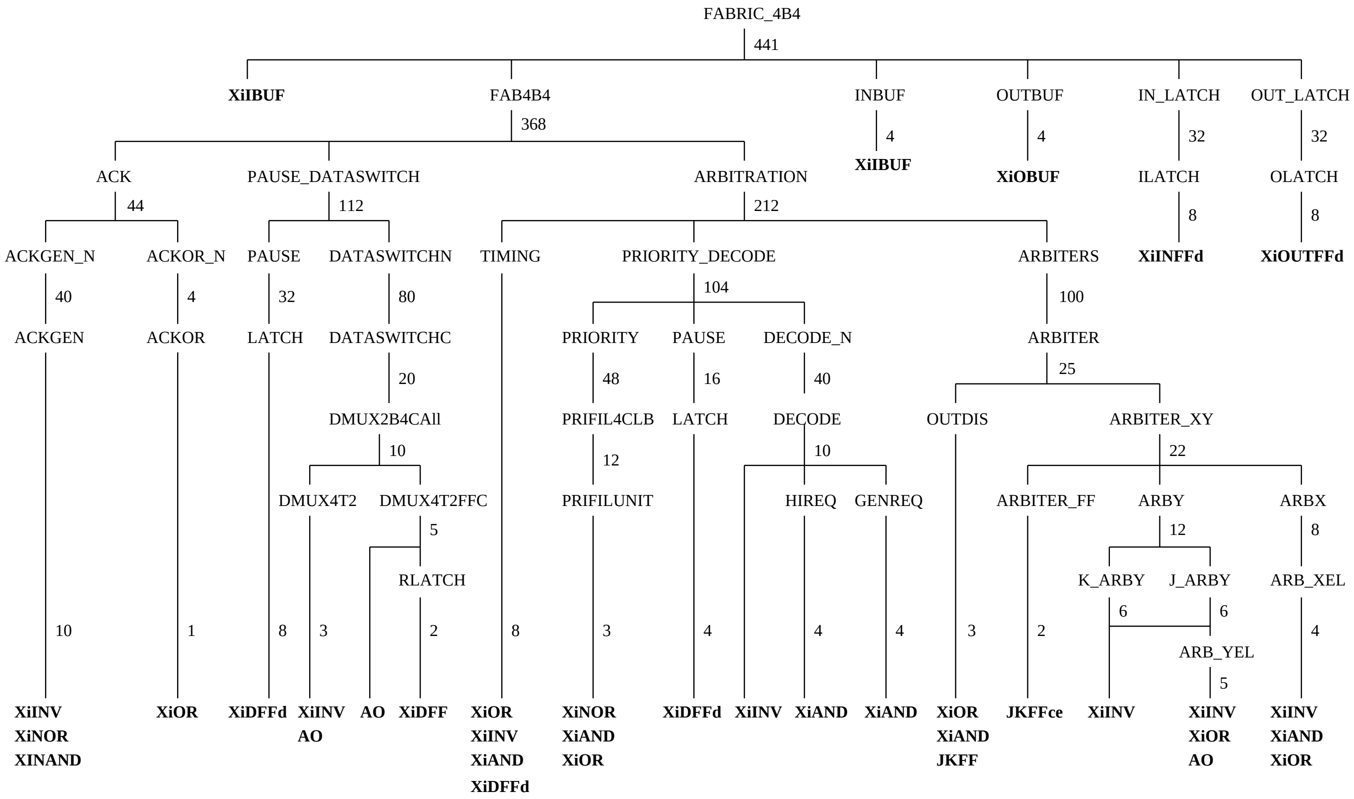

Figure 10: The design hierarchy showing the total number of gates in each module

5 A Tutorial on Formal Hardware Verification

Formal hardware verification consists of mathematically proving that two different descriptions of a design correspond in a suitable way. The two descriptions are the structural specification (the implementation) and the behavioural specification. The two descriptions must be in a formal description language. The meaning of this language must be well-defined. It is also desirable that formal reasoning tools exist for it. In this section we give a brief tutorial on hardware verification and overview the techniques we used to formally verify the switching element using simple examples as illustration.

5.1 Formal Behavioural Specification

A formal specification rigorously describes the intended behaviour of the program. The specification language must be flexible enough to allow the desired behaviour to be expressed simply and clearly. It must also have a well-defined semantics. The specifiers must be able to convince themselves that the formal specification does describe the intended behaviour. There can be no room for doubt about the meaning of the constructs of the language. This is not so for informally defined languages. The language must permit fairly natural descriptions to be given. Otherwise the meaning of a specification will be much harder to determine. The specification language we use is higher-order logic. Higher-order logic was first used in the context of hardware verification by Hanna [8]. The logic is "higher-order" because it allows functions and relations to be passed as arguments to other functions and relations. Higher-order logic is very flexible and has a very well-defined and well understood semantics. There are several verification systems based on higher-order logic. We use one such system—`HOL90`. An overview of the higher-order logic notation that we use is given in Table 1.

As a simple example of a higher-order logic specification consider an inverter with a single cycle delay. Its specification could be:

```
INV (inp:  bool signal, out) =
      ∀t.  out (t + 1) = ~(inp t)
```

This defines the behaviour of a piece of hardware, `INV`. The definition has a single pair of arguments `inp` and `out`. We use the convention that the first entry of the pair represents the inputs of a device and the second entry represents the outputs. The definition relates the values on its input `inp` with those on its output `out`. The values on these signals at any given time are "high" and "low". We model these as the boolean values `T` and `F`, respectively. We then represent the signals by functions from time to a boolean. Time is represented by natural numbers. The value of the signal at a given time is obtained by applying the function to the given time. For example, the value of signal `inp` at time 5 is given by `(inp 5)`: the result of applying function `inp` to time `5`. This is illustrated in Figure 11. Thus, in the above `(inp t)` describes the value on input line `inp` at time `t` and `(out (t + 1))` describes the value at the following time. The type of the input and output is specified by the type annotation `inp: bool signal` which states that `inp` is a boolean signal. Given this information the type of `out` is forced to be identical, as otherwise the definition will

Table 1: Notation

T	truth also used as "high"
F	falsity also used as "low"
$\sim$t	not t
$t_1 \vee t_2$	t_1 or t_2
$t_1 \wedge t_2$	t_1 and t_2
$t_1 \supset t_2$	t_1 implies t_2
$t_1 = t_2$	t_1 equals t_2
$\forall$x. t[x]	for all x, t[x] holds
$\exists$x. t[x]	for some x, t[x] holds
$\forall$x::t1. t2[x]	for all x satisfying t1 x, t2[x] holds
$\exists$x::t1. t2[x]	for some x satisfying t1 x, t2[x] holds
$t \Rightarrow t_1 \mid t_2$	if t is true then t_1 otherwise t_2
t[a/x]	substitute a for variable x in term t
f t	the application of function f to argument t
f o g	the combination of functions f and g
λx. t	function which applied to an argument a has value t[a/x]
let x = t_1 in t_2	let declaration: $t_2[t_1/x]$
(t_1, t_2)	a pair with first element t_1 and second element t_2
[]	the empty list
CONS h t	the list with head h and tail t
$[t_1; \ldots ;t_n]$	the list with elements $t_1 \ldots t_n$
WORD$[t_1; \ldots ;t_n]$	the word of length n with bits $t_1 \ldots t_n$
e:t	expression e has type t
num	the type of natural numbers
bool	the type of booleans
bool signal	the type of boolean signals
ty_1 # ty_2	a pair type
$ty_1 \rightarrow ty_2$	a function type
$\vdash$ th	th is a higher-order logic theorem

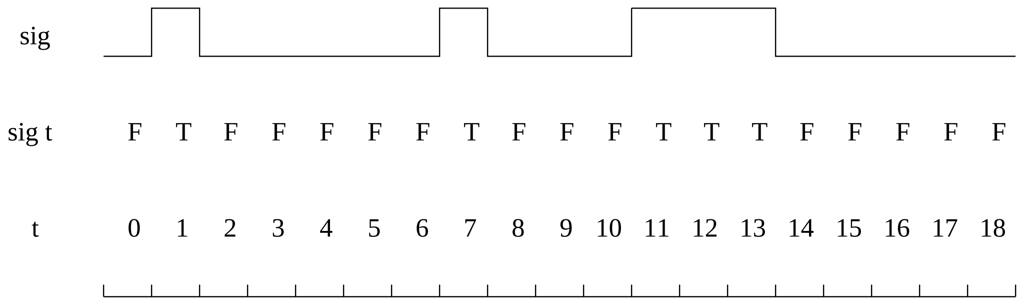

Figure 11: The Modelling of Signals

not type check. In general in this report we do not include type information in definitions as it is either clear from the context or from the English commentary. However, type information must often be supplied for the definitions to be input to HOL90.

We model the inversion of a signal using the boolean negation operator (∼). `∼(inp t)` has as value the negation of the value of `inp` at time `t`. The expression `out (t + 1) = ∼(inp t))` states that the value on output line `out` at some time `(t + 1)` equals the negated value on the input line at the previous time. For an inverter we require this to hold for all values of `t`. This is the purpose of the *universal quantifier*, ∀. The specification thus states that an inverter `INV` relates a stream of input values to a stream of output values if at all times the output value is the negation of the input value at the previous time. It is a relation specifying the values that can appear on the outputs for each set of input values. The relation is true for those pairs of input and output sequences which correspond to a possible behaviour of the device. It is false for values which do not correspond to possible behaviours. Thus the input sequence `[F, T, F, F, T, F ...]` and output sequence `[F, F, T, T, F, T, F ...]`, where `T` represents a boolean true and `F` a boolean false, correspond to a possible behaviour of the inverter. The sequences `[F, T, F, F, T, F ...]` and `[F, T, T, T, T, F ...]` do not.

The higher-order nature of the logic was used in this example. The functions `inp` and `out` are passed as arguments to the relation `INV`. Whilst the use of higher-order relations is not vital for hardware specification, it can make the descriptions much simpler and clearer.

The behaviour of a delay unit can similarly be specified by:

```
DEL_SPEC(inp:  bool signal, out) =
      ∀t.  out (t + 2) = inp t
```

This states that the delay unit outputs values 2 time units after they were input.

