Experiments in Automating Hardware Verification using Inductive Proof Planning

Francisco J. Cantu* Alan Bundy** Alan Smaill† David Basin‡

September 23, 1996

Abstract

We present a new approach to automating the verification of hardware designs based on planning techniques. A database of methods is developed that combines tactics, which construct proofs, using specifications of their behaviour. Given a verification problem, a planner uses the method database to build automatically a specialised tactic to solve the given problem. User interaction is limited to specifying circuits and their properties and, in some cases, suggesting lemmas. We have implemented our work in an extension of the *Clam* proof planning system. We report on this and its application to verifying a variety of combinational and synchronous sequential circuits including a parameterised multiplier design and a simple computer microprocessor.

1 Introduction

Confidence in the correctness of hardware designs may be increased by formal verification of the designs against specifications of their desired behaviour. Although this is common knowledge, formal verification is almost completely neglected by industry; what interest there is centres on 'push-button' systems based on model checking, where weak decidable logics are used for problem specification. But such systems are necessarily limited and cannot be applied to many important classes of problems. For example, parameterised designs and their specifications can almost never be expressed. However, systems based on more expressive logics are often resisted because proof construction is no longer fully automated and most circuit designers have neither the time, the patience, nor the expertise for semi-interactive theorem proving.

To increase the acceptance of formal methods in domains with undecidable verification problems, we must automate proof construction as much as is practically possible. The problem to overcome is the large search space for proofs. Even when semi-decision procedures like resolution are available, completely automated theorem proving is not viable because such techniques are too general and cannot exploit structure in the problem domain to restrict search. Heuristics are called for, perhaps augmented with user interaction to overcome incompleteness. One example of this is the system *Nqthm*,

*Center for Artificial Intelligence, ITESM, Mexico. Supported by CONACYT grant 500100-5-3533A
†Department of Artificial Intelligence, University of Edinburgh, UK. Supported by EPSRC grant GR/J/80702
‡Max-Planck-Institut für Informatik, Saarbrücken, Germany

which uses a fixed set of heuristics to automate the construction of proofs by induction. Proof construction is automated but sometimes the user must interrupt the prover and suggest lemmas to stop it from exploring an unsuccessful branch. A second example is embodied by tactic-based proof development systems like *HOL*, *Lambda*, *PVS*, and *Nuprl*, where users themselves raise the level of automation by writing tactics (programs that build proofs using primitive inference rules) for particular problem domains. Here incompleteness is addressed by interactive proof construction: rather than writing a 'super-tactic' which works in all cases, users interactively combine tactics to solve the problem at hand and directly provide heuristics.

We present a new approach to hardware verification that falls in between the two approaches above and offers automation comparable to systems like *Nqthm* but with increased flexibility since one can write new domain specific proof procedures as in the tactic approach. We have designed a set of tactics for constructing hardware proofs by mathematical induction. These tactics include tactics developed previously at Edinburgh for automating induction proofs [7] augmented with tactics specifically designed for hardware verification. Rather than having the user apply these tactics interactively, we use ideas from planning to automate proof construction. Each tactic is paired with a *method* which is a declarative specification of its behaviour. Given a verification problem, a planner uses the method database to build automatically a specialised tactic to solve the given problem. User interaction is mainly limited to specifying circuits and their properties. As in the case of *Nqthm*, additional lemmas are occasionally needed to overcome incompleteness, but this is rare; see section 4 for statistics on this and section 5 for a comparison to *Nqthm*.

We have implemented our approach to planning in an extension of the *Clam* proof planning system [8]. We report on this and its application to a variety of combinational and synchronous sequential circuits including a parameterised multiplier design and a simple computer microprocessor. Although we haven't attempted the verification of commercial hardware systems yet, we provide evidence that this approach is a viable way of significantly automating hardware verification using tactic-based theorem provers, that it scales well, and that it compares favourably with both systems based on powerful sets of hardwired heuristics like *Nqthm* and tactic-based approaches. We believe that it offers the best of both worlds: one can write tactics that express strategies for building proofs for families of hardware designs and the planner integrates them in a general proof search procedure. This provides an 'open architecture' in which one may introduce heuristics to reduce search in a principled way. This also suggests a possible methodology for designing provers for verification domains like hardware where experts design domain specific tactics (and methods) and systems such as ours help less skilled users construct correctness proofs.

