Problems Encountered in the Machine-assisted Proof of Hardware*

Paul Curzon

University of Cambridge Computer Laboratory,
Pembroke Street, Cambridge, UK.
Email: pc@cl.cam.ac.uk
URL http://www.cl.cam.ac.uk/users/pc/

Abstract. We describe our experiences verifying real communications hardware using machine-assisted proof. In particular we reflect on the errors found, problems encountered and the bottlenecks that slowed the progress of the proofs. We also note techniques which would alleviate the problems. Most of the problems we discuss only become significant when large designs are verified.

1 Introduction

Descriptions of formal verification projects invariably focus on the successes. However, much can also be learned from the things that slow progress. In this paper we reflect on the problems encountered in the verification of real communications hardware: the Fairisle Asynchronous Transfer Mode (ATM) switching fabrics [7]. Fairisle is an existing network, designed by the Systems Research Group in Cambridge. It was designed as a platform for research into multimedia and management issues of ATM networks, and carries real user data. The switching fabrics that we considered contain both control and data paths. They form the heart of the Fairisle communications network's switches. Higher-order logic versions of Qudos HDL descriptions of the implementation were verified. The verifications consisted of proving that logic gate level implementations of the devices satisfied specifications at the timing diagram level: describing both their timing and functional behaviour. They were carried out using the HOL90 proof assistant [5].

2 The Fairisle 16 by 16 Switching Fabric

The Fairisle switch (see Fig. 1) consists of three types of component: input port controllers, output port controllers and a switching fabric. Each port controller is connected to a transmission line and to the switching fabric. The port controllers synchronise and process incoming and outgoing cells of data, appending control information to the front of the cells in a *routeing byte*. A cell consists of

* In *Correct Hardware Design and Verification Methods*, Eds. Paolo E Camurati and Hans Eveking, LNCS 987, pp 56-70, Springer-Verlag, 1995.

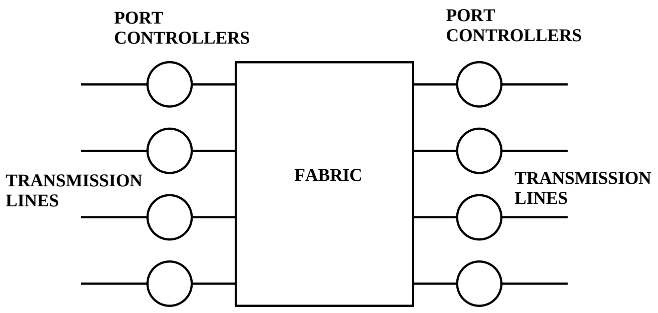

Fig. 1. The Fairisle Switch

a fixed number of bytes of data which arrive one at a time. The fabric switches cells from input port controllers to output ones according to the routeing byte. If different port controllers inject cells destined for the same output port controller (as indicated by the routeing bytes) into the fabric at the same time, then only one will succeed. The others must retry later. The routeing byte also includes priority information which is used by the fabric when arbitrating clashes. Arbitration takes place in two stages. High priority cells are 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, and 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 pass through the fabric.

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 data cells into the fabric synchronously so that the routeing bytes arrive at the same time. The behaviour of the switching fabric is cyclic. In each cycle or *frame*, it waits for cells to arrive, reads them in, processes them, sends successful ones to the appropriate output ports and sends acknowledgements. It then waits for the next round of cells to arrive. The cells from all the input ports start when a particular bit of any one of them goes high.

The inputs to the fabric consist of the data lines which carry the cells, the acknowledgements that pass in the reverse direction, and the frame start signal which is the only external control signal. The outputs consist of the switched data and the switched and modified acknowledgement signals.

The switching fabric consists of a series of switching elements connected in a regular array. The simplest (4 by 4) switching fabric consists of a single element which connects 4 input ports to 4 output ports. To make larger fabrics, several elements can be connected together in a regular array. A 16 by 16 fabric can be made from 8 elements in two rows connected as a delta network as shown in Fig. 2. However, the design of the elements used for the front and back rows differ in several ways both from each other and from the original. For example the 16 by 16 fabric uses more control information in the routeing byte, so the elements differ to cope with the extra information. The front elements must not remove the routeing information as it is needed by the back elements. There are also several aspects of the timing which differ between the different elements. For example the frame start signal is delayed on entry to the 16 by 16 fabric elements to allow the port controllers extra processing time.