The scale of the time unit can be chosen to be whatever is most suitable for the application. The time scale chosen must be small enough that the behaviour of the device is accurately modelled. However, a larger time unit often simplifies the specifications, so if the extra accuracy is not required it should not be used. For example, the above specifications could refer to un-clocked devices in which the time unit used is the smallest unit of concern—perhaps in the order of nano-seconds. Alternatively they could refer to clocked devices in which the clock has been abstracted away. The

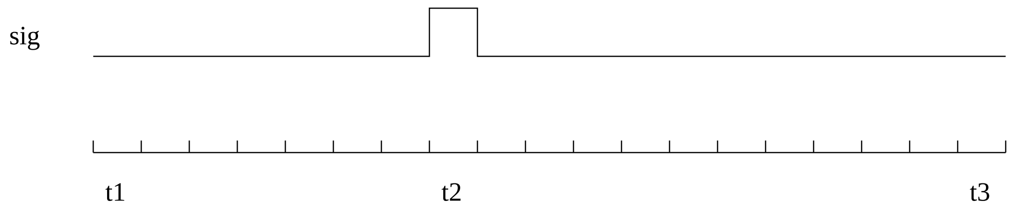

Figure 12: A simple timing diagram

time unit then refers to the clock period. In this situation, because it has been abstracted away, the clock input to the circuit is omitted. It is implicit in the timing aspects of the specifications. Combinatorial circuits are then assumed to have negligible delay at the clock level. For example, the specification of an un-clocked inverter used in a clocked environment might be:

```
INV(inp, out) =
      ∀t.  out t = ~(inp t)
```

This states that the inverter has no perceivable delay at the clock level. Of course real inverters do have some delay, so if enough are connected together, eventually the delay would be greater than the clock cycle. The formal verification of a circuit using this specification for combinatorial logic cannot detect such errors. If that were desired, a more accurate specification using a finer time scale would be needed. In the verification of the switching element, we have used the simplified model so the results do not show the absence of such errors. Other methods must be employed to check that the clock rate is sufficiently long for the design. A formal way of doing this has been given by John Herbert [13]. However, the timing of the fabricated switching element can not be modelled directly in the way suggested by Herbert by giving more accurate timing models of the logic gates. This is because field programmable gate arrays were used to implement the device. Thus the gates in the design do not correspond directly to pieces of hardware.

Frequently the specification of a piece of hardware requires that a signal holds some value over some period. This can be specified using a predicate `STABLE`. The following:

```
STABLE t1 t2 sig v
```

is defined to mean that signal `sig` has value `v` at time `t1` and all subsequent times up to but not including `t2`.

Suppose we wished to specify the behaviour shown in the timing diagram of Figure 12. We can use the formal specification:

```
STABLE t1 t2 sig F ∧
sig t2 ∧
STABLE t2 t3 sig F
```

This states that the signal is initially low, goes high for one cycle as time `t2` and then returns to the low state. We use the conjunction (logical and) operator (∧) to join the separate parts of the specification. The expression `(sig t2)` is equivalent to `(sig t2 = T)`

We also often need to specify that a signal varies with the value of some function over a given period. This can be specified using a predicate DURING. The following:

```
DURING t1 t2 sig1 sig2
```

is defined to mean that signal sig1 has the same value as sig2 at time t1 and all subsequent times up to but not including t2. sig2 is not restricted to being just a named signal such as an input line. It can be a function of other signals. For example we could specify that the signal sig1 was the sum of two signals over a period:

```
DURING t1 t2 sig1 (λt. (sig2 t) + (sig3 t))
```

Here, (λt. (sig2 t) + (sig3 t)) is a function which given a time t returns the sum of the values of the signals sig2 and sig3 at that time. Thus the above specifies that at any time t in the period from t1 to t2 signal sig1 at that time has as value the sum of the other two signals at that time. In general a *lambda* expression of the form (λt. ... t ...) represents a function which takes as an argument something with the type of t. When applied to a value, that value is substituted into the body of the lambda expression in place of t. For example, (λt. t + t) represents the doubling function. Thus, ((λt. t + t) 5) is equivalent to (5 + 5).

5.2 Words

Signals are often logically grouped into words. For example a data line might consist of 8 lines, allowing it to read in a byte at a time. The type of words can be defined in higher-order logic. The HOL system contains a word library which does this [19]. For example, a 4-bit word where the values on the lines are F, T, T and F would be represented by WORD[T; F; F; T]. A signal over words is then a function from time to word values. A 4-bit delay can be specified in a similar way to a single bit one.

```
WORD_DEL_SPEC(inp:(bool word) signal, out) =
      ∀t. out (t + 2) = inp t
```

This states that the register outputs values two time unit after they are input. The only difference between this definition and that for the single bit delay other than the name is in the types of inp and out. Here they are specified to be boolean word signals, whereas previously they were boolean signals. In fact the HOL system uses polymorphic types. This means that the same definition can be used for both cases. A type variable is used to specify the type. The type of any instance can then depend on the usage.

Many word operators are predefined in the library. For example, BIT m w returns the value from position m in the word w. SEG m k w returns a segment of length m starting at position k from word w. WCAT w_1 w_2 concatenates the two words w_1 and w_2. PWORDLEN m w tests if word w is of length m. There are also various arithmetic and bitwise operators defined over words. Corresponding operators over word signals are defined in terms of these. For example, SBIT m sig is a signal consisting of the mth-bit of the signal sig.

As with boolean signals we can specify the value of a word signal over some period using STABLE and DURING. For example,

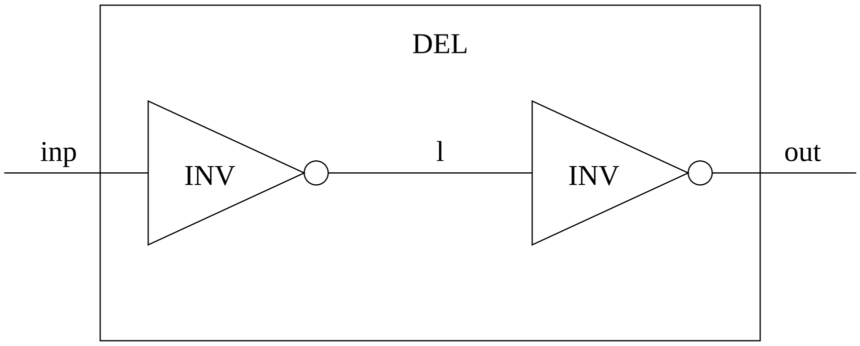

Figure 13: A Delay Implemented by Inverters

```
STABLE t1 t2 sig (WORD[T; F; F; T])
```

specifies that the 4-bit word signal `sig` has value `(WORD[T; F; F; T])` in the interval from `t1` to `t2`.

Words do not need to contain boolean bits. Any base type can be used. For example we can have a word of natural numbers. We can also have words of words. Thus a data line consisting of a bundle of four bytes from different sources could be represented by a word of length four with elements boolean words of length 8. An example of a value on such a line is:

```
WORD[
      WORD[T; T; F; F; T; T; F; F];
      WORD[T; F; F; T; T; F; F; T];
      WORD[F; F; T; F; F; T; T; T];
      WORD[T; F; F; T; F; F; T; T]]
```

5.3 A Formal Structural Specification

Whereas a formal behavioural specification describes the required *behaviour* of a design, a formal description of the implementation describes its structure. It specifies how the components are connected together. Such a description indirectly formally specifies a behaviour. The components will have associated formal behavioural specifications. The behaviour described by a structural specification is that obtained by combining the primitive behavioural specifications in the way indicated by the structure. As with the behavioural specification it must be given in a formal description language with known semantics. As with behavioural specifications, structural specifications can be given in higher-order logic.

For example suppose we require a behavioural specification of a delay unit that is constructed from two inverters as shown in Figure 13. We could use the following:

```
DEL_IMPL(inp, out) =
      ∃l.
       INV(inp, l) ∧
       INV(l, out)
```

This states that the implementation of the delay unit consists of two inverters `INV(inp, l)` and `INV(l, out)`. Components are indicated by applying their behavioural specifications to the signals on their inputs and outputs. The actual connections are specified by using the same signal names as the arguments to different units. The input of the delay unit `inp` is connected to the input of the first inverter, and its output `out` is connected to the output of the second. The output of the first inverter is connected to a line `l`, which is also connected to the input of the second inverter. The two components are combined using the logical conjunction operator $\wedge$: the behaviour of the whole is the conjunction of the behaviours of the parts. Internal lines are hidden from the outside world using *existential quantification*, $\exists$. Thus the line `l` in the above is internal to the delay unit.

As with a behavioural specification, the structural specification is just a relation. It also relates an input signal to an output signal. It therefore also specifies a behaviour. It states that the delay unit relates an input and output signal if there exists some intermediate signal `l` (`∃l`) such that the input signal is related to the intermediate signal as specified by the inverters behavioural specification (`INV(inp, l)`) and ($\wedge$) the intermediate signal is related to the output of the delay unit as specified by the inverter's behavioural specification (`INV(inp, l)`). Thus the input signal `[F, T, F, F, T, F ...]` and output signal `[F, F, F, T, F, F, T, F ...]` are related and so give a possible behaviour. This is because the signal `[F, F, T, F, F, T, F ...]` is a possible intermediate signal that fulfils both inverter specifications.

5.4 Correctness Statements

We have given two distinct specifications for a delay unit. The first is a behavioural specification. It simply and clearly states how the inputs and outputs of the delay unit are related. The second specification gives the structure of a particular implementation. This description is similar to one that might be given in a conventional hardware description language. A netlist could be generated from such a specification. However, as we have seen, it also gives an alternative behavioural description. It is not as easy to see from this description what the behaviour actually is since the behaviour of a device is not always obvious from inspecting the implementation.

Formal verification is concerned with showing mathematically that a structural specification and a behavioural specification of a device do have corresponding behaviours. The correctness statement states this formally. It describes exactly how the two descriptions must correspond.

A suitable correctness theorem for the delay unit would be:

```
∀inp out.  DEL_IMPL(inp, out) ⊃ DEL_SPEC(inp, out)
```

That is we wish to show that the relation giving the structural specification implies ($\supset$) the relation giving the behavioural specification. We must prove that the latter is true for those values of the input and output signals for which the former is true. This ensures that the structural specification does not exhibit any behaviour that cannot be exhibited by the behavioural specification. However, it is possible for the behavioural specification to exhibit behaviour that the structural specification cannot. In the above example we could alternatively prove that the two descriptions are equal. This will not be possible in general, however. For example the specification for a delay unit might say that it must delay the input by at least 1 time unit. An implementation such as the one above specifies that the delay is exactly 2 time units. It thus fulfils the specification, but the two are not equal. Thus, in general the correctness statement for a device will state that its structural specification implies its behavioural specification. For the purposes of correctness theorems the $\supset$ operator can be read as "implements".

A similar correctness theorem could be used for the word delay, assuming its implementation was specified using `WORD_DEL_IMPL`:

```
∀inp out.  WORD_DEL_IMPL(inp, out) ⊃ WORD_DEL_SPEC(inp, out)
```

However, this will only hold if `inp` and `out` are signals of the same length. We can specify this using *restricted quantifiers*. They give a convenient notation for specifying that a property holds for all values that satisfy some predicate. The notation used is a combination of that used for normal quantifiers and for type annotations, using a double colon rather than a single one. For words, if the predicate used is `PSIGLEN n`, then we are specifying that the property holds for all signals of length `n`. Thus, the required correctness theorem is:

```
∀n.  ∀inp out::PSIGLEN n.  WORD_DEL_IMPL(inp, out) ⊃ WORD_DEL(inp, out)
```

This states that for all values of `n` and all signals `inp` and `out` having length `n` the structural specification implements the behavioural one.