The remainder of this paper is organised as follows. In section 2 we describe proof planning and extensions required for hardware verification. In section 3 we present our methodology for hardware verification and illustrate it with two examples: the verification of a parallel array multiplier and a simple computer design proposed by Mike Gordon (and hence called the *Gordon computer*). Results of experiments in applying proof planning to verify other combinational and synchronous sequential circuits are reported in section 4. In section 5 we compare some of our results with approaches used in systems such as *Nqthm, PVS* and *HOL*. Finally, in section 6, we summarise advantages and limitations of our approach and discuss future work.

2 Verification Based on Proof Planning

We begin by reviewing proof planning and methods for induction. These two parts contain only a summary of material that has been published in detail elsewhere: see [4, 7] for more on proof planning and [6, 1] for details on rippling. After, we discuss extensions required to apply *Clam* to hardware verification.

2.1 Proof Plans

Proof planning is a meta-level reasoning technique for the global control of search in automatic theorem proving. A proof plan captures the common patterns of reasoning in a family of similar proofs and is used to guide the search for new proofs in the family. Proof planning combines two standard approaches to automated reasoning: the use of tactics and the use of meta-level control. The meta-level control is used to build large complex tactics from simpler ones and also abstracts the proof, highlighting its structure and the key steps.

The main component of proof planning is a collection of methods. A *method* is a specification of a tactic and consists of an input formula (a sequent), preconditions, output formulae, postconditions, and a tactic. A method is *applicable* if the goal to be proved matches the input formula of the method and the method's preconditions hold. The preconditions, formulated in a meta-logic, specify syntactic properties of the input formula. Using the input formula and preconditions of a method, proof planning can predict if a particular tactic will be applicable without actually running it. The output formulae (there may be none) determine the new subgoals generated by the method and give a schematic description of the formulae resulting from the tactic application. The postconditions specify further syntactic properties of these formulae. For each method there is a corresponding general-purpose tactic associated to that method. Methods can be combined at the meta-level in the same way tactics are combined using tacticals.

The process of reasoning about and combining methods is called *proof planning*. When planning is successful it produces a tree of methods, called a *proof plan*. A proof plan yields a composite tactic, which is built from the tactics associated with each method, custom designed to prove the current conjecture. Proof plan construction involves search. However, the planning search space is typically many orders of magnitude smaller than the object-level search space. One reason is that the heuristics represented in the preconditions of the methods ensure that backtracking during planning is rare. Another reason is that the particular methods used have preconditions which strongly restrict search, though in certain domains they are very successful in constructing proofs. There is of course a price to pay: the planning system is incomplete. However, this has not proved a serious limitation of the proof planning approach in general [7] nor in our work where proof plans were found for all experiments we tried.

The plan formation system upon which our work is built is called *Clam*. Methods in *Clam* specify tactics which build proofs for a theorem proving system called *Oyster*, which implements a type theory similar to *Nuprl's*.

Methods in *Clam* are ordered and stored in a database, c.f. figure 1. Each of the names in the left-hand side is a method and the indentation of names represents the nesting of methods (a method can call subroutines called sub-methods, which include methods themselves), i.e. an inner method is a sub-method of the one immediately

Methods:

1. symbolic_evaluation	Simplifies symbolic expression
a. elementary	Tautology checker
b. eval_definition	Unfold definition
c. reduction	Apply reduction rules
d. term_cancellation	Cancel-out additive terms in equation
2. induction_strategy	Applies induction strategy
a. induction	Selects induction scheme and induction variables
b. base_case	Applies symbolic_evaluation to the base case
c. step_case	Applies ripple and fertilise to the step case
i. ripple	Applies rippling heuristic
- wave	Applies wave-rules
ii. fertilise	Simplifies induction conclusion with induction hypothesis
3. generalise	Generalises common term in both sides of equation
4. bool_cases	Does a case split on Boolean variable
5. difference_match	Matches and annotates differences on two expressions
6. apply_lemma	Applies existing lemma to current subgoal

Figure 1: Method Database

outside it. Given a goal, *Clam* tries the methods in order. If an applicable method is found then the output yields new subgoals and this process is applied recursively. If none is applicable, then *Clam* backtracks to a previous goal or reports failure at the top level. A number of search strategies exist for forming plans. These include depth-first, breadth-first, iterative deepening, and best-first search. We have conducted our experiments in hardware verification using the depth-first planner.

2.2 Induction

A number of methods have been developed in *Clam* for inductive theorem proving and we used these extensively to prove theorems about parameterised hardware designs. Induction is particularly difficult to automate as there are a number of search control problems including selection of an induction rule, control of simplification, possible generalisation and lemma speculation, etc. It turns out though that many induction proofs have a similar shape and a few tactics can collectively prove a large number of the standard inductive theorems.