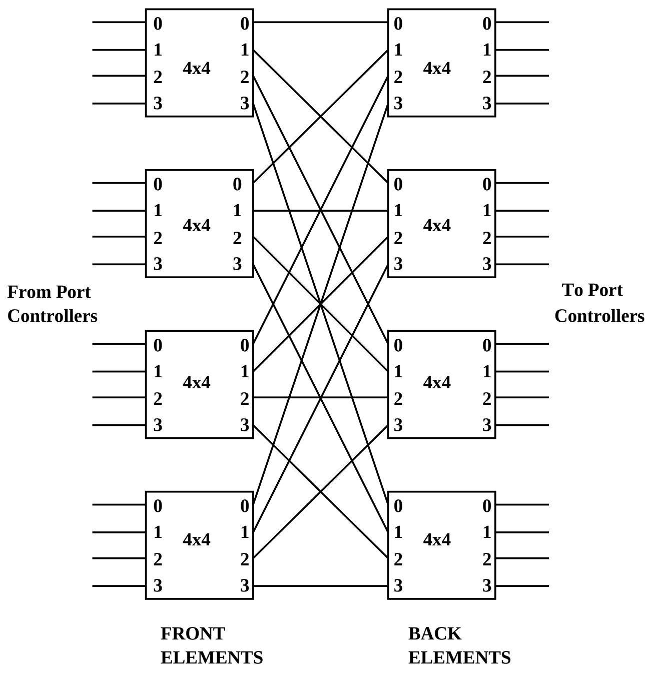

Fig. 2. The 16x16 Switching Fabric

The fabrics were designed using a Hardware Description Language: Qudos HDL. This is a simple HDL which allows the structure of hardware to be specified. It does not allow behaviour to be specified directly. These descriptions were translated into very similar higher-order logic descriptions for the verification. Extra features of HOL were used to simplify the descriptions. This was not intended to change the underlying implementation. The fabricated elements were implemented on 4200 gate-equivalent Xilinx programmable gate arrays.

3 The HOL system

The formal verification was conducted using the HOL90 proof assistant: an LCF style theorem prover for higher-order logic [5]. A user of the HOL system gives proof commands in the form of Standard ML (SML) function calls. Proofs are developed interactively, with a record of the proof script (ie SML program) being kept in a separate file. This file can be rerun later in batch mode to recreate the theorems. Proofs are structured into theories. Each theory has an associated theory file which stores details of the theorems and definitions (though not proofs) of the theory. This allows the theorems to be accessed in the future without being reproved from the proof scripts.

The HOL system is fully expansive. This means that all proof commands ultimately must call sequences of the SML functions corresponding to the primitive inference rules and axioms of the logic in order to create theorems. This is enforced by the type system since the theorem type is an abstract type. Coupled with the fact that proofs are just SML programs, this means that the user can freely develop their own proof tools, perhaps specific to the theories developed. The tool writer has complete assurance that such tools will not compromise the system. The worst a tool can do is either raise an exception or prove the wrong theorem. It cannot call something a theorem if it is not.

4 The Verification

We have been concerned with the verification of the fabric, not the port controllers. In particular, we have verified the fabricated 4 by 4 fabric and the switching elements of the 16 by 16 fabric. We are also in the process of verifying versions of the 16 by 16 fabric constructed from the verified elements. Several different versions of the elements were verified. Two back elements were verified because one was found to be erroneous when the verification of the 16 by 16 fabric using it was conducted. Additional front elements were also verified. One corresponded to an alternative way of implementing the 16 by 16 fabric (though such a design has not been fabricated). Others were used as stepping stones to aid the verification of the fabricated front element, consisting of some though not all of the design changes made to the original 4 by 4 fabric.

The verification consists of proving for each module in the design, a correctness theorem of the form:

```
⊢ Assumptions ⊃ (Implementation ⊃ Specification)
```

That is the description of the implementation implies the specification under certain assumptions on the environment. Here `Implementation` gives a structural description of the implementation: the components used, how they are wired together and which wires are inputs, outputs and local to the module. `Specification` gives the more abstract behavioural description consisting of the timing and functional behaviour of the module, usually in the form of interval temporal style operators. `Assumptions` give the conditions on the environment (ie inputs) under which the module will satisfy the specification. Details of the specifications and correctness theorems are given elsewhere [2]. Once the separate modules have been verified, their correctness theorems are combined to give a correctness theorem for the whole device.