5.5 Proof of Correctness

The formal verification of a device consists of mathematically proving that its correctness statement is true. That is, a reasoned argument must be constructed to show its truth.

Informally, the reason the correctness statement for the delay unit is true is as follows. The first inverter delays the signal by one time unit, and negates it. Thus the signal on the internal line `l` will be delayed and inverted. The second inverter takes this signal and delays it by a further time unit and negates it again. Thus the signal is delayed by a total of 2 time units, and is negated twice. A double negation has no effect on the final signal. Thus the signal output is the same as the signal input but is delayed by two time units. This is precisely the behaviour specified by the behavioural specification.

In the above simple example, such an informal proof may be sufficient to convince us that the correctness statement is true. In fact the implementation is "obviously" correct so even the informal specification appears laborious. However, for a more complex device, such an informal presentation could easily contain non-obvious errors.

A formal proof attempts to overcome this problem using a rigorously defined inference system for the specification language used. For our purposes a formal proof is a finite sequence of inferences in a deductive system. A deductive system consists of a set of axioms (initial theorems that are assumed to be true) and a set of inference rules (rules which say how theorems may be manipulated to produce new theorems). A proof is a sequence of theorems with associated justifications. Each theorem will be either an axiom or will follow from earlier theorems in the sequence using one of the inference rules. The justification indicates which axiom or inference rule was used.

A formal proof differs from an informal one in that only well-defined arguments can be used — those embodied in the inference rules. Furthermore, the arguments are reduced to "symbol-pushing" manipulations. The inference rules effectively just dictate how strings of symbols (theorems previously proved) can be manipulated to produce new strings of symbols (new theorems). In an informal proof it is left to the reader to determine whether the arguments are convincing as they arise. There is also scope for mistakes to be made due to ambiguities in the language used. The reader of a formal proof must still ensure that the inference rules have been correctly applied. However, the inference rules of a particular deductive system are rigorously defined. Hence, such a check is a mechanical process. In an informal proof the reader must decide whether the step is valid in the first place in addition to whether it has been correctly applied. Writing a formal proof rather than an informal one gives a similar advantage to that of writing a formal specification as opposed to an informal one. The former defines rigorously what is intended. There is no scope for confusion to arise due to the ambiguities of the language used.

The main advantages of informal proofs are that large steps can be made, and that the proof is easy to read. However, these are also its disadvantages. It is when making large steps that mistakes are most easily made, and proofs that are easy to read by humans are difficult to read by computers. A formal proof on the other hand can easily be checked by a computer. It may also be possible for a computer to do some parts of the proof automatically.

A pragmatic approach is to provide an informal outline of the proof in English that can be read by a human and a fully formal one that can be machine checked. Little extra work is involved since when constructing proofs it is useful for the verifier to have an overview in mind. An informal sketch is a help rather than a hindrance when producing a formal one.

Hardware verification, whether using formal or informal proof, can be performed using pencil and paper, just as any form of mathematics can be done in this way. However, the nature of the proofs makes this undesirable. They tend to be very long with lots of fiddly detail. Humans can easily make mistakes. Fortunately, this is precisely the type of thing that computers can perform accurately and quickly. Thus it is highly desirable to have some form of machine assistance. The degree of automation in such systems can vary to a great extent. At one extreme lies the proof checker. It simply checks that a series of steps given by a user do correspond to a proof of the theorem in question. At the other extreme, lies the fully automated theorem prover. Given the goal to be proved, it constructs a proof automatically. Practical theorem proving systems fall somewhere in between these extremes. The user provides a series of large steps that lead to a proof, with the theorem prover filling in the details. The way such guidance is given and the size of the steps varies between provers.

We have used the HOL theorem proving system. This mechanizes a deductive system for higher-order logic — the language we have used to write our specifications and correctness statements. It is a very flexible system. It contains many pre-proved general theorems and comes with a wide range of tools (ie inference rules) which perform proof steps of varying sizes. However, it also allows the user to program new inference rules on top of these. It is an LCF-style system; that is, type-checking is used to ensure that such user-supplied tools do not perform incorrect proof. The system ultimately performs proofs using a very small set of primitive inference rules. The large step inference rules that provide the interface to the system just perform sequences of these primitive inferences. User written tools use this interface. Therefore the tools also ultimately just perform a sequence of primitive inferences. The type mechanism does not allow something to be called a theorem unless it has been created by a sequence of primitive inferences. The disadvantage of this approach is that it is slower than if the large step inference rules were programmed directly rather than in terms of primitive inference rules. However, this use of primitive inference rules makes the system very trustworthy. Only the basic set of primitive inference rules must be correct to ensure that only valid theorems can be proved by the system. If the larger step ones provided by the system or by the user contain errors, they will either cause exceptions to be raised, or the wrong theorem to be proved. *They will never claim that something is a theorem when it is not.*

In the HOL system a proof can be given in either a forwards manner or a backwards *tactic-based* manner. In a forward proof, we start with the axioms and apply rules until we arrive at the desired theorem. In a backwards proof, we start with a statement of the theorem (the goal) we would like to prove, and apply the inverse of inference rules (tactics). A tactic breaks the original goal into subgoals. If they can be proven then the actual inference rule can be applied to them to give the desired theorem. These subgoals can then be broken into further subgoals in the same way until axioms, or theorems that have been previously proved are reached. The actual theorem is obtained by re-running the proof in the forwards direction, using the actual inference rules rather than their inverses. This is done automatically.

In HOL a goal consists of a set of assumptions and a conclusion. For example, we can set the correctness statement for the delay unit as a goal. The statement given previously is the conclusion of the goal and there are no initial assumptions. We can then apply tactics, gradually manipulating it until it is in the form of a theorem we have already proved or of an axiom. An axiom is something that is accepted as being true without proof. The HOL system has very little trust in something being obviously true: the set of axioms is very small. However, the tactics allow us to jump towards these axioms in large leaps.

A proof is input to HOL as a program in the language ML. Appeals to tactics or inference rules correspond to function calls. These functions call sequences of functions corresponding to primitive inference rules. A low-level proof script, consisting of a sequence of theorems with justifications corresponding to primitive inference rules can be produced by the system. Such a script could then be checked by an independent proof checker [20].

5.6 Hierarchical Hardware Verification

Hardware designs are often hierarchical in nature. The design is split into a series of levels. At the lowest level, hardware modules are implemented in terms of logic gates. Modules on subsequent levels are implemented using modules on lower levels. For example, we could implement a long delay unit using 2 short delay units. We do not need to know how they are implemented.

This approach has many advantages and is common practice. It is particularly useful in the context of formal verification. Firstly, it simplifies the structural specifications. Rather than providing a single definition showing all the interconnections between the logic gates in a design, we can specify each module separately. For example, a 4-tick delay unit constructed from inverters could be specified in terms of that of the DEL units

```
DEL4_IMPL(inp, out) =
      ∃l.
       DEL_SPEC(inp, l) ∧
       DEL_SPEC(l, out)
```

This is clearer than the corresponding description in terms of inverters

```
DEL4_IMPL(inp, out) =
      ∃l1 l2 l3.
       INV(inp, l1) ∧
       INV(l1, l2) ∧
       INV(l2, l3) ∧
       INV(l3, out)
```

Secondly, the implementation of a module is independent of the implementation of its components. This means that modules can be independently verified. Only the behavioural specifications of the components need to be considered, rather than their more complex structural specifications. For example to verify the 4-tick delay unit, we do not need to know about inverters. Furthermore, the correctness theorem can be reused if the long delay is re-implemented using different implementations of the short delay.

The behavioural specification of the long delay could be:

```
DEL4_SPEC(inp, out) =
      ∀t.  out (t + 4) = inp t
```

The desired correctness theorem has the form:

```
∀inp out.  DEL4_IMPL(inp, out) ⊃ DEL4_SPEC(inp, out)
```

This could be proved directly by expanding the definitions and reasoning about the fact that each inverter causes one unit of delay and negates the signal once. This involves more work than is necessary, however. In particular, we would be effectively giving the argument about the effects of placing 2 inverters in sequence twice. For this simple example, this overhead is not too great. For real designs, the verification would become intractable. Instead we can mirror the modular structure of the design in the proof. In particular, we can formulate a structural implementation of the long delay in terms of the behavioural specification of the DEL units.

```
DEL4_SIMPL(inp, out) =
      ∃l.
       DEL_SPEC(inp, l) ∧
       DEL_SPEC(l, out)
```

This says nothing about the underlying structure of the DEL units. By using their behavioural specifications we have "black boxed" them.

We can then prove a correctness theorem:

```
∀inp out.  DEL4_SIMPL(inp, out) ⊃ DEL4_SPEC(inp, out)
```

We do not make any reference to inverters in this proof. We just use the fact that a DEL module causes a 2 unit delay.

We can then prove a theorem which states that the two structural specifications of DEL4 correspond.

```
∀inp out.  DEL4_IMPL(inp, out) ⊃ DEL4_SIMPL(inp, out)
```

The two versions of the structural specification differ only in the use of DEL_SPEC rather than DEL. To prove it, it is therefore sufficient to prove that the DEL structural specification satisfies its behavioural specification. This is precisely the correctness theorem we have already proved about DEL.

We can now combine the above two theorems by chaining the implications. If a implements b and b implements c then we can deduce that a implements c. We thus obtain the desired correctness theorem:

```
⊢ ∀inp out.  DEL4_IMPL(inp, out) ⊃ DEL4(inp, out)
```

This is illustrated in Figure 14. Behavioural specifications are shown as black boxes — they hide the details of the implementation. Structural specifications are shown as open boxes. A structural specification defined in terms of the behaviour of its parts is thus shown with those parts black-boxed. Similarly a structural specification defined in terms of the structure of its parts is shown with those parts as open boxes.

We thus split the task of verifying a design into the sub tasks of verifying each module with respect to the specification of its components. This has the added advantage that if we reuse sub-modules in other modules we do not repeat work unnecessarily. Also if we change the implementation of some modules, we do not have to reverify the whole design. We just prove correctness theorems for the new implementations of the modules, and recombine the correctness theorems. A further advantage is that separate subtrees of the design can be verified independently by different people. The interface between teams occurs at the point where the subtrees are joined. Here, provided the behavioural specifications of the modules are agreed on, the upper levels can also be verified independently.