The induction strategy is a method for applying induction and handling subsequent cases. After the application of induction, the proof is split into one or more base and step cases. The *base case* method attempts to solve the goal using simplification and propositional reasoning (with the sub-method *symbolic_evaluation*). If necessary, another induction may be applied. The *step case* method consists of two parts: rippling and fertilisation. The first part is implemented by the *rippling* method. Rippling is a kind of annotated rewriting where annotations are used to mark differences between the induction hypothesis and conclusion. Rippling applies annotated rewrite rules (called *wave-rules* which are applied with the *wave* method) which minimise these differences. Rippling is goal directed and manipulates just the differences between the induction conclusion and hypothesis while leaving their common structure preserved; this is in contrast to rewriting based on normalisation, which is used in other inductive theorem provers such as *Nqthm* [3]. Rippling also involves little search, since annotations severely restrict rewriting. The second part of the step case, fertilisation, can apply when rippling has succeeded (e.g. when the annotated differences are removed or moved 'out of the way', for example, to the root of the term). The *fertilise* method then uses the induction hypothesis to simplify the conclusion.

2.3 Extensions

The above completes our review of *Clam* and some of the methods for induction. We were able to apply *Clam* to hardware verification with fairly minor modifications and extensions.

First there were modifications to *Clam* methods which resulted, in part, from the fact that the experiments reported here are the largest that have been carried out with *Clam* and revealed some inefficiencies which required improvement. For example: the method *symbolic_evaluation* was extended with a *memoisation* procedure for efficiently computing recursive functions in verifying the Gordon computer, and with an efficient representation of the meta-logic predicate *exp_at* that finds a sub-expression in an expression; the *fertilisation* method was extended to deal with a post-rippling situation that arose in verifying a parameterised version of the n-bit adder and which has appeared in many other combinational circuits; the *generalisation* method was extended with type information to generalise over terms of type bool; the *equal* sub-method was extended to apply a more general form of rewrite equations in the hypotheses list of the sequent to simplify the current goal. We also needed to extend *Clam*'s database of induction schemas. To do this we formalised in *Oyster* new data-types appropriate for hardware such as a data-type of words, which are lists of booleans. Then we derived new induction schemas based on these which we added to *Clam*'s collection, for example induction over the length of words (i.e., a special case of list induction), simultaneous induction over two words of the same length, and induction on words where step cases are generated by increment and addition of words, and the like.

Finally, we developed a number of new methods. For example, the method *bool_cases* was added to solve goals by exhaustive case-analysis over booleans. This suffices often to automate base cases of induction proofs about parameterised designs. The method *difference_match* was added to reason about sequential circuits modelled as finite-state machines; Also, the method *term_cancellation* was added to strengthen arithmetic reasoning in *Clam*.

3 Verification Methodology

We now describe how proof planning can be used for hardware verification and afterwards provide two examples.

The user begins by giving *Clam* definitions, in the form of equations, which define the implementation of the hardware and its behavioural specification. Then, the user provides *Clam* with the conjecture to be proven. For combinational hardware this is typically an equation stating that the specification is equal to an abstract form of the implementation:

$$\forall x_1, \ldots, x_n \; spec(x_1, \ldots, x_n) = abs(imp(x_1, \ldots, x_n))$$

For synchronous sequential hardware this is typically an implication stating that if the specification is equal to an abstract form of the implementation at some time t, then the specification must be equal to an abstract form of the implementation at all time t' greater that t. If the specification and the implementation involve different time scales, then we must provide a mapping f that converts times from one scale to the other:

$$\forall t, t' : time\ \forall x_1, \ldots, x_n$$

$$spec(t, x_1, \ldots, x_n) = abs(imp(f(t), x_1, \ldots, x_n)) \wedge t \leq t'$$

$$\rightarrow$$

$$spec(t', x_1, \ldots, x_n) = abs(imp(f(t'), x_1, \ldots, x_n))$$

To create a plan, the user instructs *Clam* to find a proof plan for the conjecture by selecting one of the built-in planners, e.g., to find a plan using depth-first search. At this point *Clam* has been loaded not only with definitions and the conjecture but also the method database and wave-rules. If *Clam* finds a plan, the user may execute the tactic associated with it to construct an actual proof. Otherwise, the user can correct a bug in the theorem statement or in the definitions or suggest a lemma (which will provide new wave-rules) and try again. After *Clam* completes a proof plan, the tactic produced is executed to build an actual proof. Failure at this stage occurs rarely and happens only when a method improperly describes a tactic; in such cases, we improve the method or the tactic and plan again.