5 Duration of the Proof Effort

The original 4 by 4 fabric was formally specified and verified in approximately 4 person-months [1]. Of this time, over half was spent understanding and specifying the design and its modules. This compares with the designers estimate that "several months" had been required to design and implement the element. Much of the verification time was spent on two of the higher modules.

Next the 4 by 4 elements of the 16 by 16 design were verified, including several designs that were not fabricated. This was completed much more quickly, each new element taking only hours or days (see Fig. 3). The speed up was partly because many modules were reused and so did not need to be reverified. However, roughly the same number of modules were reverified over all the new versions as in the original. The total time required was much less however [3]. A significant portion of the speedup came from the fact that the proofs of modified modules were obtained by adapting old proofs for the new theorems, rather than by creating new proofs from scratch. Problems related to this are discussed in Sec. 8.1.

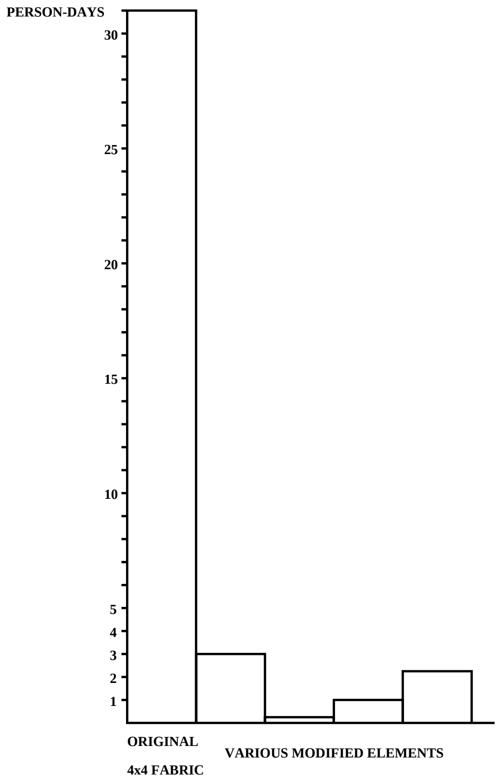

Fig. 3. Comparison of verification time for variants of the 4 by 4 fabric

Once the separate elements of the 16 by 16 design were verified, the proof that when connected together they successfully implement the 16 by 16 fabric was started. This proof is still in progress. It has taken much longer than originally expected: over 20 person days to date.

6 Errors

In this section we give an overview of the errors discovered during the formal verification process. 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. All errors found during the course of the verification were corrected and the proofs continued. The presence of errors consumed a large amount of time. Many errors were trivial to fix: for example when an incorrect word length had been given. However, the presence of large numbers of trivial errors hindered the proof attempt by disrupting the thought processes of the verifier.

6.1 The Implementation

No errors were discovered in the fabricated implementation of the original 4 by 4 switching element. This is perhaps not surprising, since the fabric has been in use for some time and 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.

An error was discovered in the original fabricated version of the back elements of the 16 by 16 fabric. The fact that the cells arrived at the back elements later than at the front elements had not been accounted for. Consequently, data was lost from the end of the cells. This error was known of by the designers, having been found when the fabricated fabric was in use. Simulation of the design had failed to discover it. The verifier had not been aware of the problem. The faulty element was successfully formally verified! The error was only found by the formal verification process when the verification of the 16 by 16 fabric constructed of faulty elements was attempted. The faulty element was itself successfully verified because its specification was reverse engineered from the implementation. This initially masked the problem. The error was corrected by adding an extra delay to the frame start signal. However, this correction introduced an "anomaly" into the design (see Sec. 7.1).

6.2 The Structural Specification

Several errors were found in the HOL structural specification. These were introduced in the translation from Qudos HDL, for example, due to the introduction of multi-level words. Had a more direct translation from Qudos to HOL been made, these errors would have been avoided. The verification task would have been made much more difficult however.

In several places the wrong number of copies of a unit was specified, due to the specifier believing that the replication operator used took as an argument an index value (as in Qudos HDL) when it actually took the number of copies. This could have been avoided if a duplication construct of HOL 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, but this information was needed in the proof.