5.7 Duplication of Components

Hardware often consists of a series of identical components, wired together in some way. For example, an n-bit adder might be constructed from a series of one bit adders with the carry-out from one bit connected to the carry in of the next. A simpler example is of a word delay, constructed from single bit delays. The structural specifications of such designs could be given by specifying each component separately. SBIT is used to specify individual bits. For example, that of the delay unit might be given by:

```
WORD_DEL_IMPL(inp, out) =
      DEL_SPEC(SBIT 0 inp, SBIT 0 out)
      DEL_SPEC(SBIT 1 inp, SBIT 1 out)
      DEL_SPEC(SBIT 2 inp, SBIT 2 out)
      DEL_SPEC(SBIT 3 inp, SBIT 3 out)
```

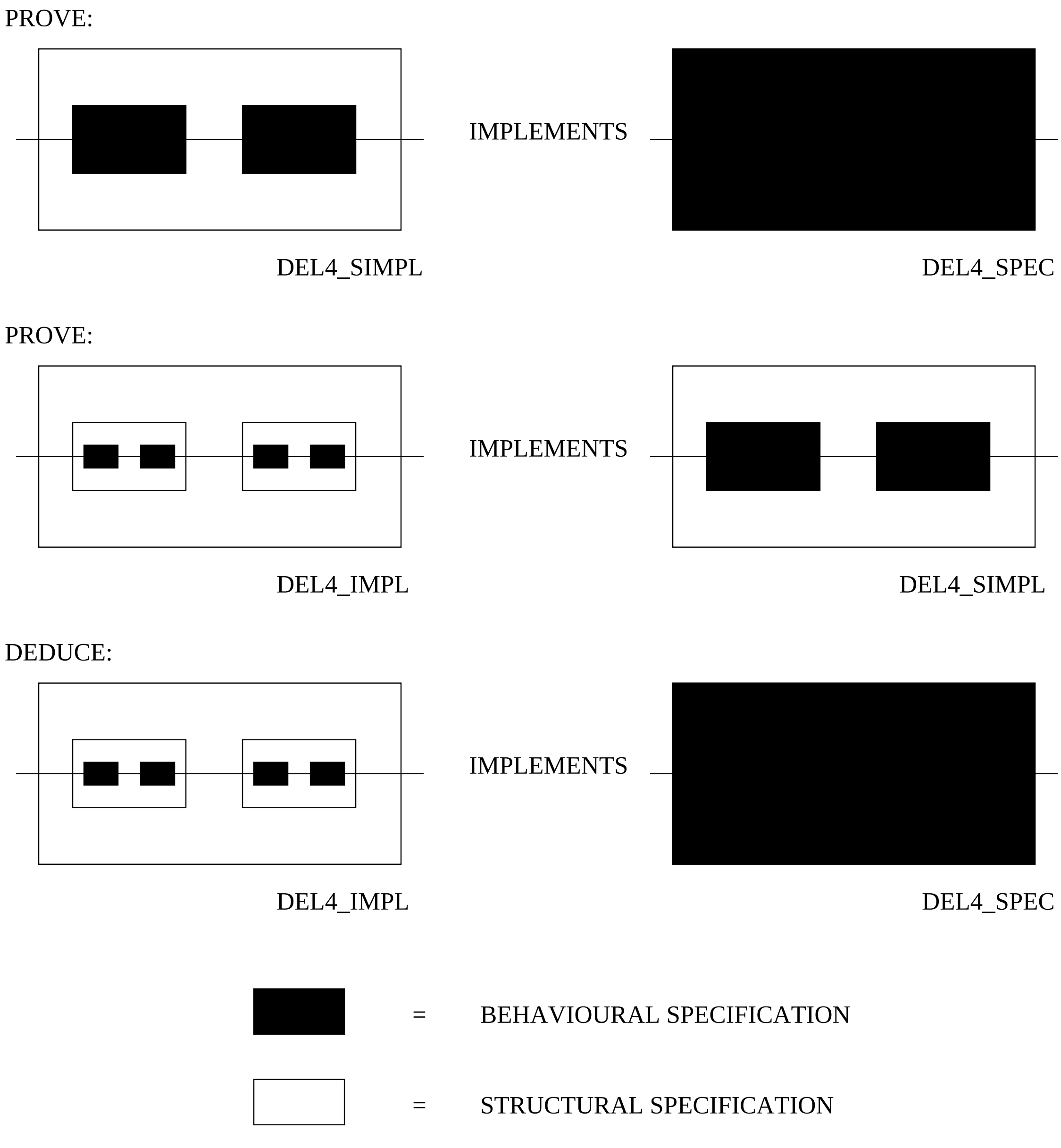

Figure 14: Splitting the proof of a module containing non-primitive sub-modules

However, this is inconvenient if the component is duplicated many times. Instead we provide a duplication construct FOR. The delay can then be specified using:

```
WORD_DEL_IMPL(inp, out) =
      FOR i ::  TO 4 .  DEL_SPEC(SBIT i inp, SBIT i out)
```

FOR introduces an index, here i. It ranges over values from zero up to but not including the value of the expression after TO, here 4, in the body which follows the full stop. Thus the above specifies that there are four occurrences of DEL_SPEC, with inputs and outputs wired according to the index of each. More complex wiring is also possible, by performing arithmetic on the index for example.

A further advantage of using FOR is that the specification can be parametrised over the number of copies to be placed.

```
WORD_DEL_IMPL n (inp, out) =
      FOR i ::  TO n .  DEL_SPEC(SBIT i inp, SBIT i out)
```

The number of copies n is passed as an argument. Thus this one definition describes implementations of delay units of any size. Alternatively, the number of copies can just be specified to be the size of the words involved.

```
WORD_DEL_IMPL(inp, out) =
      FOR i ::  TO (SIGLEN out) .  DEL_SPEC (SBIT i inp, SBIT i out)
```

SIGLEN returns the length of a signal. Thus, this definition also describes implementations of delay units of any size, but now the size is fixed to be that of the output word.

5.8 Formal Verification versus Testing

The results of formal verification are stronger than can be obtained by inexhaustive testing because they apply to *all* combinations of input and output values, rather than to just those combinations chosen by the validater. However, it should be noted that the results relate to *descriptions* of the implementation and specification. If the description of the implementation does not correspond to the actual design then the results will tell us nothing about that design. We have formally verified an implementation down to the gate level. That is, the lowest level units referred to are logic gates. However, the design is actually implemented on a Xilinx programmable gate array so the logic gates in the verified design do not correspond to the reality of the implementation. If the gate array is incorrectly programmed, then the implementation will not correspond to the actual implementation. The same problem arises when using a simulator to test a piece of hardware. If the hardware description used by the simulator does not correspond to the design actually fabricated then the results will not apply to the actual design. The simulator used in the original validation of the design we verified used the same logic gate model. However, tests can normally be rerun on the actual fabricated hardware. Formal verification can only ever apply to mathematical models of the hardware.

Similarly, if the description of the specification does not correspond to the specification actually intended, then even though the implementation has been formally verified it may still not be correct.

However, in this case we will at least have an unambiguous description of what the implementation does actually do. If we have proven that the two radically different descriptions correspond, this gives us a fairly strong assurance that they are both correct. More importantly, in the act of formal verification we examine the implementation and specification very closely. Thus, there is a good chance errors will be picked up. We must have a very thorough understanding of why the implementation implements the specification, as without it formal verification would not be possible. This is a major difference to testing where no such understanding is needed.

A more detailed discussion of the limitations of formal verification is given by Cohn [4]

5.9 Hardware Description Languages

In the above discussion of formal hardware verification, we used higher-order logic as a hardware description language for giving both the behavioural and structural specifications. An alternative would be to use a more conventional HDL, such as VHDL or ELLA. However, to perform formal verification a deductive system would be required for the language used. A prerequisite is that the description language has a formal semantics. This is not so for most hardware description languages, though formal semantics do exist for subsets of ELLA, VHDL and SILAGE, for example [2]. If the formal semantics is given in a logic for which a theorem prover exists, then that theorem prover can be used to perform the formal verification. The hardware description languages described above all have formal semantics written in higher-order logic so the HOL system could be used. To facilitate such proofs additional proof tools can be added to a general purpose system such as HOL, so that the prover does not need to be so aware of the underlying logic. This was done fairly successfully for the ELLA subset [1][3]. However, these languages were not designed with formal methods in mind. As such they contain idiosyncrasies that make reasoning about designs more difficult than necessary.

This contrasts with the use of higher-order logic as a hardware description language. It is very expressive and does not have the problems associated with using a conventional HDL. It does not contain ambiguities. It is also the language used by the theorem prover HOL. Therefore the prover is already tailored to proving facts about descriptions in the language. The disadvantage of using higher-order logic is that it is less familiar to engineers than a conventional HDL, and descriptions cannot in general be simulated (though it is possible to use a subset of the language that is simulatable).

Qudos HDL was used by the designers of the switching element. It is a very simple hardware description language which permits structural implementations to be described but has no facilities for giving behavioural descriptions. It is also very inexpressive. It does provide a simple module facility, allowing hierarchical design. Also a simulator was available for it, so it was possible to validate the design using simulation. Unfortunately, Qudos HDL, has no formal semantics. The meaning of hardware descriptions is effectively defined by the simulator.

One approach would have been to define a formal semantics for Qudos HDL in higher-order logic as described above. Due to the simplicity of the language, this should not have been too difficult. However, the Fairisle designers had expressed reservations with the language. It was too simple. The main problem was the lack of facilities for specifying multi-level words. For example, the data input lines of the switching element consist of 4 words of 8 lines each. In Qudos HDL, this had to be described as a word of 32 single bit lines. This and other similar problems, such as the lack of arithmetic expressions, meant that the description of the design was not as clear as it might have been. The designers were therefore keen to obtain a formal specification which overcame these problems. Also, theorem proving tools for reasoning about the constructs of that language would have been needed. Whilst being straightforward to develop, this would have been time-consuming.

Instead we used a subset of higher-order logic that was similar to Qudos HDL as the hardware description language. Constructs such as `DUP` described earlier were defined to mimic those of Qudos HDL. We also gave formal definitions of the behaviours of the Qudos primitive components used in the design of the switching element—logic gates, flip-flops and buffers. These descriptions form the basis of the proof. We did not formally verify the implementation of these components. We assumed they were correct and that their specifications provided sufficiently accurate models of their behaviour. In a similar way, when using the simulator to validate Qudos HDL designs, it is assumed that the code implementing these same primitive components in the simulator accurately reflects their actual behaviour. We thus performed verification down to the gate-level. This level was chosen because it is the same level as used by the Qudos simulator. As noted previously, the implementation actually uses a gate array; we thus must trust the process by which the gate array is programmed from the logic gate description. This is also the case if the simulator is used for validation. Fears that the silicon compiler could introduce errors could be allayed by formally verifying the silicon compiler. Hardware compilation is an active area of research. For example, work is in progress at Oxford University in this area [10].

The resulting hardware description language (which we will refer to as HOL-HDL) was very similar in structure and semantics to that of Qudos HDL, though more flexible. The surface syntaxes differed somewhat. This could be largely overcome by providing pretty-printing and parsing support as has been done for other hardware description languages. On the whole hardware descriptions in the two languages could easily be related even without such support. Producing HOL-HDL descriptions from the Qudos HDL consisted largely of global editing. The extra flexibility of the former, however, meant we could make some of the descriptions much cleaner than the originals. This made them easier to understand and facilitated formal reasoning about them. We discuss this further in Section 7.