Example: A parallel array multiplier

We now describe the verification of an nm-bit parallel array multiplier. The external behaviour of the multiplier is expressed by the formula:

$$word2nat(x) \times word2nat(y)\,.$$

We represent words by lists of bits, and if x and y are words of length n and m respectively, then $word2nat$ returns the natural numbers which they represent and the above specifies their product. Multiplication of binary words can be implemented by a simple parallel array multiplier using binary additions. Consider, for example, multiplying a 3-bit word by a 2-bit word. This is represented by:

$$\begin{array}{ccccc}
 & & x_2 & x_1 & x_0 \\
 & & \times & y_1 & y_0 \\
\cline{3-5}
 & & x_2y_0 & x_1y_0 & x_0y_0 \\
 & x_2y_1 & x_1y_1 & x_0y_1 & \\
\hline
z_4 & z_3 & z_2 & z_1 & z_0
\end{array}$$

To multiply an n bit word by an m bit word the array multiplier uses $n \times m$ AND gates to compute each of these intermediate terms in parallel, and then, m binary additions are used to sum together the rows. This requires a total of $n \times m$ one-bit adders. The following equations formalise the above as a recursive description of a (parameterised) implementation:

$$\begin{aligned}
mult(x, nil) &= zeroes(length(x)) \\
mult(x, h :: y) &= cadd(mult_one(x, h) <> zeroes(length(y)), mult(x, y), 0)
\end{aligned}$$

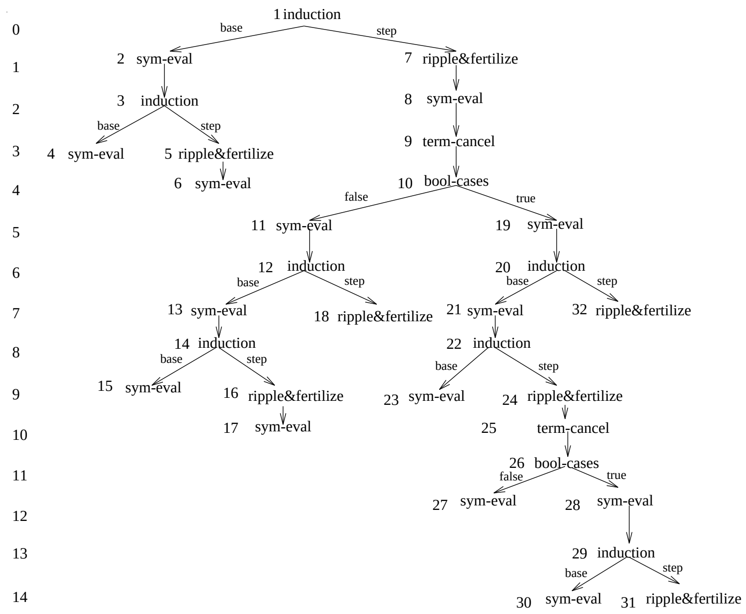

Figure 2: Proof plan for verifying an nm-bit multiplier

Here *cadd* is the definition of an n-bit adder with arguments of the same length, *mult_one* multiplies a word times a bit, <> appends two lists (built from nil and 'consed' together using ::), $zeroes(n)$ yields n zeroes. To show the equivalence of the specification and the implementation, we give *Clam* the conjecture

$$\forall x, y : word\ word2nat(x) \times word2nat(y) = word2nat(mult(x, y))$$

This theorem has been proof planned by *Clam* using the depth-first planner in about 1 minute. Figure 2 displays the structure of the proof plan. The plan requires 15 levels of planning (number on the left) and 32 plan steps (numbers on the nodes of the tree). Each node in the tree indicates the application of a method. In particular, steps 1, 3, 12, 14, 20, 22 and 29 correspond to the application of the induction method. We briefly explain steps 1, 2 and 7 corresponding to the first induction. In step 1, the induction method analyses the conjecture and available definitions and uses heuristics (similar to those used in *Nqthm*) to suggest an induction on the word y. In step 2, the *base case*,

$$word2nat(x) \times word2nat(nil) = word2nat(mult(x, nil))$$

is simplified by symbolic evaluation using the base equation of *mult*, *word2nat*, and multiplication by zero. This yields

$$0 = word2nat(zeroes(len(x)))$$

which is solved by another induction, symbolic evaluation, rippling, fertilise and another application of symbolic evaluation. In step 7, for the *step case*, the induction conclusion:

$$word2nat(x) \times word2nat(v0 :: v1) = word2nat(mult(x, v0 :: v1))$$

is simplified by rippling with the recursive equations of $mult$, the wave-rule obtained from the verification of the n-bit adder, $word2nat$, distributivity of times over plus, and fertilisation, to yield:

$$word2nat(x) \times (bitval(v0) \times 2^{length(v1)}) + word2nat(x) \times word2nat(v1)$$

$$=$$

$$word2nat(mult\_one(x, v0) <> zeroes(length(v1)))+$$
$$(word2nat(x) \times word2nat(v1) + bitval(false))$$