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

The two wires of a control signal were inadvertently swapped in the description of two modules. If this had occurred in the implementation it would have resulted in some cells being sent to the wrong outputs. 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.

The original HOL descriptions of the 16 by 16 front elements did not include the extra circuitry required to prevent the routeing byte being removed. This mistake was made because the structural specifications were created by modifying those for the original element. This was faster than translating the Qudos sources. As the same mistake was made in the behavioural specifications, it was only discovered when the verified element was used in a subsequent proof.

6.3 Documentation

The original documentation for the 16 by 16 fabric specified the delay through the fabric incorrectly.

In addition to the existing documentation for the whole design, an English commentary was written by the verifier for each module's behavioural specification. They give brief, if not precise, overviews of the behaviour of the modules. An error was discovered in the English commentary of the priority decoder modules. 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.

6.4 The Behavioural Specifications

Many errors were found in the behavioural specifications of the modules. Most involved the incorrect specification of word lengths similar to those in the structural specifications. They were generally easy to detect and correct.

The timing of several modules was incorrectly specified. For example, in the specification of the timing module, 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 such as `(ts=ts+1)` were obtained.

When writing the specifications it was assumed that the two bits of the main control signal indicating the successful input port for a given output port were sampled at a single time by the dataswitch. In fact the implementation samples the two bits at consecutive times. This was discovered because a proof obligation required information about the signal at one time, when the information was only known for a different time.

It was also initially 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. This means that the definition of a frame must vary between modules to account for the different relative times of the frame start and other signals arriving at a module.

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 a signal at this time, but no information was available.

The initial specifications for the upper modules in the hierarchy did not consider frames in which no cells arrive 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 selection of the k-th bit of a signal was specified when what was needed was to test if the word was equal to k. This arose due to confusion over the form in which the data was being output from those modules.

6.5 The Correctness Statement

An additional assumption needed to be added to the correctness statements of some modules. This concerned the effect of the cells arriving close to the frame start signal. It had initially been thought that the switching element functioned correctly irrespective of when the cells arrived. This was not so. An assumption stating that the cells did not arrive at an inopportune moment was needed.

6.6 Overview of Errors

The large number of errors in the original behavioural specifications were due largely to the fact that the specifier was not originally familiar with the designs being specified and little informal documentation was available. Confusion between the different representations used for the request information by different modules was responsible for many errors.

The specifications would have contained fewer errors if written by the designers during the design process, or if they had produced informal documentation for each module. We were specifying and verifying an existing fabricated design. It would have obviously been better if the design had been implemented from the formal specifications, rather than the other way round.

Many errors were introduced in the structural specifications by the translation process. Most of these were introduced because a straight translation was not performed. Instead extra structure was added to the signals, extra modules were added, and other features of HOL were used to make the descriptions more intuitive and simplify the verification. A simple way that such translation errors could have been detected would have been to generate a netlist for the circuit from the HOL description and compare it syntactically with that generated from the Qudos description.

There were fewer errors in the structural specifications of the modified modules than in the original. This was because correct HOL descriptions were modified rather than conducting the error prone translation process. However, other errors were introduced precisely because of this. In particular, modifications made to the implementation were missed from the structural specifications. Errors of this form could be avoided if the designers used HOL (or a pretty printed version of it – ie an extended version of Qudos HDL embedded in HOL) to produce designs in the first place. This would have the additional advantage of allowing the designers to express the design more naturally. However, it would require simulation and fabrication technology, similar to that available for Qudos, to be built around HOL.

Many errors concerned the sizes of words. These might have been discovered earlier (during type checking) if the sizes of the words could have been included in the type information. Some systems such as VERITAS [6] support such dependent typing, though HOL currently does not.

The verification process did detect the error that had been missed by simulation. This in itself would have made the verification worthwhile.

7 Troublesome Design Features

A great deal of verification time was spent coping with a small number of troublesome design "features". They did not render the design incorrect since the system as a whole still fulfilled the task for which it was intended. However, there was also no overriding reason why they should have been chosen in preference to some other approach. In some cases they could be thought of as design "anomalies", either because the behaviour had to be compensated for in other parts of the design, or because an alternative would have yielded a simpler or cleaner behaviour.