A disadvantage of HOL-HDL is that no simulator exists for it. However, it should be possible to write a translator from HOL-HDL (annotated with values for the sizes of words where this had not been specified and with names for each occurrence of components) to Qudos HDL. This would allow the Qudos simulator to be used. It has not however been done, so the HOL-HDL specifications were not simulated prior to formal verification. The only validation performed prior to verification on the modified descriptions was that of type-checking.

6 The Formal Behavioural Specification of the Switching Element

The formal behavioural specification of the switching element is given by a predicate `FABRIC4B4_SPEC`. The inputs of the element are represented in the formal specification by signals, `data_in`, `ack_in` and `frameStart`. The outputs are represented by `data_out` and `ack_out`. The inputs and outputs are passed as arguments to the definition. The switching element retains state about the last successful input for each output. This is represented by a signal `last` which is passed as a further argument. Finally, the definition is parametrised by the default data value output, `default_data_out`. This is the value sent on the `data_out` line when the data of a cell is not being output. To allow the specification to be applicable to implementations which use different values, the default value is given as an argument to the definition rather than being rigidly specified. The definition therefore has the form:

```
FABRIC4B4_SPEC default_data_out last
      ((data_in, frame_start, ack_in), (data_out, ack_out)) = ...
```

We have omitted the clock input. It is implicit in the specification since all the other inputs and outputs are represented by sequences of values sampled when the clock signal occurs.

The behaviour of each output signal is specified separately. That of `data_out` is specified by the predicate `FABRIC4x4_DATA_OUT`, whilst that of `ack_out` is specified by `FABRIC4x4_ACK`. The state at each time instance, which records the last successful input for each output, is specified by a further predicate, `FABRIC4x4_LAST`. The full definition is:

```
FABRIC4B4_SPEC defaultDataOut last
       ((dataIn, frameStart, ackIn), (dataOut, ackOut)) =
        FABRIC4x4_DATA_OUT defaultDataOut (dataIn, last, dataOut) ∧
        FABRIC4x4_ACK (dataIn, last, ackIn, ackOut) ∧
        FABRIC4x4_LAST (dataIn, last)
```

We will briefly overview the specification for the acknowledgement out signal. The others are similar.

Since the behaviour of the fabric is cyclic, we can give the specification in terms of a cycle or `frame`. We can specify that if some interval represents a frame, then the element will have some given behaviour over that period. We differentiate between inactive frames and active frames. During the former no cells arrive, so no arbitration need be performed. During the latter at least one input port injects a cell into the fabric.

An inactive frame is characterised by two times: the start time and end time. These times are not under the control of the element, but are totally determined by its environment. For two points in time to represent a frame, the frame start signal should be high at the start and end times and low at all times in between. The active signals from all the input ports should also remain low throughout the frame: that is no cells arrive. We model an inactive frame using a predicate `IFRAME`. The predicate is true of a start time, end time, frame start and word of active signals if the above conditions hold and is false otherwise. Its formal definition is:

```
IFRAME ts te fs active =
      FRAME ts te fs ∧
      STABLE ts te (SEXISTSABIT I active) F
```

The first clause gives the requirements on the frame start signal `fs`. It is formalised in a predicate `FRAME`. We do not give its definition here. The second clause gives the requirements on the active signal: each bit in the word of active signals must have the constant value `F` (or low) during the interval from time `ts` to time `te`. The expression `SEXISTSABIT I active` represents a signal that is true at a given time if any of the bits on signal `active` is true at that time and false otherwise.

An active frame is characterised by three times: the start time and end time, as for an inactive frame, and an active time. The active time is the time at which the first byte of the cells arrive. The active signal should then remain low up until the header time when it should go high. It is formalised by the predicate `AFRAME`:

```
AFRAME ts ta te fs active =
      FRAME ts te fs ∧
      STABLE ts ta (SEXISTSABIT I active) F ∧
      SEXISTSABIT I active ta ∧
      ts <= ta ∧
      ta + 1 < te
```

This definition is similar to that of `IFRAME`, except the active signals must be low only until the active time `ta`. At that time it must be true `(SEXISTSABIT I active ta)`. Nothing is stipulated about the active cycle from that point until the end of the frame. The active time must not be earlier than the start time `(ts <= ta)` and must be at least two cycles before the end time `(ta + 1 < te)`.

The specification for the acknowledgement signal then has the form:

```
FABRIC4x4_ACK (dataIn, fs, last, ackIn, ackOut) =
      ∀ts te th.
       (IFRAME ts te fs (Actives o dataIn) ⊃ ...) ∧
       (AFRAME ts th te fs (Actives o dataIn) ⊃ ...)
```

That is, for all times `ts`, `th` and `te`, if they represent an inactive frame then one behaviour holds. If they represent an active frame another behaviour holds. The expression `(Actives o dataIn)` represents a signal made up of the active bits from the dataIn signal (which is a word of four bytes; one from each input port). If applied to a time `t`, the result would be `Actives (dataIn t)`.

The specification does not place any restrictions on the behaviour if the times do not represent a frame. Provided the environment is maintaining the frame structure, the end of one frame is the start of the next. Therefore this covers all time from the first frame occurring for as long as the frame structure is maintained. The fabric is only designed to work provided the frame structure is maintained so we are not required to specify its behaviour when it is not.

The timing diagram for an inactive frame was given in Figure 6. The `ackOut` signal must be kept low throughout the cycle from time `ts+1` until time `te+1` (where the frame start signal arrives at time `ts`). We formally define this behaviour as below.

```
(IFRAME ts te fs ...) ⊃
STABLE (ts+1) (te+1) ackOut (ZEROW (SIGLEN ackOut))
```

This states that for an inactive cycle the `ackOut` signal (which is made up of one bit per input port) is low. `ZEROW` returns a word of Fs of a given length. `SIGLEN ackOut` returns the length of signal `ackOut`. Thus `(ZEROW (SIGLEN ackOut))` represents a word of Fs having the same length as `ackOut`.

The timing diagram for an active frame is shown in Figure 7. Now the `ackOut` signal must be kept low until time `ta+3` (where the headers arrived at time `ta`). Thereafter, the `ackOut` signal to an input port must be kept low if the cell from that port was rejected by the arbitration process. The `ackOut` signal is identical to the `ackIn` signal from the requested output port (indicated by the dark shading in the timing digram) if it was accepted. The formal specification is split into two parts: one describing the behaviour prior to an arbitration decision being made and one describing the behaviour afterwards. Prior to the decision, the `ackOut` signal must be low, as for an inactive frame. The specification is thus similar:

```
(AFRAME ts ta te fs ...) ⊃
STABLE (ts+1) (ta+3) ackOut (ZEROW (SIGLEN ackOut)) ∧ ...
```

We specify the final part of an active cycle using `DURING` which describes the value of the signal `ackOut` over the cycle in terms of a function `(λt. ...)` from time to a four bit word—one acknowledgement bit per input port. We do not give the full details here but note that during this period the value of `ackOut` depends on three things. It depends on the value of the data injected into the fabric at the time the header arrives, `ta`, since the header holds the requests that are behind made. It depends on the value of `last` at time `ta+2`, since this holds the information about the last successful inputs for each output port used in the round robin arbitration. Finally, it depends on the value of the acknowledgements coming in from the output ports, since these are passed on to the successful input ports.

```
(AFRAME ts ta te fs ...) ⊃... ∧
      DURING (ta+3) (te+1) ackOut
       (λt.  ... (d ta) ... (last (ta+2)) ... (ackIn t) ...)
```

The function `(λt. ...)` decodes the headers, filters cells destined for the same output port based on the priority fields in the headers and performs round-robin arbitration on the results. This is performed separately for each output port. The bit of `ackOut` for a particular input is low if it was not selected by the output port it was requesting, or if it were not making a request, and is the acknowledgement from the output port it was requesting otherwise.

The function is defined in terms of several independent functions which, for example, specify what it means for priority requests to be filtered and what round-robin arbitration is. These functions are also used in the specifications of the data out signal and the state holding the last successful requests.

As an example, we will describe the definition of round-robin arbitration. It is specified by a function `RoundRobinArbiter` which has several arguments: the number of input ports, `n`; a set giving the input ports making requests for the output port under consideration, `request_set`; and the last successful input port for this output port, `last`. It returns either an indication that it cannot make a selection if the request set is empty (`NO_RESULT`). Otherwise it returns the result of round robin arbitration.

```
RoundRobinArbiter n request_set last =
      ((request_set = {}) ⇒
       NO_RESULT |
       RESULT(RoundRobin n request_set last))
```

The notation `(a ⇒ b | c)` is a 2-branch conditional. If `a` is true then the result is `b`, otherwise `c`. The function `RoundRobin` does the actual work. It is defined recursively on the numeric argument provided. In the base case when this argument is zero, zero is returned. This should never arise however. The numeric argument is initially higher than any value in the request set. Its purpose is simply to ensure that the recursion terminates. However, it should never be called in such a way. It is given the value of the last successful input and must find the next highest one. It therefore adds 1 modulo 3 to the last successful input `(SUC_MODN 3 last)` giving a possible candidate, `trynext`. As the value 3 is explicitly used in the definition, it is only applicable to round robin arbitration over 4 choices. It could be made more general by using a variable in place of the 3. If `trynext` is in the set of requests, then it is the new successful input. Otherwise, it recurses repeating the process, though with `trynext` as the new "last successful input". Provided the set of requests is not empty and only contains values less than 3, then after at most 4 attempts it will have found the correct result. Thus by giving it a counter with initial value of 4, it will return the required result without reaching the base case.

```
(RoundRobin 0 request_set last = 0) ∧
(RoundRobin (n+1) request_set last =
      let trynext = SUC_MODN 3 last
      in
       trynext IN request_set ⇒
       trynext |
       RoundRobin n request_set trynext)
```

7 The Formal Structural Specification of the Switching Element

To formally verify the switching element, we needed a structural description of the implementation in a language with a formally defined semantics. Without this no formal reasoning about the behaviour of the circuit is possible. As we intended to perform the verification using the HOL theorem proving system, we ultimately needed a semantics in higher-order logic.

We manually translated the original Qudos HDL descriptions to HOL-HDL. On the whole this was a mechanical process, involving the changing of surface syntax. We did however make some changes to the description. These changes were largely superficial. They did not alter the design, only the description of it. Both descriptions describe the same collection of logic gates. This could be checked by comparing the netlists resulting from the two versions. The changes made the descriptions clearer and made formal reasoning about the design much more tractable. Two kinds of changes were made: adding extra layers to the hierarchy and making use of features of HOL-HDL which were not available in Qudos HDL.