This equation is solved by the methods indicated in steps 8-32 in figure 2. In order to generate the proof plan, we provided wave-rules which correspond to the verification of the n-bit adder, distributivity of times over plus, associativity of times, and a non-definitional wave-rule of plus on its second argument. These wave-rules come from lemmas which we previously verified using proof planning. Hence, we use proof planning to develop, hierarchically, theories about hardware. Finally, we ask *Oyster* to execute our proof plan. This consists of executing a tactic for each of the methods indicated in the proof plan, following the structure displayed in figure 2 in a depth-first manner. The development of the proof plan took about 40 hours distributed over a two weeks period. The most time-consuming part was identifying the required lemmas.

Example: the Gordon computer

This is a 16-bit microprocessor, with 8 programming instructions, no interrupts, and a synchronous communication interface with memory, designed by Mike Gordon and his group at Cambridge University and verified interactively using the HOL system [12]. The specification is given in terms of the semantics of the 8 programming instructions. Each instruction consists of the set of operations that determines a new computer state, where a state is determined by the contents of the memory, the program counter, the accumulator and the idle/running status of the computer. The execution of an instruction defines a transition from a state into a new state and this transition determines the time-unit of an instruction-level time-scale. Thus, for each instruction we must specify the way in which each of the four components of a computer state are calculated. The implementation is at the register-transfer level. It consists of a data-path and a micro-programmed control unit. A computer state at the register-transfer level is determined by the contents of 11 components: the memory, the program counter, the accumulator, the idle/running flag, the memory address register, the instruction register, the argument register, the buffer register, the bus, the microcode program counter and the ready flag. The control unit generates the necessary control flags to update the computer registers. Communication between the bus and the registers is regulated by a set of gates. The implementation uses a microinstruction time-scale. The number of microinstructions required to compute a given instruction is calculated automatically by using the ready flag in the microcode, and the associated time at the microinstruction-level time-scale mapped onto the respective time at the instruction-level time-scale. A translation from the relational description used by Gordon into a functional description required by *Clam* was done by hand. This translation can be automated. For instance, PVS provides assistance in producing the functional representation from the relational one

[15]. The correctness theorem asserts that the state of the computer at the specification level is equal to an abstract state of the implementation level each time an instruction is executed. After doing the extensions explained in the next section, the verification proceeded without user intervention. See [9] for more details.

4 Experiments

We have applied the methodology just described to a variety of combinational and sequential circuits: some circuits are from the *IFIP WG10.2 Hardware Verification Benchmark Circuit Set* (n-bit adder, parallel multiplier, Gordon computer) and from other sources [14]. Table 1 displays some statistics. A detailed analysis of these proofs can be found in [9].

We shall explain here what these numbers measure. The first column lists the circuits. A parameterised representation means that there is an explicit parameter n corresponding to the length of a word. *Big-endian* means that the most significant bit is at the end of the list; *little-endian* means that the least significant bit is at the end of the list. Reasoning about big-endian representations when using lists to represent words is easier than the little-endian counterparts because in the big-endian case there is no need to traverse the list to access the least significant bits. In a word interpretation, the specification has type word, as the implementation does, so that the verification theorem establishes the equality of the specification and the implementation on the type word. The word arithmetic operations establish a relationship between the set of words and the set of the natural numbers.

The next column lists timings, broken down three ways: first, planning time, is the time *Clam* took to generate a plan [1]. The time to execute the proof plan is not included here. This time is in general much higher, mainly because *Oyster* the object-level theorem prover uses a complex type theory formalism with type checking proof obligations which are time consuming. For instance, generating the proof plan for the incrementer takes just 6 seconds, but its execution in *Oyster* took 5:40 minutes. Second, plan-development time, which is the time spent developing the proof plan, from understanding the problem, finding the right representation for the specification and the implementation, debugging them, and finding and justifying missing lemmas, until a proof plan was obtained. This time doesn't include the time spent extending, tuning, and debugging *Clam*, which is the Clam-development time displayed in the third column. Some of these times may appear excessive, but, to give an accurate picture, we are accounting for everything, including many one-time costs.