7.1 Examples

An example of a troublesome design feature occurs in one of the low level modules of the arbiters of the 4 by 4 elements. It controls the disabling of outputs in response to the arbitration decision made. The module has a consistent behaviour except in the case when the frame start signal arrives at the same time as the start of a cell. In this case the behaviour depends on whether a cell arrived in the previous cycle or not. The way the elements are intended to be used is that the frame interval is constant (currently 64 cycles) and the cells arrive on a fixed (currently the 8th) cycle in the frame. In this environment the situation should never arise. However, an extra assumption was required in the correctness statement of the module since it was only true provided the two events did not occur together. Variations of this assumption propagate throughout the design. It can only be discharged at the point where the whole switch, with the fabric connected to the frame start generator and the port controllers, is verified. It thus must be dealt with in the proofs throughout the rest of the design hierarchy including modules connected to, as well as parents of, the affected module. The assumption changes form between the different modules and refers to different (internal) signals. Thus a single theorem cannot be proved stating that the situation never arises. It also becomes more complex in the higher level modules. Consequently, the proof effort required to deal with it becomes increasingly significant.

The need for this assumption could be removed by the addition of 2 extra logic gates per arbiter to the design – a total of 8 per element. This would effectively discharge the assumption immediately. The change would not be significant because the elements were fabricated on Xilinx gate arrays. As such it would actually involve no extra logic at all.

An example of a design anomaly occurring in the 16 by 16 fabric is the relative delays placed on the frame start signal in the front and back elements. This is the control signal that synchronises the timing of cell processing. It tells the elements to stop forwarding cell bytes and start waiting for the next cell to arrive. The delays on the front and back elements are different because the cells arrive at the back elements only after they have passed through the front elements. Thus the delay through the front element must be accounted for. However, the delay through the front element is 5 cycles, but the extra delay added to the frame start signal is only 3 cycles. Thus the back elements stop forwarding bytes two cycles too soon, losing the last bytes. This is not a problem because the frame cycles are of longer duration than the cell length. Consequently, the lost bytes should not contain data. However this means that the rate at which cells are forwarded through the fabric is less than it might be.

7.2 Design for provability

The troublesome features meant that special cases needed to be considered in the proof or assumptions were more complex than necessary. Furthermore, the design was more difficult to understand, and thus the informal proof harder to construct. The designer accepted that the changes identified could have been made without adverse effects to the design.

Problem features in low level modules of the design can have an inordinate effect on the whole proof, especially if they complicate assumptions. This is because assumptions often need to be discharged during the verification of all components up the design hierarchy which are dependent on the anomalous component. At higher levels the assumptions can change form as the signals involved are manipulated by other hardware. Extra work may be required to validate that the outputs of connected components do fulfil the conditions of the assumptions.

Most of the design changes suggested as a result of the verification were completely acceptable to the designer. This suggests that it is possible for designers to design with provability in mind, simplifying the proof obligations without seriously affecting the designer's freedom. This idea is similar to ideas suggested on the CHARME project in which fixed design styles could be used to improve verifiability. It is not in general clear how particular design decisions will affect the proof until it is underway. However, our experiences suggest that big improvements can be obtained by concentrating on the assumptions of each module. They should be avoided or simplified wherever possible.

8 Problems concerned with Machine-Assisted Proof

In this section we discuss aspects of the use of machine-assisted proof which hampered the proof effort.

8.1 Proof Modification

There are many situations where the whole or part of a previously developed proof script can be used to prove a new theorem with only small modifications. By modifying old proofs, new proofs can be developed much more quickly than if they are created from scratch. Furthermore, proofs must often be reengineered in this way when errors are found. In the Fairisle verification proof reuse was used extensively. However, several problems arose.

Reusing Proof Scripts One problem was that the available proof tools were designed for proof creation rather than proof reuse. The way we reused proofs was to replay the script on the new goal. Invariably, at some point the proof would fail. The point where a change was needed was determined by examining the final proof state if there was one, or by single stepping through the proof, examining the intermediate proof steps. Single stepping was necessary because the proof often failed long after the point where a change needed to be made. Single stepping was performed manually by cutting and pasting fragments of the proof from the source file into the HOL session. Once the appropriate change was made the proof was rerun until a further problem arose. Clearly a single step debugger for proof scripts was really needed. This is being addressed in the latest version of the TkHolWorkbench graphical user interface for HOL [8].