When giving the structural specification of the switching element we added several levels of hierarchy that were not used in the original Qudos description. This was done to facilitate the formal verification. For example, in the original description the top level module described the fabric in terms of input and output buffers, latches, a header decoder, priority filter, timing unit, arbiter, dataswitch and acknowledgement unit. We grouped all but the latches and buffers into a single unit at the top level. It was then subdivided at the next level into the arbitration unit, acknowledgement unit and dataswitch, with the arbitration unit further subdivided into a header decoder, priority filter, timing unit and arbiter.

In Qudos HDL there is no facility for describing multi-level words. Thus the data-in lines are modelled as 32 individual lines. Multi-level words can be used in HOL-HDL. Therefore, the data-in line was modelled as a group of 4 8-bit bytes. This is closer to the designers' mental model.

In some modules, the design was not dependent on the word sizes used. In Qudos-HDL, the number of input and output signals must be fixed. The Qudos loop construct which is used to define a set of identical components must be given a value for the number of copies. Thus an n-bit adder cannot be described using Qudos HDL. A separate description must be given for adders of each different size. The HOL-HDL loop construct allows the number of copies to be left as a variable or restricted to being the same as the size of a given word. Furthermore, the word sizes do not always have to be specified. This means that generic descriptions can be made, such as an n-bit adder, where n is not specified. The correctness theorems for some modules of the switching element are therefore directly applicable to switching elements of different sizes.

The different copies of a duplicated component can be wired in more general ways using HOL-HDL than with Qudos HDL. In particular, arithmetic can be used to specify which bit of an input word is connected to which bit of an output word. This meant that the duplication construct could be used in the HOL-HDL version where the copies had to be written out in full in the Qudos version.

To illustrate the kinds of changes made, we will consider the Qudos HDL and HOL-HDL structural descriptions of one of the components of the dataswitch - `DMUX4T2`. It is a multiplexor. We first give the Qudos HDL.

```
DEF DMUX4T2 (d[0..3], x:  IN; dOut[0..1]:  IO);
xBar :  IO;
BEGIN
Clb := XiCLBMAP5i2o (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;
```

The description starts with a declaration part. This first line states that we are defining the module `DMUX4T2` and that it has two inputs: `d` which is 4 bits long (with bit positions numbered from 0 to

3) and `x` which is one bit. It has one two bit output `dOut`. The second line declares a local variable `xBar`. We then have the description of the layout, enclosed by `BEGIN` and `END`. The first statement, is a dummy statement that provides information about the way design should be mapped onto a Xilinx gate array. It provides no semantic information. The next statement describes a Xilinx inverter `XiINV`. It has input `x` and output `xBar`. This particular inverter is given the name `InvX`. This name is used by the simulator. There then follows two AND-OR logic gates, `AO`. They each produce one bit of the `dOut` output, using differing bits from `d` and the `x` and `xBar` signals. They are given the array name `B`, each being one entry. The separate components are delimited by semicolons. The individual bit positions are given in square brackets.

This description can be mimicked in HOL-HDL.

```
DMUX4T2 ((d, x), dOut) =
∃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))
```

We use the convention that definitions of modules take a single pair argument. The inputs form the first part of this pair, and the outputs the second. Multiple inputs and outputs are grouped into a tuple in the appropriate part of the pair. The local variable `xBar` is introduced using an existential quantifier. The three components are then given separated by conjunctions $\wedge$. The CLB definition of the Qudos HDL is omitted as are the names of the components. They provide no semantic information and are not needed to perform formal verification. If a specification were to be originally written in HOL-HDL and translated to Qudos HDL for simulation, this additional information could be provided in the form of annotations or dummy definitions that throw away the extra information. The bit positions are indicated using the function `SBIT`, and the inputs and outputs have the structure described above, but otherwise the component descriptions are the same.

In HOL-HDL, we can do better than the above. `d` can be thought of as being two signals each two bits wide. The input `x` chooses either the first bits of each of these signals or the second. We cannot describe this in Qudos HDL as structured signals are not supported, but we can in HOL-HDL.

```
DMUX4T2 ((d, x), dOut) =
∃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))
```

Now `d` is a two level signal. The low level bits are accessed using two calls to `SBIT`. The use of the two level signal makes the description closer to the designers' mental model. Furthermore, whilst making the structural specification look superficially more complex, it simplifies the behavioural specification. This makes that specification much easier to understand and also simplifies the verification task. We can now simplify the structural specification further. The two `AO` gates are performing identical functions, the only difference is in the bit positions. We can therefore introduce the duplication binder FOR. It introduces an index variable i, to range over the different values of the bit positions, from zero up to, but not including, the value after the keyword TO.

```
DMUX4T2 ((d, x), dOut) =
∃xBar.
      XiINV(x, xBar) ∧
      FOR i ::  TO 2 .
       AO ((SBIT 0 (SBIT i d), xBar, SBIT 1 (SBIT i d), x), SBIT i dOut)
```

The body of the duplication binder FOR describes the replicated components in terms of the index, here i.

8 The Formal Verification

Each module in the design was formally verified separately. The correctness theorems for modules implemented in terms of primitive components were proved directly. Those constructed from non-primitive modules were proved in three steps using the structural specification based on the behavioural specifications of the modules. This approach was outlined in the tutorial section. The proofs were roughly of two kinds. The simplest were for the low level modules. Their specifications were in the form of an equation stating that the outputs at some time were a function, say f, of the inputs at earlier times:

```
value of output at some time = f (values of inputs at earlier times)
```

The specifications of the sub-modules had a similar form. The proof proceeded by rewriting the specification of the module with the specifications of its sub-modules. This involved replacing occurrences of outputs in the specification with functions of the input values. This was done automatically. It yielded an expression of the form:

```
g (values of inputs at earlier time) = f (values of inputs at earlier times)
```

That is: one function g of the inputs was equal to the original function, g of the inputs. The proof was then completed by showing that the two sides of this equality were the same. This was generally straightforward, typically requiring lemmas about words.

A second kind of proof involved modules with specifications based on timing diagrams over the period of a frame. For example, the value of an output in the interval up to the header time and after that time within a frame were given.

```
STABLE ts th output F ∧
STABLE th te output T
```

In general, a separate proof was performed for each output over each interval. These proofs normally involved considering a typical point within the interval, and showing that at that time the output in question had the value specified. In some cases, the start of an interval was treated separately, because at that point a change of behaviour occurred. The reason for the change of behaviour was often different for the reason the behaviour then remaining constant—for example, the frame start signal might trigger the change in behaviour at the start of the interval putting the module into a different state. Once in that state it remains there as long as no active signal occurs. The reasoning required in these cases is different so they are proved separately.

For modules with sub-modules specified as equations, the proof was split into two parts. A second equational specification of the module was written. The implementation of the module was first verified against this behavioural specification. The proof was of the first kind described above. The second behavioural specification was then shown to satisfy the original. This was easier than to prove the implementation satisfied the original because the equational specification was simpler than the implementation.

Separate proofs were also performed for active and inactive frames. The proofs of the latter were virtually identical to the proofs for the start of the interval of an inactive frame.

As with proofs about equational specifications, the proofs of interval specifications involved an initial, largely automatic part concerned with expanding definitions and rewriting the values of output signals using the specifications of the sub-modules. This was followed by a user-guided part, which involved proving that two expressions on the input values were the same. These proofs were more tricky, due to the more complex notions being reasoned about: for example, modules involved with arbitration require reasoning about round-robin arbitration.

The higher level modules often had several outputs, with each output described in terms of up to three intervals. Thus there were many cases to consider. In addition, each case dealt with complex notions. Consequently these proofs tended to take longer than proofs of lower level modules.

9 Time taken

The project made use of the new HOL word library. Unfortunately, it was not available in the version of HOL used from the start of the project. This meant that initially only specification could initially be done rather than proof. It was therefore decided to write all the specifications first. By the time this was done the word library was available and so verification could proceed.

Prior to the verification work, over a month was spent writing the formal specifications, (both behavioural and structural) for each of the 43 modules. No detailed breakdown of this time has been kept however. Much of the time was spent attempting to understand 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 or of the Fairisle Network. Documentation was minimal. There was a good English overview of the intended function of the switching element. This also outlined the function of the major components. Whilst it gave a good introduction, it was not sufficient to construct an unambiguous behavioural specification of all the modules. The behavioural specifications were constructed by analysing the HDL due to the lack of documentation. This was very time-consuming. It also meant that the formal specifications described what the implementation did, rather than describing what the designer intended it to do. This leaves open the possibility of errors in the design becoming "features" of the specification, rather than being identified as errors. However, since the switching element was designed and in use prior to the formal specification and verification being carried out, this is unavoidable. The behavioural specification must be studied by the designers to ensure that the behaviour described does not contain undesirable features. This is easier than just studying the structural specification, because the behavioural one is a much simpler and clearer description. Ideally, formal specifications should be written by the designers of the module. This would also aid the design process.

Approximately two man-months were spent performing the verification. Of this one week was spent proving general purpose theorems about, for example, machine words and signals, that were needed for the proof. These theorems will be of use in further proofs. They had not previously been proved by others because they concerned the new word library. This had only been used on one other verification project and contained only the theorems required there. In particular there were no definitions or theorems concerning multi-level words. Approximately 3 weeks were spent verifying the upper modules of the arbitration unit, and a further week was spent on the top 2 modules of the switch. 3.5 days were spend combining the correctness theorems of the 41 modules to give a single correctness theorem for the whole circuit. The remaining time of just over 2 weeks was spent proving the correctness theorems for the other 36 units. These proofs were largely automated using tactics which were developed throughout the verification, though in most cases additional human effort was required to finish the proofs. The breakdown of the time is given in Figure 15. It shows the cumulative time taken as each module was verified. The time spent proving general word theorems mentioned above is not included. On the whole, the simpler modules were verified first. The latches and buffers were particularly easy as they all had very similar structures. Tactics developed for the early proofs were then used in the later proofs, thus speeding up the more difficult ones. Also, the verifier had not previously performed a hardware verification and was not initially familiar with the word library, though was a competent `HOL` user. Proof times were consequently reduced as experience was gained.

Much of the effort was expended in understanding informally why the implementation was correct. This was hampered to some extent by the delay between writing the specifications for a module and doing the proof. This meant that much time was wasted re-understanding the specifications and working out how the implementations actually worked. It would have been much better to perform the proof of a module when it was specified had this been possible.

Errors in the specifications slowed progress. When an error was present in a module, some time was spent attempting to prove theorems that were untrue. When this was discovered, the error had then to be located. The manner in which the proofs failed usually gave a strong indication as to the location of the error. However, it was not always immediately clear whether the error was in the behavioural or structural specification of the module, or because a component module's specification was too weak. Determining which involved examining the specifications of other modules as well as the faulty one. The specification then needed to be corrected and the proof completed. The main errors were in the modules `DMUX2B4CAll`, and `ARBITRATION`, where there are corresponding plateaus in the graph. The original behavioural specification for the latter contained several minor errors and some more serious ones. It was largely rewritten during the verification. We discuss the errors found in the next section.