The Clam-development time column includes time spent writing new methods, extending existing methods, experimenting with method orderings, writing new predicates and modifying existing ones for the meta-language, finding new induction schemes, and debugging *Clam* itself. The time to obtain a proof tended to be high for the first circuit of a certain kind, but dramatically decreased for subsequent circuits. For instance, the n-bit adder was the first circuit we verified and took about 116 man-hours, of which approximately 100 are of work extending previous methods such as *fertilise* and *generalise*, writing new methods such as *bool_cases* and *term_cancellation*, experi-

[1]Experiments were done in a Solbourne 6/702 dual processor with 128Mb of memory, which is equivalent to a two-processor SparcStation 20

Circuit	time planning (min:sec)	plan-development (hours)	Clam-development (hours)	lemmas
COMBINATIONAL				
n-bit adder				
parameterised	4:50	16	100	0
word interpretation	15:50	8	16	0
little endian	5:30	8	16	2
big endian	0:57	4	0	1
look-ahead carry	2:40	8	0	1
n-bit alu				
parameterised	4:40	56	0	0
little endian	8:30	16	0	1
big endian	5:35	8	0	0
nat. number interpretation	4:50	8	0	0
n-bit shifter				
parameterised	4:30	32	30	0
big endian	3:20	16	0	0
n-bit processor unit				
big endian	5:00	8	2	0
parallel array				
nm-bit multiplier	1:03	40	0	6
n-bit incrementer				
little endian	0:58	4	0	1
big endian	0:06	2	0	1
word arithmetic				
addition	0:08	4	8	1
subtraction	0:10	4	0	2
multiplication	0:15	4	0	2
quotient	0:35	8	4	3
remainder	0:30	8	0	5
exponentiation	0:26	4	0	2
factorial	0:40	8	0	3
SEQUENTIAL				
n-bit counter	2:04	20	0	1
Gordon computer	45:00	200	360	0

Table 1: Some Circuits Verified using Proof Planning

menting with method ordering, and writing new predicates such as *find_type* to provide the proof planner with boolean type information. The rest of the circuits in the table utilised these extensions and their proofs were obtained in shorter times because these extensions were already there. The word-interpretation version, which took 16 hours, required 16 hours of extensions to the methods *generalise* and *normalise*. The little-endian version, required 16 hours to derive in *Oyster* a new induction scheme (double induction on two words of the same length) and this is used in many of the other verification proofs. The big-endian representation, which took just 4 hours, used all the previous extensions. When we tried the parameterised version of the n-bit alu, it turned out that all the extensions required were already done for the parameterised version of the n-bit adder including method ordering; the 56 hours reported were mainly spent debugging the specification. Thus, the time for the little-endian and big-endian versions became shorter (16 and 8 hours, respectively). For the word arithmetic the extensions required included deriving new induction schemes such as induction with increment of a word and addition of two words, which made the proofs very easy to find. The multiplier didn't require any extensions; most of the time was spent in finding the lemmas required by the proof.

The Gordon computer required a huge effort to scale-up *Clam* capabilities. The very scale of the specification required that we make a number of extensions to *Clam*, such as memoisation, so that the system would more gracefully handle large terms. Also significant is that the theorem involves two different time-scales with automatic calculation of the number of cycles for each instruction; methods like *difference_match* were designed to handle this and to automate time abstraction. Again, all these extensions are required for the verification of similar circuits, so the 360 hours of development time (9 man-weeks) is a one-time effort and the verification effort should significantly decrease for new circuits of the same kind. The 200 hours of plan-development time (5 man-weeks) include learning about microprocessor architecture, translating the original relational specification of the computer into a functional one, finding appropriate abstraction mechanisms, formalising recursive definitions for the implementation devices (memory, program counter, accumulator, microcode program counter, etc), and debugging the specification and the implementation.

The final column in our table indicates the number of lemmas used in the proof planning. The Gordon computer didn't require any lemmas, the multiplier requires lemmas about the n-bit adder, distributivity of $+$ over $\times$, and associativity of times. In general, proof planning utilises very few lemmas compared with most other systems.

5 Comparisons

We compare *Clam/Oyster* with *Nqthm*, *PVS* and *HOL*. *Clam/Oyster* is a *fully-expansive* system, so it provides the user with the security characteristic of these systems, as opposed to systems like *Nqthm* and *PVS* which don't necessarily incorporate this feature [2]. In a tactic-based fully expansive system, the execution of a tactic results in the execution of the inference rules that support that tactic. Theorem provers like *HOL* and *Nuprl* are also fully-expansive systems.