Tactics could also be better designed to help proof reuse. For example, they could give a better indication of where the proof first went awry. A common early indication of a problem is that the goal has not been changed by an applied tactic. Thus the location where changes are needed would be easier to find if tactics failed in this situation. Small changes in the design of tools like this can have large effects on the ease of reengineering a proof script.

Proof Maintenance If proof scripts are reused, then their maintenance is an important issue. It is not sufficient, simply to hack out a proof, obtaining the theorem in whatever way is possible. The final script should be in a state suitable for modification and reuse. However, proof creation is often experimental in nature with much backtracking. This is not conducive to the creation of maintainable proofs. Neither is, the act of repeatedly modifying old proofs since this is likely to degrade their reusability.

For small projects this is not a problem. The scripts can be quickly and simply tidied, and the consequences of a certain amount of untidiness are small. However, on larger projects such as the Fairisle verification it becomes a problem. Individual proofs, possibly consisting of many lemmas can take days or weeks to develop and can be very large. The untidied script developed for the top level of the 16 by 16 fabric was over 6000 lines long, for example. If the tidying is left to the end of the development in this situation it becomes a formidable task. It is unlikely to be done well if at all and will be time consuming. Furthermore it hampers the continuing development of the script itself. On the other hand, if the proof of every lemma is tidied as it is created, the thought processes of the verifier are liable to be disrupted. Also, proofs will be tidied only to be later discarded.

The best solution, where possible, is for proof management tools to prevent the proof script getting into an untidy state in the first place. HOL is currently lacking in such tools. A particular problem of this kind that we encountered was the lack of tools to allow multiple theories to be worked on within a session. This is useful because it is often discovered mid-proof that a lemma is needed that would be best placed in some other theory. HOL provides only limited facilities to do this. Consequently, in the Fairisle verification everything was initially developed in a single working theory that was regularly tidied by moving lemmas to appropriate theories. This can be avoided by providing a more sophisticated theory manager which allows lemmas to be placed directly into the most appropriate theory [4].

Comprehending Proofs Even when high level tactics are used, proof scripts are difficult to understand, and thus to reengineer. Clearly it is important that informal proofs are given with, or are derivable from, proof scripts. Tools that allow proofs to be stepped through interactively are also clearly important. Such tools are under development for HOL.

8.2 Finding Things

The full proof scripts for the Fairisle proofs are very large, running to hundreds of pages of A4 text. As a consequence finding definitions, previously proved lemmas, theorems and tools became problematic. The TkHolWorkbench graphical user interface for HOL [8], currently being developed, is likely to ease this problem in the future, providing a more sophisticated graphical interface for navigation around theories. The recognition that proof script reuse is an important way of creating new proofs means that the ability to find suitable old proofs becomes an important consideration. It would be useful if tools could be developed to help find and suggest suitable proofs.

8.3 Losing Sight of the Proof

A problem frequently encountered was that the verifier became bogged down in the details of the proof and consequently lost sight of the informal high level reasons for the design's correctness. This is a common problem with HOL: because of the relatively low level that proofs must currently be performed it is easy to get engrossed in proving small details and miss the fact that the wrong theorem is being proven. The verifier must make a constant effort to keep the outline of the proof in sight. Ideally an informal proof sketch would be written first, and then followed. However, often the easiest way to determine the informal proof is to experiment with the machine-checked proof: a theorem prover can be a powerful tool to help understand a design and develop an informal proof.

8.4 Generic Theorems

Much time can be saved in a proof if definitions and theorems can be made generic, such as considering an n-bit adder rather than separate adders for different word lengths. This approach has perhaps been most successful in the area of microprocessor verification [9]. Generic specification is not a universal panacea however. It requires appropriate regularity in the design and for there to be a need for design variations which exploit this regularity. It can impose overly rigid constraints on the designer. We adopted it for some modules of the Fairisle designs, though the more flexible proof reuse was generally more useful.

There was one case where a generic proof was possible but was not done. Had it been, much time would have been saved. It concerned the delay on the frame start signal to the switching elements, mentioned earlier. There was no delay on the original 4 by 4 fabric verified, and delays of 5 and 8 time units for various versions of the elements for the 16 by 16 fabric. Furthermore, one of the design modifications suggested by the verification process for the 16 by 16 fabric was to change the 8 cycle delay to a 10 cycle delay.