After the proof had been successfully completed the behavioural specification was reviewed. Major changes had been made to the behavioural specification of the element during the course

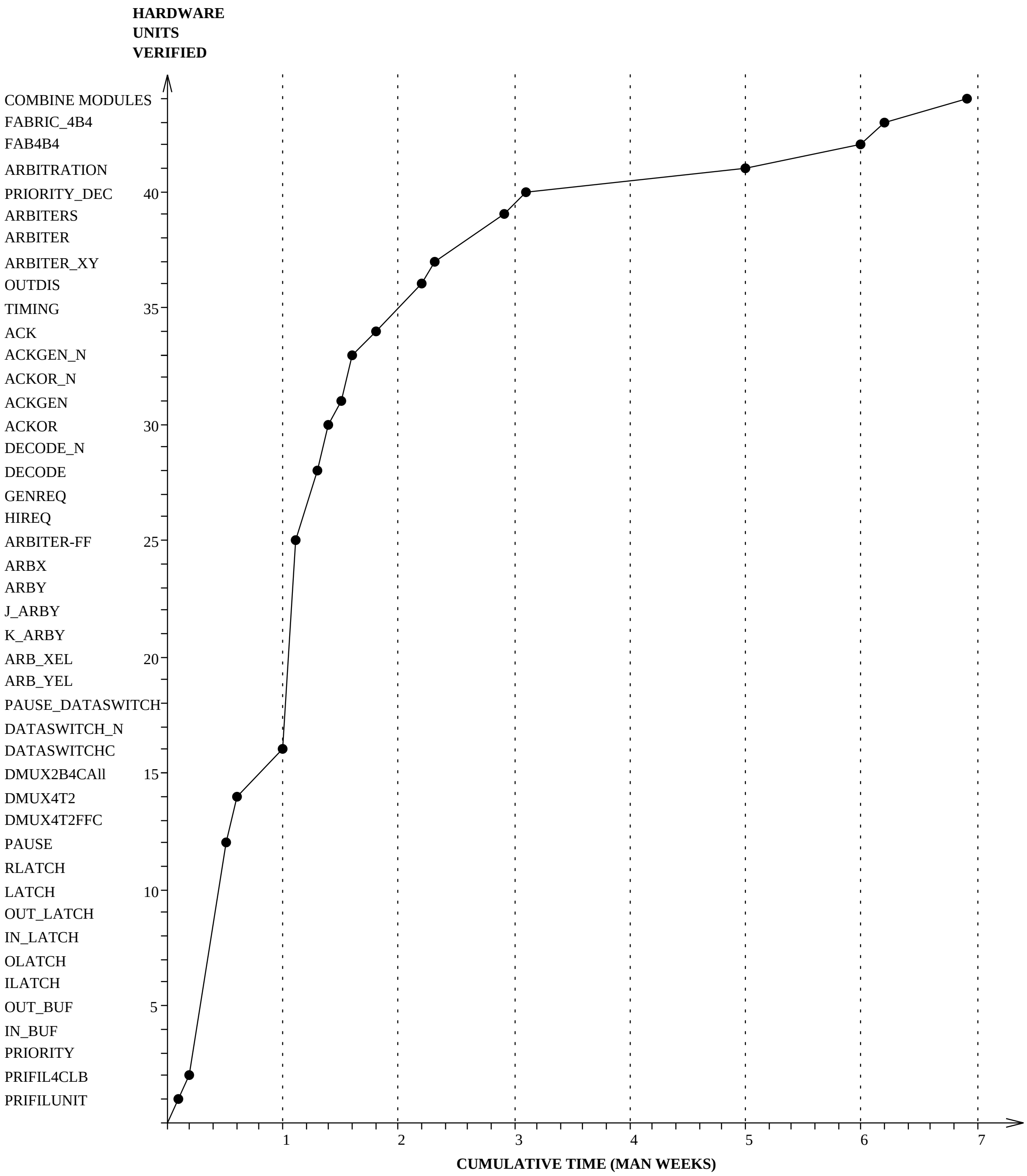

Figure 15: Time taken to verify the switching element

of the verification. Consequently, some aspects of the specification were overly complex. The specification was therefore changed and the proof redone. In the original version, a single predicate had been used to specify that a set of times represented either an active frame or an inactive frame. It was originally thought that this would simplify the specification as an inactive frame is just a degenerate case of an active frame. However, the two cases did ultimately need to be specified separately. The specifications were therefore modified to use two predicates one for an inactive frame and one for an active frame as described earlier. This involved the major reworking of the proofs for the timing module, the upper level modules of the arbitration unit, and the upper level modules of the fabric itself. Due to the modular nature of the design and proof, only the modules which were affected needed to be reverified. It took a total of one month to complete the original proofs of these modules. It took less than three days to modify the specifications and redo the proofs. Re-doing the proof was quicker for several reasons. The proofs were split into a series of lemmas. Many lemmas were not affected by the changes and so their proofs did not need to be redone. The new specifications did not contain errors. The reason why the design was correct was well-understood because at an abstract level it was unchanged from the original. Most proofs that did need to be changed only needed to be changed in small ways. Thus the scripts could be rerun with only a few modifications. Some proofs were even simplified by the changes.

No detailed record of the time spent originally designing and testing the element was recorded. The design evolved from earlier designs, and several different designs were produced at the same time, making it difficult to accurately estimate the time scale involved. However, the designer estimated that had it been designed from scratch the initial design time would have been in the order of several months. The time spent testing would have been in the order of several weeks. However, errors were discovered after the testing process had been completed when the fabric was in use. Thus the time spent to formal specify and verify the design was not unreasonable. Had it been performed as an integral part of the design, it is unlikely that it would have unduly slowed the design cycle. Furthermore, it is likely that the formal specification and verification would have been much quicker if done as the element was designed since much of the time was spent attempting to understand the design. Had formal verification been applied to the ancestors of the design, the formal verification could possibly have tracked the changes made, with a minimal amount of time spent adapting the proof for the new generation. Similarly the proof could have been quickly adapted to the other versions of the element that were designed at the same time.

10 Errors Found During Formal Verification

In this section we give an overview of the errors discovered during the formal verification process in the various descriptions of the switching element. We outline the various kinds of errors which occurred. In the more interesting cases, we give an indication of why they occurred and how they were discovered. Our aim is to give a flavour of the wide range of errors that can occur in formal specifications as well as in implementations and how formal verification can help in their detection. All errors found during the course of the verification were corrected and the proofs completed successfully.

10.1 The Implementation

No errors were discovered in the actual implementation of the switching element. This is not surprising, since the fabric has been in use for some time. Also as noted above, the behavioural specification was written by examining the implementation. It is therefore possible that discrepancies between the implementation and designers' intended behaviour have become "features" of the behavioural specification.

10.2 The Structural Specification

Several errors were found in the `HOL` structural specification. These were introduced in the translation from Qudos HDL, due to the introduction of multi-level words, etc. For some errors this was due to the specifier misunderstanding the original description.

In several places the wrong number of copies of a unit was specified. For example, in `DMUX4T2`, `(FOR i :: TO 1 ...)` was originally used to replicate the `AO` module. This created only one copy rather than the required two. This was because Qudos HDL takes an initial index and final index rather than the number of copies we used. This could have been avoided if the duplication construct of HOL-HDL had been defined to mirror the HDL more closely. This had originally been intended. The length was eventually used since this made generic specifications simpler. In other modules, where a piece of hardware was duplicated using the length of one of the signals, the length of the wrong signal was used. In several modules, the sizes of the local signals were not specified, and this information was needed in the proof.

In `FAB4B4`, 2 bytes of a signal were selected when actually it should have been 2 bits from each byte.

In `DMUX2b4CAll` and `ARBITER`, two signals were incorrectly wired. 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. This illustrates how the discovery of an error can give a strong indication of its cause. It was clear from the proof attempt that two signals had been swapped and also which signals they were, from the context of the subgoal. It was not immediately clear in which specification they had been swapped.

10.3 The English Commentary

An English commentary was given with the behavioural specification of each module. Such a commentary is useful as it gives a brief, if not precise, overview of the behaviour of the module. This is of use to a formal verifier, to help keep a mental picture of the purpose of each module. It would also be of use to engineers who are not familiar with the formal notation.

An error was discovered in the English commentary of `PRIORITY_DECODER`. It mistakenly stated that the priority decoder outputs one word per output port, when in fact it returns one word per input port. The formal specification was correct. The confusion arose because other components output one word per output port. The error was discovered during the formal verification because the commentaries were being used to construct informal arguments to guide the formal proof.

10.4 The Behavioural Specifications

Many errors were found in the behavioural specifications of the modules, though most involved the incorrect specification of word lengths. Such errors were generally easy to detect and correct.

The specification of the NOR gate was correct only for 2 input gates, but was used for gates of varying sizes. When originally specified it had been assumed that only 2 inputs would be used. This error was noted the first time an incorrect NOR gate was used.

One of the primitive word operators, `WSEG`, which selects a segment from a word, was incorrectly used. This was because the specifier had misunderstood its definition, assuming it took the end points of the segment as arguments, when it took one end point and a length. Again this was spotted in the first proof where it was used.

The timing of several modules was incorrectly specified. For example, in the specification for the module `TIMING`, an event was stated to occur at the same time as the frame start signal when it actually occurred on the subsequent cycle. Such errors were normally easy to detect and correct. Goals of the form `(ts = ts + 1)` were obtained. In fact in the specification for the upper modules, the timing was purposely worked out in this way. Educated guesses were made about when events occurred and used in the specification. The actual values were then discovered during the proof and corrections made. This was quicker than attempting to get the timing right from the start.

It was assumed that the two bits of the grant signal for an output port were sampled at the same time in the specifications for the dataswitch to determine which input was successful. In fact the implementation samples them at different times. This meant that the number of the successful input was not just a function of the grant signal at a time, but depended on its values at two consecutive times.

When initially writing the specifications it was assumed that the same definition of a time frame between successive frame start signals could be used for all modules. However, the frame start signal is passed to all modules with no delay, whereas other signals suffer delays at various points in the circuit. In particular the active signal is delayed at several points. This means that the definition of a frame must vary between modules to account for the different relative times of the frame start and the active signal arriving at a module.

In several modules, the structure of the signals output was confused in a similar way to the error in the English commentary mentioned earlier.

Some of the specifications contained redundant information that either was not required for the proof, or was essentially stated twice within the specification. Whilst this did not effect the proofs for the modules in question, having unduly complex specifications would have complicated the proofs of the upper levels.

The specification of the arbiter did not specify its behaviour on the last cycle of the frame in the case when no cells arrived. This was discovered because a subgoal had to be proved about the value of the grant signal at this time, but no information was available.

The initial specifications for the upper modules in the hierarchy did not consider empty frames as a special case. It was believed that empty frames were covered in the behaviour given. However, this was not so. Consequentially an extra case needed to be added.