5.1 Nqthm

Most of the circuits in the table have been verified elsewhere using *Nqthm*. For instance, the n-bit adder was verified (big-endian) in half a man-day, where the discovery of the required lemmas was the most difficult part[17]. A combinational processing unit (alu and shifter) was verified by Warren Hunt as part of the verification of the FM8501 microprocessor. This processing unit is verified in 3 theorems corresponding to the word, natural number, and two's complement interpretations. It took about 2 months effort, runs in a few seconds[2], and used about 53 lemmas [11]. Although the processor unit reported here is less complicated than FM8501's because we don't include the two's complement interpretation, we use just 2 lemmas in its proof planning.

As an experiment, some of these circuits were re-implemented in *Nqthm* by a newcomer to formal verification [16]. The following table summarises the results:

Circuit	time run (min:sec)	time proof-development (hours)	lemmas
COMBINATIONAL			
n-bit adder (big endian)	3:40	32	6
n-bit adder look-ahead carry	3:20	12	6
n-bit alu (big endian)	13:30	40	11
n-bit shifter (big endian)	2:30	24	2
n-bit processor unit (big endian)	0:37	6	13
n-bit incrementer (big endian)	0:35	20	3

Most of the proof-development time was spent determining the lemmas required by the proof. For instance the proof of the n-bit adder uses the following lemmas: commutativity of plus, commutativity of times, and addition, and multiplication by recursion on the second argument. Of these, *Clam* requires only the lemma addition by recursion on the second argument. In general, *Clam* required fewer lemmas than *Nqthm* to verify these circuits. On the other hand, *Nqthm* provides a very stable implementation, and was much easier to use. There is of course a danger in such quantitative comparisons as metrics can oversimplify and even mislead. For example, the development times of novice users can be orders of magnitude higher than experts and moreover experts have better insights on how to structure problems to avoid lemmas. Still, we have tried to compare like with like: both the *Clam* and *Nqthm* proofs were carried out by relative beginners to both automated theorem proving and formal reasoning about hardware.

5.2 PVS

The n-bit adder, the n-bit alu, and the Tamarack microprocessor [3] have been implemented in *PVS* [10, 15]. The run time for verifying each of these circuits was 2:07, 1:27 and 9:05 minutes respectively in a Sun SparcStation 10. These low run-times are explained by the built-in decision procedures available to *PVS*. In these verifications the user must provide the induction parameters and use a predefined proof strategy. There are two

[2] Personal communication

[3] A refined implementation of the Gordon computer. Its verification in HOL and PVS is also more abstract, as tri-state values and gates to access the bus, and the input of manual information through the switches and the knob, are not considered. However, Tamarack-3 includes an option for asynchronous communication with memory.

ways in which *Clam* could enhance *PVS* automation facilities: (1) The decisions for selecting an induction scheme and induction variables are done by the user. These decisions could be automated by interfacing *Clam* and *PVS*. (2) There are proof strategies packaged as sets of *PVS* commands for a certain kind of circuits. The adder and the alu use the same proof strategy, the Tamarack and the pipelined Saxe microprocessor share another proof strategy. Our set of methods and the planning mechanism comprise these proof strategies, and can in principle create new ones by customising a composite tactic for a new conjecture using the same methods.

5.3 HOL

The *Gordon computer* was originally designed and verified using *HOL* by Mike Gordon and his group [12] and later implemented and verified as the Tamarack microprocessor by Jeffrey Joyce [13]. The verification took about 5 weeks of proof-development effort and required the derivation of at least 200 lemmas including general lemmas for arithmetic reasoning and temporal logic operators which are now built into *HOL*. It didn't require to tune *HOL* and runs in about 30 minutes in a modern machine[4]. Although the architecture of the *Gordon computer* verified here is less complicated than the architecture of Tamarack-3 verified by Joyce because we assume a synchronous communication with memory, we don't use lemmas in its proof planning. There are also two ways in which *Clam* could enhance *HOL* automation facilities: (1) the selection of induction parameters (scheme and variables), and (2) the generation of proof strategies for families of circuits. A project to interface *Clam* and *HOL* is about to start [5].

6 Conclusions

We have described how a hardware verification methodology based on proof planning can be applied to guide automatically a tactic-based theorem prover in verifying hardware designs. We have applied this technique to verify a variety of combinational and synchronous sequential circuits. Our experience shows that the *Clam* system and the proof planning idea carry over well to this new domain, although a number of extensions in the details (as opposed to the spirit) of *Clam* and the development of domain specific tactics and methods were required.