Each version was verified separately by reengineering earlier proofs (in very minor ways), rather than verifying a single generic version. The latter would in retrospect, have saved time in the long term. This was not done initially because the first design verified had no delay at all, and it was not originally intended to verify the other versions. The next elements verified were 5 cycle delay versions of the front and back elements of the 16 by 16 fabric. Since at this stage it was believed that only 5 cycle delay versions were required, it was decided that the extra effort required to create a generic version was not warranted given the time constraints. However, the back element design was incorrect, and so the correct 8 cycle version then needed to be verified. At this point, there was even more time pressure to get the correct version verified, so a generic version was still not created.

The problem was that in the short term it was always quicker to verify specific versions. Furthermore, the need for a generic design was not foreseen. Interestingly, some other aspects of the design *were* made generic, where this added little extra cost, but the flexibility was never exploited as it was not needed in any of the design variations verified. It is clearly important that the verifiers are aware of the kind of ways that a design might vary so this can be utilised from the outset.

9 Conclusions

We have outlined the errors discovered and described the problems encountered when using a proof assistant for the formal verification attempt of a relatively large hardware design. Many of the problems encountered were specific to the use of a proof assistant, and would not have arisen had a fully automated theorem prover been used. Clearly, trusted tools that can verify designs fully automatically are more desirable than a proof assistant. However, with current technology, for large designs this is often infeasible. For example, model checking approaches often require that the description of the design used is simplified such as by reducing word lengths. However, this can lead to errors being missed. The inadvertent swapping of the two control signals, and the assumption that the two bits of the control signal were sampled at the same rather than consecutive times would have been missed had only a single control bit been modelled. There is a growing movement towards combining specialised automatic tools with more general proof assistants. For example, it would have been sensible to verify the lower level modules and simpler lemmas using a BDD tool, leaving only the upper levels to machine-assisted proof. If this approach is taken the problems we have encountered will still need to be overcome, however.

The detection and correction of errors consumed a large proportion of verification time. Most of the errors could have been found more easily by other means, or prevented from occurring in the first place. Formal proof should be used to find obscure bugs and sort out subtleties in the understanding of why the design is believed to be correct. For such problems formal verification may be the only solution. Debugging of specifications, designs and implementations by traditional methods such as testing, code walk-throughs, comparison of netlists and even model checking where appropriate should be conducted first. The formal verification task will then be much simpler.

Designer's can ease the verification task without compromising other design considerations. Our work suggests that one way this can be done is by ensuring that the operating assumptions of modules are as few and as simple as possible. It thus can be done early in the design cycle. The development of design constraints for formal verification would be useful.

As proof scripts increase in size, maintainability and reusability become important considerations. The creation of a proof script should be an engineering discipline, following software engineering principles. Proof tools should ideally prevent scripts from getting into an untidy state in the first place.

Acknowledgements

This work was supported by EPSRC grants GR/J11133 and GR/G23654. I am grateful to Mike Gordon, Ian Leslie, Donald Syme and the members of the Automated Reasoning Group in Cambridge for their support, and to the anonymous referees for their comments.

References

1. Paul Curzon. Experiences formally verifying a network component. In *Proceedings of the 9th Annual IEEE Conference on Computer Assurance*, pages 183–193. IEEE Press, 1994.
2. Paul Curzon. The formal verification of the Fairisle ATM switching element. Technical Report 329, University of Cambridge Computer Laboratory, March 1994.
3. Paul Curzon. Tracking design changes with formal machine-checked proof. *The Computer Journal*, 38(2):91–100, 1995.
4. Paul Curzon. Virtual theories. In E. Thomas Schubert, Phillip J. Windley, and James Alves-Foss, editors, *Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, volume 971 of *Lecture Notes in Computer Science*, pages 138–153. Springer-Verlag, 1995.
5. M.J.C. Gordon and T.F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic.* Cambridge University Press, 1993.
6. 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.
7. I.M. Leslie and D.R. McAuley. Fairisle: An ATM network for the local area. *ACM Communication Review*, 19(4):327–336, September 1991.
8. Donald Syme. A new interface for HOL - ideas, issues and implementation. In *Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, Lecture Notes in Computer Science. Springer-Verlag, 1995.
9. P.J. Windley, K. Levitt, and G.C. Cohen. The formal verification of generic interpreters. Technical Report 4403, NASA Contractor Report, October 1991.