In several places the expression `SBIT k` which selects the k-th bit of a signal was used when what was needed was `($= k o BNVAL)` which converts the word to a number and then tests if it was equal to k. This arose due to confusion over the form in which the data was being stored on the outputs of those modules.

10.5 The Correctness Statement

An additional assumption needed to be added to the correctness statements of some modules. This concerned the effect of the active signal arriving close to the frame start signal. It had initially been thought that the switching element would function correctly irrespective of when the active signal arrived. This was not so. An assumption stating that the active signal did not arrive at an inopportune moment was needed. This assumption appeared in various forms for various modules as well as in the full correctness statement for the element. The fabric has no control over when these signals arrive as they are determined by the external environment. The design of the port controllers must ensure that the assumption is upheld. The assumption could have been included in the specification of the modules concerned rather than being explicitly in the correctness theorem. This would have made little difference to the proofs.

10.6 Overview of Errors

Many errors were found in the original formal specifications. This highlights the fact that specifications can be just as hard to get right as implementations. This is eased by using an expressive specification language such as higher-order logic. However many errors still occur.

The large number of errors in the original formal specifications of the switching element are due to the fact that the specifier was not originally familiar with the designs being specified and very little informal documentation was available. The specifications would probably have contained fewer errors if written by the designers during the design process, or if they had produced informal documentation for each module. Errors corresponding to those found in the behavioural specification could just as easily have been in the structural specification. The formal verification would have found such errors in the same way.

The exercise does illustrate how well formal verification can discover errors and ensure that their correction does not introduce new errors, whether they are in the implementation or specification.

Many errors concerned the sizes of words. These might have been discovered earlier if the sizes of the words could have been included in the type information – dependent typing. Then some of the errors might have been discovered during type-checking. Some systems such as VERITAS [9] and Nuprl [15] have this ability, though HOL does not.

Several errors arose due to the different representation of the request information in the headers of cells used by different modules. For example, initially the request information has the form of two bits per input port giving the binary version of the output port number. This information appears as 4 bits per input port. Each bit is then a flag indicating whether a particular output port is making a request for an input. Elsewhere it is in a similar format except with 4 bits per output port with flags indicating whether an input port is making a request for the output. Furthermore, in the top level behavioural specification, ports are also referred to by natural numbers since this is a higher level view. With more complete informal documentation of the implementation, such errors would have been less likely.

11 Design for Verifiability

The element was not designed with verification in mind. Despite this we did not need to change its implementation to complete the verification. However, we did change the description of the design; adding extra layers of hierarchy for example. These changes made the verification much more tractable. Had the element been designed with verification in mind, these changes could have been incorporated into the original description, thus simplifying the verification task from the outset. More complete informal documentation of each module could also have been produced which would have significantly reduced the time spent writing the formal specifications.

The verification suggested ways that the implementation could have been changed that would have simplified the verification. In particular, with a small amount of additional logic the assumption that the frame start and cell headers do not arrive close together would not have been needed. As the element was never intended to be used in this situation, and was not originally intended to be verified, it was perfectly reasonable for the designers not to add the extra logic. However, had it been included, the verification would have been simplified. Because the behaviour would have been more uniform, the formal behavioural specifications of several modules would have been simpler. Less time would have been spent determining what the intended behaviour was. The proofs would also have been simpler and easier to produce.

Making the change would have had other advantages too. The re-usability of the design would have been increased. Designers of switches which used the element would not need to worry about invalidating the assumption. Furthermore, the design would be more fault tolerant. A rogue short cell created by a transient or design error in a port controller would cause less damage.

It would not have been a failure for formal methods if the change had been made just to simplify the verification. Design for testability is a well respected methodology. Design for verifiability should be similarly respected.

12 Conclusions

We have demonstrated that a fully machine-checked formal verification of a real piece of communications hardware can be conducted in a time scale comparable to that required for its original design, implementation and informal testing. This was despite the formal specification and verification only starting after the design had been implemented, the documentation being sketchy and the verifier having no previous knowledge of the design or its application.

Whilst no errors were found in the implementation, problems were found in the original versions of the formal specifications. These were corrected and the verification completed. We now have a rigorous description of the behaviour of the element, and of all its constituent modules. Having formally verified the implementation against it, we can have a high degree of confidence that the element does have that behaviour, provided it was correctly fabricated from the HDL description.

The formal specification provides a rigorous description of the behaviour of the element, and in particular its timing behaviour. This will be of use if changes are made to the design, and when interfacing the fabric to other components of the switch. Having formally verified that it corresponds to the current implementation means that it can be assumed to describe the behaviour of the element with a high degree of confidence.

We verified all modules down to the gate level. This was done to illustrate the feasibility of such a complete formal verification. However, the hierarchical method allows a more pragmatic approach to be taken if desired. Modules that are simple enough to be exhaustively simulated, or for which there is already a high degree of confidence for other reasons do not need to be formally verified. They can be taken as being basic modules, as was done with the specifications of the logic gates. Their formal specifications are then assumed to be correct. The formal verification of the rest can be carried out as normal. This allows more effort to be expended on the verification of modules where errors are thought most likely to occur. However, if this approach is taken, it must be remembered that it is not only the implementation of the module that must be correct. The formal specification must be an accurate description of it. As the errors discovered in our formal specifications illustrate, errors can just as easily be made in specifications as in implementations.

Traditional networks are designed to throw away data when congestion occurs. Thus all that can be said about a network is that transmitted cells may or may not arrive some time later. Complex and time-consuming protocols must then be employed to provide a reliable service to the end-user. This may be unacceptable in a safety critical and possibly real-time situation. However, it need not occur with an ATM network. One of the many advantages of ATM over traditional transfer modes is the ability to provide quality of service guarantees. That is, users of the network can request a certain amount of bandwidth. The network grants the request only if suitable resources are available. The appropriate bandwidth is then reserved. Critical applications can be allocated sufficient resources in advance. If the resources are not available, this is discovered before the problem occurs rather than after cells have been discarded. It may then be possible for alternative arrangements to be made. It is thus theoretically possible for an ATM network to be *designed* not to lose cells from the outset. In addition, large improvements have been made in the reliability of transmission links, so transmission errors are increasingly rare. Thus it may be possible for an ATM network of the future to be designed to meet even the high standards required for safety-critical applications. Even if the network is designed so that cell loss should not happen, errors in the implementation could still cause this to occur in reality. If no cell loss is desired, stringent validation is required. Thus techniques such as formal verification would be vital.

13 Further Work

The behavioural specification used in the formal verification was a byte-clock level description. That is, cells are perceived to consist of a series of bytes. A more abstract description would be based on the frame start clock. A cell would then be perceived as a single atomic entity which could be switched in one high level cycle. In this way the byte-clock timing details could be abstracted away from. Such a description could be formal verified against the byte-level specification. Since we have verified the implementation against the byte-level description we could then conclude that the implementation implements the high level specification.

We have suggested that given we now possess a machine-checked proof of an implementation of the switching element it should be possible to track further evolution of the design within a similar time scale to that required to make the changes. A demonstration of this claim is required.

We have verified the implementation of one part of the Fairisle switch: the switching element. The next stages in the wider project to verify an ATM network are to verify a larger switching fabric made of elements and to verify the port controllers of the switch.

Acknowledgements

This work was supported by SERC grant GR/J11133. I would like to thank Mike Gordon and Ian Leslie for their help and advice and for providing valuable comments on an earlier draft of this paper. I am also grateful to the members of the Automated Reasoning Group in Cambridge for proving a stimulating research environment. I am particularly indebted to Wai Wong and Brian Graham for their constant willingness to help solve various problems that I encountered.

References

[1] R. Boulton, M. Gordon, J. Herbert, and J. Van Tassel. The HOL verification of ELLA designs. In *Proceedings of the International Workshop on Formal Methods in VLSI Design*, 1991.

[2] Richard Boulton, Andrew Gordon, Mike Gordon, John Harrison, John Herbert, and John Van-Tassel. Experience with embedding hardware description languages in HOL. In V. Stavridou, T. F. Melham, and R. T. Boute, editors, *Theorem Provers in Circuit Design*, pages 129–156. North-Holland, 1992.

[3] Richard J. Boulton. A HOL semantics for a subset of ELLA. Technical Report 254, University of Cambridge, Computer Laboratory, April 1992.

[4] Avra Cohn. The notion of proof in hardware verification. *Journal of Automated Reasoning 5*, (5):127–138, 1989.

[5] Paul Curzon. The formal verification of the Fairisle ATM switching element. Technical Report 329, University of Cambridge Computer Laboratory, March 1994.

[6] K. Edgcombe. The Qudos quick chip user guide. Qudos Limited.

[7] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[8] 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.

[9] F.K. Hanna and N. Daeche. Dependent types and formal synthesis. In C. A. R. Hoare and M. J. C. Gordon, editors, *Mechanized Reasoning and Hardware Design*, pages 49–68. Prentice Hall, 1992.

[10] He Jifeng, I. Page, and J.P. Bowen. Towards a provably correct hardware implementation of Occam. In G.J. Milne and L. Pierre, editors, *Correct Hardware Design and Verification Methods (CHARME'93)*, volume 683 of *Lecture Notes in Computer Science*, pages 214–225. Springer-Verlag, 1993.

[11] 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.

[12] J.M.J. Herbert and M.J.C. Gordon. Formal hardware verification methodology and its application to a network interface chip. *IEE Proceedings Part E, Computers and Digital Techniques*, 133(E-5), 1986. Special issue on Digital Design Verification.

[13] John Herbert. Temporal abstraction of digital designs. Technical Report 122, University of Cambridge, Computer Laboratory, February 1988.

[14] Mark B. Josephs, Rudolph H. Mak, Jan Tijmen Udding, Tom Verhoeff, and Jelo T. Yantchev. High level design of an asynchronous packet-routing chip. In J. Staunstrup and R. Sharp, editors, *Proceedings of the Second IFIP Workshop on Designing Correct Circuits*, pages 261–274, January 1992.

[15] M. Leeser. Using Nuprl for the verification and synthesis of hardware. In C.A.R. Hoare and M.J.C. Gordon, editors, *Mechanized Reasoning and Hardware Design*, pages 49–68. Prentice Hall, 1992.

[16] I.M. Leslie and D.R. McAuley. Fairisle: An ATM network for the local area. *ACM Communication Review*, 19(4), September 1991.

[17] T.F. Melham. *Higher Order Logic and Hardware Verification*, volume 31 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1993.

[18] P.G. Neumann and contributors. Risks to the public in computers and related systems. *Software Engineering Notes*.

[19] Wai Wong. Modelling bit vectors in HOL: the word library. In *Proceedings of the 1993 International Workshop on Higher Order Logic Theorem Proving and its Applications*, 1993.

[20] Wai Wong. Recording HOL proofs. Technical Report 306, University of Cambridge Computer Laboratory, July 1993.