Overall, our experience was quite positive. We investigated several kinds of parameterised circuits and were able to develop methods which captured heuristics suitable for reasoning about families of such designs. We reported on our development time for both proving particular theorems and doing extensions to *Clam*. Although the times are sometimes high for initially verifying new kinds of circuits, subsequent development times were respectable and the majority of time was spent on simply entering and debugging the specification. This provides some support to our belief that a system like *Clam* might be usable by hardware engineers, provided that there is a 'proof engineer' in the background who has worked on the design of a set of methods appropriate for the domain.

There are several directions for further work. As noted, much of our time was spent entering and debugging specifications. Part of this is due to our use of a somewhat

[4] Personal communication

complicated type-theory as a specification language. However, our planning approach should work with any tactic based theorem prover; one need only port the tactics associated with the current methods from one system to another so that they produce the same effects. Interfacing *Clam* with a prover like HOL could significantly reduce specification and tuning times. It would also provide us immediate access to the large collection of tactics and theorems already available for this system. An interface to a standard hardware description language would also ease specification and make it possible to integrate better proof planning into the hardware design cycle. Another area for further work concerns improving the power or efficiency of some of our methods. For example, reasoning about boolean circuits by case evaluation, could be replaced by a more efficient routine based on BDDs. We also will investigate the development of a temporal reasoning methods for reasoning about circuits with asynchronous interfaces. Finally, we plan to apply our approach to the verification of commercial circuits including microprocessors systems.

References

[1] David Basin and Toby Walsh. A calculus for and termination of rippling. Technical report, MPI, 1994. To appear in special issue of the Journal of Automated Reasoning.

[2] Richard J. Boulton. Efficiency in a fully-expansive theorem prover. Technical Report 337, University of Cambridge Computer Laboratory, 1994.

[3] R.S. Boyer and J.S. Moore. *A Computational Logic*. Academic Press, 1979. ACM monograph series.

[4] A. Bundy. The use of explicit plans to guide inductive proofs. In R. Lusk and R. Overbeek, editors, *9th Conference on Automated Deduction*, pages 111–120. Springer-Verlag, 1988. Longer version available from Edinburgh as DAI Research Paper No. 349.

[5] A. Bundy and M. Gordon. Automatic guidance of mechanically generated proofs. Research proposal, Edinburgh-Cambridge, 1995.

[6] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993. Also available from Edinburgh as DAI Research Paper No. 567.

[7] A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7:303–324, 1991. Earlier version available from Edinburgh as DAI Research Paper No 413.

[8] A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In M.E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 647–648. Springer-Verlag, 1990. Lecture Notes in Artificial Intelligence No. 449. Also available from Edinburgh as DAI Research Paper 507.

[9] Francisco J. Cantu. *Inductive Proof Planning for Automating Hardware Verification*. PhD thesis, University of Edinburgh, 1996. Forthcoming.

[10] D. Cyrluk, N. Rajan, N. Shankar, and M.K. Srivas. Effective theorem proving for hardware verification. In *2nd Conference on Theorem Provers in Circuit Design.* Springer-Verlag, 1994.

[11] Warren Hunt. Fm8501: A verified microprocessor. TechReport 47, Institute for Computing Science, University of Texas at Austin, 1986.

[12] Jeff Joyce, G. Graham Birtwistle, and M. Gordon. Proving a computer correct in higher order logic. Technical Report 100, University of Cambridge Computer Laboratory, 1986.

[13] Jeffrey J. Joyce. Multi-level verification of microprocessor-based systems. Technical Report 195, University of Cambridge Computer Laboratory, 1990.

[14] M. Morris Mano. *Digital Logic and Computer Design.* Prentice Hall, Inc, 1979.

[15] S. Owre, J.M. Rushby, N. Shankar, and M.K. Srivas. A tutorial on using pvs for hardware verification. In *2nd Conference on Theorem Provers in Circuit Design.* Springer-Verlag, 1994.

[16] Victor Rangel. Metodos formales para verificacion de hardware: Un estudio comparativo. Master's thesis, Instituto Tecnologico de Monterrey, Mexico, 1996.

[17] V. Stavridou, H. Barringer, and D.A. Edwards. Formal specification and verification of hardware: A comparative case study. In *Proceedings of the 25th ACM/IEEE Design Automation Conference*, pages 89–96. IEEE, 1988.